Distributed Programming with Ice

Michi Henning
Mark Spruiell

With contributions by

Dwayne Boone, Brent Eagles, Benoit Foucher,
Marc Laukien, Matthew Newhook, Bernard Normier

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and ZeroC was aware of the trademark claim, the designations have been printed in initial caps or all caps. The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

This manual is provided under one of two licenses, whichever you prefer:

• the Creative Commons Attribution-No Derivative Works 3.0 Unported License (http://creativecommons.org/licenses/by-nd/3.0/). This license does not permit you make modifications.

• the Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License (http://creativecommons.org/licenses/by-nc-sa/3.0/). This license permits you to make modifications. If you distribute this manual under this license, you must prominently include the following text:

This document is derived from "Distributed Computing with Ice", Copyright © ZeroC, Inc. 2003-2010. You can find the latest version of "Distributed Computing with Ice" at http://www.zeroc.com/Ice-Manual.pdf.

Copyright © 2003-2010 by ZeroC, Inc.
mailto:info@zeroc.com
http://www.zeroc.com

Revision 3.4, June 2010

This revision of the documentation describes Ice version 3.4.1.

The Ice source distribution makes use of a number of third-party products:

• Berkeley DB, developed by Oracle (http://www.oracle.com)

• bzip2/libbzip2, developed by Julian R. Seward (http://sources.redhat.com/bzip2)

• The OpenSSL Toolkit, developed by the OpenSSL Project (http://www.openssl.org)

• SSLeay, developed by Eric Young (mailto:eay@cryptsoft.com)

• Expat, developed by James Clark (http://www.libexpat.org)

• STLport, developed by the STLport Standard Library Project (http://www.stlport.org)

• mcpp, developed by Kiyoshi Matsui (http://mcpp.sourceforge.net)

• QT, developed by Nokia (http://qt.nokia.com)

See the Ice source distribution for the license agreements for each of these products.

Contents

Chapter 1 Introduction **1**
1.1 The Internet Communications Engine (Ice) 1
1.2 Organization 2
1.3 Typographical Conventions 3
1.4 Source Code Examples 4
1.5 Contacting the Authors 4
1.6 Ice Support 4

Part I Ice Overview 5

Chapter 2 Ice Overview **7**
2.1 Chapter Overview 7
2.2 The Ice Architecture 7
2.3 Ice Services 23
2.4 Architectural Benefits of Ice 26

Chapter 3 A Hello World Application **29**
3.1 Chapter Overview 29
3.2 Writing a Slice Definition 30
3.3 Writing an Ice Application with C++ 30
3.4 Writing an Ice Application with Java 39
3.5 Writing an Ice Application with C# 46
3.6 Writing an Ice Application with Visual Basic 53
3.7 Writing an Ice Application with Objective-C 61
3.8 Writing an Ice Application with Python 70
3.9 Writing an Ice Application with Ruby 75
3.10 Writing an Ice Application with PHP 78
3.11 Summary 81

Part II Slice 83

Chapter 4 The Slice Language 85

4.1 Chapter Overview 85
4.2 Introduction 85
4.3 Compilation 86
4.4 Source Files 89
4.5 Lexical Rules 91
4.6 Modules 94
4.7 The Ice Module 95
4.8 Basic Slice Types 96
4.9 User-Defined Types 98
4.10 Interfaces, Operations, and Exceptions 105
4.11 Classes 131
4.12 Forward Declarations 148
4.13 Type IDs 149
4.14 Operations on `Object` 150
4.15 Local Types 151
4.16 Names and Scoping 153
4.17 Metadata 159
4.18 Serializable Objects 160
4.19 Deprecating Slice Definitions 162
4.20 Using the Slice Compilers 163
4.21 Slice Checksums 164
4.22 Generating Slice Documentation 165
4.23 Summary 171

Chapter 5 Slice for a Simple File System 173

5.1 Chapter Overview 173
5.2 The File System Application 173
5.3 Slice Definitions for the File System 174
5.4 The Complete Definition 176

Part III C++ Mapping 179

Chapter 6 Client-Side Slice-to-C++ Mapping 181
6.1 Chapter Overview 181
6.2 Introduction 181
6.3 Mapping for Identifiers 182
6.4 Mapping for Modules 183
6.5 The `Ice` Namespace 184
6.6 Mapping for Simple Built-In Types 184
6.7 Mapping for User-Defined Types 186
6.8 Mapping for Constants 196
6.9 Mapping for Exceptions 197
6.10 Mapping for Run-Time Exceptions 201
6.11 Mapping for Interfaces 202
6.12 Mapping for Operations 212
6.13 Exception Handling 218
6.14 Mapping for Classes 220
6.15 Asynchronous Method Invocation (AMI) 244
6.16 `slice2cpp` Command-Line Options 260
6.17 Using Slice Checksums 266

Chapter 7 Developing a File System Client in C++ 267
7.1 Chapter Overview 267
7.2 The C++ Client 267
7.3 Summary 272

Chapter 8 Server-Side Slice-to-C++ Mapping 273
8.1 Chapter Overview 273
8.2 Introduction 273
8.3 The Server-Side `main` Function 274
8.4 Mapping for Interfaces 290
8.5 Parameter Passing 293
8.6 Raising Exceptions 294
8.7 Object Incarnation 295
8.8 Asynchronous Method Dispatch (AMD) 300
8.9 Summary 306

Chapter 9 Developing a File System Server in C++ 307
9.1 Chapter Overview 307
9.2 Implementing a File System Server 307
9.3 Summary 324

Part IV Java Mapping 327

Chapter 10 Client-Side Slice-to-Java Mapping 329
10.1 Chapter Overview 329
10.2 Introduction 329
10.3 Mapping for Identifiers 330
10.4 Mapping for Modules 331
10.5 The `Ice` Package 332
10.6 Mapping for Simple Built-in Types 332
10.7 Mapping for User-Defined Types 332
10.8 Mapping for Constants 336
10.9 Mapping for Exceptions 337
10.10 Mapping for Run-Time Exceptions 340
10.11 Mapping for Interfaces 341
10.12 Mapping for Operations 351
10.13 Exception Handling 356
10.14 Mapping for Classes 358
10.15 Serializable Objects 367
10.16 Customizing the Java Mapping 368
10.17 Asynchronous Method Invocation (AMI) 379
10.18 Slice Compiler 392
10.19 Using Slice Checksums 397

Chapter 11 Developing a File System Client in Java 399
11.1 Chapter Overview 399
11.2 The Java Client 399
11.3 Summary 403

Chapter 12 Server-Side Slice-to-Java Mapping 405
12.1 Chapter Overview 405
12.2 Introduction 405
12.3 The Server-Side `main` Method 406
12.4 Mapping for Interfaces 413
12.5 Parameter Passing 416
12.6 Raising Exceptions 417
12.7 Tie Classes 418
12.8 Object Incarnation 422
12.9 Asynchronous Method Dispatch (AMD) 426
12.10 Summary 431

Chapter 13 Developing a File System Server in Java 433
13.1 Chapter Overview 433
13.2 Implementing a File System Server 433
13.3 Summary 442

Part V C# Mapping 443

Chapter 14 Client-Side Slice-to-C# Mapping 445
14.1 Chapter Overview 445
14.2 Introduction 445
14.3 Mapping for Identifiers 446
14.4 Mapping for Modules 447
14.5 The `Ice` Namespace 448
14.6 Mapping for Simple Built-in Types 448
14.7 Mapping for User-Defined Types 449
14.8 Mapping for Constants 467
14.9 Mapping for Exceptions 469
14.10 Mapping for Interfaces 472
14.11 Mapping for Operations 480
14.12 Exception Handling 484
14.13 Mapping for Classes 486
14.14 Serializable Objects 497
14.15 C#-Specific Metadata Directives 498
14.16 Asynchronous Method Invocation (AMI) 498
14.17 `slice2cs` Command-Line Options 513
14.18 Using Slice Checksums 513

Chapter 15 Developing a File System Client in C# 517
15.1 Chapter Overview 517
15.2 The C# Client 517
15.3 Summary 521

Chapter 16 Server-Side Slice-to-C# Mapping 523
16.1 Chapter Overview 523
16.2 Introduction 523
16.3 The Server-Side `Main` Method 524
16.4 Mapping for Interfaces 530
16.5 Parameter Passing 533
16.6 Raising Exceptions 534
16.7 Tie Classes 536
16.8 Object Incarnation 539
16.9 Asynchronous Method Dispatch (AMD) 543
16.10 Summary 548

Chapter 17 Developing a File System Server in C# 549
17.1 Chapter Overview 549
17.2 Implementing a File System Server 549
17.3 Summary 558

Part VI Objective-C Mapping 559

Chapter 18 Client-Side Slice-to-Objective-C Mapping 561
18.1 Chapter Overview 561
18.2 Introduction 561
18.3 Mapping for Modules 562
18.4 The `ICE` Prefix 564
18.5 Mapping for Identifiers 564
18.6 Internal Identifiers 565
18.7 Mapping for Built-In Types 566
18.8 Mapping for User-Defined Types 567
18.9 Mapping for Constants 575
18.10 Mapping for Exceptions 576
18.11 Mapping for Interfaces 583
18.12 Mapping for Operations 589
18.13 Exception Handling 597
18.14 Mapping for Local Interfaces 599
18.15 Mapping for Classes 600
18.16 Interfaces by Value 611
18.17 Asynchronous Method Invocation (AMI) 612
18.18 `slice2objc` Command-Line Options 619

Chapter 19 Developing a File System Client in Objective-C 621
19.1 Chapter Overview 621
19.2 The Objective-C Client 621
19.3 Summary 626

Chapter 20 Server-Side Slice-to-Objective-C Mapping 627
20.1 Chapter Overview 627
20.2 Introduction 627
20.3 The Server-Side `main` Function 628
20.4 Mapping for Interfaces 631
20.5 Parameter Passing 635
20.6 Raising Exceptions 637
20.7 Object Incarnation 638
20.8 Summary 642

Chapter 21 Developing a File System Server in Objective-C 643
21.1 Chapter Overview 643
21.2 Implementing a File System Server 643
21.3 Summary 655

Part VII Python Mapping 657

Chapter 22 Client-Side Slice-to-Python Mapping 659
22.1 Chapter Overview 659
22.2 Introduction 659
22.3 Mapping for Identifiers 660
22.4 Mapping for Modules 661
22.5 The `Ice` Module 661
22.6 Mapping for Simple Built-In Types 661
22.7 Mapping for User-Defined Types 663
22.8 Mapping for Constants 668
22.9 Mapping for Exceptions 669
22.10 Mapping for Run-Time Exceptions 671
22.11 Mapping for Interfaces 672
22.12 Mapping for Operations 678
22.13 Exception Handling 683
22.14 Mapping for Classes 684
22.15 Asynchronous Method Invocation (AMI) 691
22.16 Code Generation 701
22.17 Using Slice Checksums 711

Chapter 23 Developing a File System Client in Python 713
23.1 Chapter Overview 713
23.2 The Python Client 713
23.3 Summary 717

Chapter 24 Server-Side Slice-to-Python Mapping 719
24.1 Chapter Overview 719
24.2 Introduction 719
24.3 The Server-Side Main Program 720
24.4 Mapping for Interfaces 726
24.5 Parameter Passing 729
24.6 Raising Exceptions 730
24.7 Object Incarnation 731
24.8 Asynchronous Method Dispatch (AMD) 735
24.9 Summary 740

Chapter 25 Developing a File System Server in Python 741
25.1 Chapter Overview 741
25.2 Implementing a File System Server 741
25.3 Thread Safety 748
25.4 Summary 749

Part VIII Ruby Mapping 751

Chapter 26 Client-Side Slice-to-Ruby Mapping 753
26.1 Chapter Overview 753
26.2 Introduction 753
26.3 Mapping for Identifiers 754
26.4 Mapping for Modules 755
26.5 The `Ice` Module 755
26.6 Mapping for Simple Built-In Types 755
26.7 Mapping for User-Defined Types 756
26.8 Mapping for Constants 761
26.9 Mapping for Exceptions 762
26.10 Mapping for Run-Time Exceptions 764
26.11 Mapping for Interfaces 764
26.12 Mapping for Operations 771
26.13 Exception Handling 775
26.14 Mapping for Classes 777
26.15 Code Generation 786
26.16 The `main` Program 791
26.17 Using Slice Checksums 797

Chapter 27 Developing a File System Client in Ruby 799
27.1 Chapter Overview 799
27.2 The Ruby Client 799
27.3 Summary 803

Part IX PHP Mapping 805

Chapter 28 Client-Side Slice-to-PHP Mapping 807
28.1 Chapter Overview 807
28.2 Introduction 807
28.3 Mapping for Identifiers 808
28.4 Mapping for Modules 808
28.5 The `Ice` Module 809
28.6 Mapping for Simple Built-In Types 810
28.7 Mapping for User-Defined Types 811
28.8 Mapping for Constants 814
28.9 Mapping for Exceptions 815
28.10 Mapping for Run-Time Exceptions 816
28.11 Mapping for Interfaces 817
28.12 Mapping for Operations 825
28.13 Exception Handling 828
28.14 Mapping for Classes 829
28.15 `slice2php` Command-Line Options 837
28.16 Application Notes 838
28.17 Using Slice Checksums 846

Chapter 29 Developing a File System Client in PHP 849
29.1 Chapter Overview 849
29.2 The PHP Client 849
29.3 Summary 853

Part X Advanced Ice 855

Chapter 30 Ice Properties and Configuration 857
30.1 Chapter Overview 857
30.2 Properties 857
30.3 Configuration Files 859
30.4 Setting Properties on the Command Line 862
30.5 Using Configuration Files 862
30.6 Alternate Property Stores 865
30.7 Command-Line Parsing and Initialization 866
30.8 The `Ice.ProgramName` property 869
30.9 Using Properties Programmatically 869
30.10 Unused Properties 879
30.11 Summary 879

Chapter 31 Threads and Concurrency with C++ 881
31.1 Chapter Overview 881
31.2 Introduction 881
31.3 Library Overview 882
31.4 Mutexes 882
31.5 Recursive Mutexes 889
31.6 Monitors 892
31.7 Condition Variables 901
31.8 Threads 905
31.9 Priority Inversion 915
31.10 Portable Signal Handling 916
31.11 Summary 917

Chapter 32 The Ice Run Time in Detail 919
32.1 Introduction 919
32.2 Communicators 920
32.3 Communicator Initialization 925
32.4 Object Adapters 927
32.5 Object Identity 941
32.6 The `Ice::Current` Object 945
32.7 Servant Locators 946
32.8 Default Servants 963
32.9 Server Implementation Techniques 969
32.10 The Ice Threading Model 1003
32.11 Proxies 1028
32.12 The `Ice::Context` Parameter 1042
32.13 Connection Timeouts 1051
32.14 Oneway Invocations 1053
32.15 Datagram Invocations 1058
32.16 Batched Invocations 1060
32.17 Location Services 1063
32.18 Administrative Facility 1072
32.19 The `Ice::Logger` Interface 1081
32.20 The `Ice::Stats` Interface 1090
32.21 Location Transparency 1092
32.22 Automatic Retries 1094
32.23 Dispatch Interceptors 1100
32.24 C++ Strings and Character Encoding 1105
32.25 Developing a Plug-In 1113
32.26 Custom Class Loaders 1119
32.27 Summary 1120

Chapter 33 Facets and Versioning 1121
33.1 Introduction 1121
33.2 Concept and APIs 1121
33.3 The Versioning Problem 1128
33.4 Versioning with Facets 1134
33.5 Facet Selection 1134
33.6 Behavioral Versioning 1136
33.7 Design Considerations 1138
33.8 Summary 1140

Chapter 34 Object Life Cycle 1141
34.1 Chapter Overview 1141
34.2 Introduction 1142
34.3 Object Existence and Non-Existence 1143
34.4 Life Cycle of Proxies, Servants, and Ice Objects 1148
34.5 Object Creation 1150
34.6 Object Destruction 1154
34.7 Removing Cyclic Dependencies 1171
34.8 Object Identity and Uniqueness 1178
34.9 Object Life Cycle for the File System Application 1180
34.10 Avoiding Server-Side Garbage 1198
34.11 Summary 1208

Chapter 35 Dynamic Ice 1209
35.1 Chapter Overview 1209
35.2 Streaming Interface 1209
35.3 Dynamic Invocation and Dispatch 1249
35.4 Asynchronous Dynamic Invocation and Dispatch 1266
35.5 Summary 1273

Chapter 36 Connection Management 1275
36.1 Chapter Overview 1275
36.2 Introduction 1275
36.3 Connection Establishment 1276
36.4 Active Connection Management 1281
36.5 Obtaining a Connection 1282
36.6 Connection Closure 1290
36.7 Bidirectional Connections 1291
36.8 Summary 1296

Chapter 37 The Ice Protocol 1297
37.1 Chapter Overview 1297
37.2 Data Encoding 1297
37.3 Protocol Messages 1322
37.4 Compression 1332
37.5 Protocol and Encoding Versions 1334

Part XI Ice Services 1339

Chapter 38 IceGrid 1341

38.1 Chapter Overview 1341
38.2 Introduction 1342
38.3 IceGrid Architecture 1344
38.4 Getting Started 1348
38.5 Using Deployment 1353
38.6 Well-known Objects 1362
38.7 Templates 1371
38.8 IceBox Integration 1377
38.9 Object Adapter Replication 1380
38.10 Load Balancing 1383
38.11 Sessions 1387
38.12 Registry Replication 1394
38.13 Application Distribution 1399
38.14 Administrative Sessions 1407
38.15 Glacier2 Integration 1414
38.16 Using an SQL Database 1418
38.17 XML Reference 1419
38.18 Variable and Parameter Semantics 1449
38.19 Property Set Semantics 1455
38.20 XML Features 1460
38.21 Server Reference 1463
38.22 Administrative Facility Integration 1472
38.23 Securing IceGrid 1481
38.24 Administrative Utilities 1486
38.25 Server Activation 1494
38.26 Solving Problems 1498
38.27 Summary 1501

Chapter 39 Freeze 1503

39.1 Chapter Overview 1503
39.2 Introduction 1504
39.3 Freeze Evictors 1504
39.4 Using the Freeze Evictor in a File System Server 1521
39.5 The Freeze Map 1541
39.6 Using a Freeze Map in the File System Server 1576
39.7 The Freeze Catalog 1602
39.8 Backups 1603
39.9 Summary 1604

Chapter 40 FreezeScript 1605
40.1 Chapter Overview 1605
40.2 Introduction 1605
40.3 Database Migration 1606
40.4 Transformation Descriptors 1612
40.5 Using **`transformdb`** 1626
40.6 Database Inspection 1634
40.7 Using **`dumpdb`** 1645
40.8 Descriptor Expression Language 1650
40.9 Summary 1653

Chapter 41 IceSSL 1655
41.1 Chapter Overview 1655
41.2 Introduction 1655
41.3 Using IceSSL 1658
41.4 Configuring IceSSL 1661
41.5 Programming with IceSSL 1675
41.6 Advanced Topics 1690
41.7 Setting up a Certificate Authority 1698
41.8 Summary 1704

Chapter 42 Glacier2 1705
42.1 Chapter Overview 1705
42.2 Introduction 1705
42.3 Using Glacier2 1710
42.4 Callbacks 1717
42.5 Helper Classes 1720
42.6 Router Security 1729
42.7 Session Management 1738
42.8 Dynamic Filtering 1742
42.9 Request Buffering 1744
42.10 Request Contexts 1745
42.11 Firewalls 1747
42.12 Advanced Client Configurations 1748
42.13 IceGrid Integration 1750
42.14 Summary 1751

Chapter 43 IceBox 1753
43.1 Chapter Overview 1753
43.2 Introduction 1753
43.3 Developing a Service 1754
43.4 Starting IceBox 1763
43.5 IceBox Administration 1764
43.6 Summary 1769

Chapter 44 IceStorm 1771
44.1 Chapter Overview 1771
44.2 Introduction 1771
44.3 Concepts 1773
44.4 IceStorm Interface Overview 1776
44.5 Using IceStorm 1778
44.6 Publishing to a Specific Subscriber 1789
44.7 Highly Available IceStorm 1791
44.8 IceStorm Administration 1796
44.9 Topic Federation 1798
44.10 Quality of Service 1802
44.11 Delivery Mode 1804
44.12 Configuring IceStorm 1806
44.13 Summary 1812

Chapter 45 IcePatch2 1813
45.1 Chapter Overview 1813
45.2 Introduction 1813
45.3 Using `icepatch2calc` 1814
45.4 Running the Server 1817
45.5 Running the Client 1818
45.6 Object Identities 1821
45.7 The IcePatch2 Client Utility Library 1821
45.8 Summary 1826

Appendixes 1827

Appendix A Slice Keywords 1829

Appendix B Slice Metadata Directives 1831
B.1 General Metadata Directives 1831
B.2 Metadata Directives for C++ 1832
B.3 Metadata Directives for Java 1835
B.4 Metadata Directives for C# 1836
B.5 Metadata Directives for .NET and Mono 1836
B.6 Metadata Directives for Objective-C 1837
B.7 Metadata Directives for Python 1837
B.8 Metadata Directives for Freeze 1838

Appendix C Slice API Reference 1839

Appendix D Properties 1841
D.1 Ice Configuration Property 1841
D.2 Ice Trace Properties 1842
D.3 Ice Warning Properties 1845
D.4 Ice Object Adapter Properties 1848
D.5 Ice Administrative Properties 1853
D.6 Ice Plug-In Properties 1855
D.7 Ice Thread Pool Properties 1858
D.8 Ice Default and Override Properties 1861
D.9 Ice Proxy Properties 1866
D.10 Ice Transport Properties 1869
D.11 Ice Miscellaneous Properties 1871
D.12 IceSSL Properties 1881
D.13 IceBox Properties 1898
D.14 IceBoxAdmin Properties 1902
D.15 IceGrid Properties 1902
D.16 IceGrid Administrative Client Properties 1923
D.17 IceStorm Properties 1925
D.18 Glacier2 Properties 1935
D.19 Freeze Properties 1950
D.20 IcePatch2 Properties 1958

Appendix E Proxies and Endpoints 1961
E.1 Proxies 1961
E.2 Endpoints 1963

Appendix F The C++ Utility Library 1973
F.1 Introduction 1973
F.2 `AbstractMutex` 1973
F.3 `Cache` 1976
F.4 `CtrlCHandler` 1979
F.5 `Exception` 1980
F.6 `generateUUID` 1980
F.7 `Handle` Template 1981
F.8 `Handle` Template Adaptors 1984
F.9 `ScopedArray` 1989
F.10 `Shared` and `SimpleShared` 1990
F.11 Threads and Synchronization Primitives 1991
F.12 `Time` 1991
F.13 `Timer` and `TimerTask` 1995
F.14 Unicode and UTF-8 Conversion Functions 1998
F.15 Version Information 1999

Appendix G The Java Utility Library 2001
G.1 Introduction 2001
G.2 The `IceUtil` Package 2001
G.3 The Ice.Util Class 2004

Appendix H Windows Services 2007
H.1 Introduction 2007
H.2 Installing a Windows Service 2008
H.3 The Ice Service Installer 2008
H.4 Manual Installation 2014
H.5 Troubleshooting 2022

Appendix I The .NET Utility Library 2025
I.1 Introduction 2025
I.2 Communicator Initialization Methods 2025
I.3 Identity Conversion 2025
I.4 Property Creation Methods 2026
I.5 Proxy Comparison Methods 2026
I.6 Stream Creation 2026
I.7 UUID Generation 2026
I.8 Version Information 2026

Appendix J Binary Distributions 2029
J.1 Introduction 2029
J.2 Developer Kits 2029
J.3 Guidelines 2030

Appendix K Deprecated AMI Mapping **2035**
K.1 Chapter Overview 2035
K.2 Introduction 2035
K.3 Using AMI 2038
K.4 Language Mappings 2042
K.5 Summary 2058

Bibliography **2059**

Chapter 1
Introduction

1.1 The Internet Communications Engine (Ice)

The rise of object-oriented middleware in the mid-nineties was an important step forward toward making distributed computing available to application developers. For the first time, it was possibe to build distributed applications without having to be a networking guru: the middleware platform took care of the majority of networking chores, such as *marshaling* and *unmarshaling* (encoding and decoding data for transmission), mapping logical object addresses to physical transport endpoints, changing the representation of data according to the native machine architecture of client and server, and automatically starting servers on demand.

Despite these advances, the leading object-oriented middleware platforms suffered from a number of serious practical limitations that prompted ZeroC to develop the Internet Communications Engine, or Ice for short.[1] The main design goals of Ice are:

- Provide an object-oriented middleware platform suitable for use in heterogeneous environments.
- Provide a full set of features that support development of realistic distributed applications for a wide variety of domains.

1. The acronym "Ice" is pronounced as a single syllable, like the word for frozen water.

- Avoid unnecessary complexity, making the platform easy to learn and to use.
- Provide an implementation that is efficient in network bandwidth, memory use, and CPU overhead.
- Provide an implementation that has built-in security, making it suitable for use over insecure public networks.

To be more simplistic, the Ice design goals could be stated as "Let's build a more powerful middleware platform that makes the developer's life easier and avoids the mistakes of its predecessors."

1.2 Organization

This book is divided into four parts and a number of appendixes:

- Part I: *Ice Overview* provides an overview of the features offered by Ice and explains the Ice object model. After reading this part, you will understand the major features and architecture of the Ice platform, its object model and request dispatch model, and know the basic steps required to build a simple application in C++, Java, C#, Visual Basic, Objective-C, Python, and Ruby.
- Part II: *Slice* explains the Slice definition language. After reading this part, you will have detailed knowledge of how to specify interfaces and types for a distributed application.
- Part III through Part IX cover language mappings. Each section shows how to implement an application in your language of choice.
- Part X: *Advanced Ice* presents many Ice features in detail and covers advanced aspects of server development, such as properties, threading, object life cycle, object location, persistence, and asynchronous as well as dynamic method invocation and dispatch. After reading this part, you will understand the advanced features of Ice and how to effectively use them to find the correct trade-off between performance and resource consumption as appropriate for your application requirements.
- Part XI: *Ice Services* covers the services provided with Ice, such as IceGrid (a sophisticated deployment tool), Glacier2 (the Ice firewall solution), IceStorm (the Ice messaging service), and IcePatch2 (a software patching service).[2]
- *Appendixes* contain Ice reference material and explain the feature differences between Ice and Ice-E (the version of Ice for embedded systems).

NOTE: This entire manual is also available online as a set of HTML pages at http://www.zeroc.com/doc/Ice-3.4.1/manual.

You can always find the latest version of the manual at http://www.zeroc.com/Ice-Manual.html.

In addition, you can find an online reference of all the Slice APIs that are used by Ice and its services at http://www.zeroc.com/doc/Ice-3.4.1/reference.

You can always find the latest version of this reference at http://www.zeroc.com/Slice-Reference.html.

1.3 Typographical Conventions

This book uses the following typographical conventions:

- Slice source code appears in `monospace`.
- Programming-language source code appears in `monospace`.
- File names appear in `monospace`.
- Commands appear in **`monospace bold`**.

Occasionally, we present copy of an interactive session at a terminal. In such cases, we assume a Bourne shell (or one of its derivatives, such as **`ksh`** or **`bash`**). Output presented by the system is shown in `monospace`, and input is presented in **`monospace bold`**, for example:

```
$ echo hello
hello
```

Slice and the various programming languages often use the same identifiers. When we talk about an identifier in its generic, language-independent sense, we use `monospace`. When we talk about an identifier in its language-specific (for example, C++ or Java) sense, we use `monospace`.

2. If you notice a certain commonality in the theme of naming Ice features, it just goes to show that software developers are still inveterate punsters.

1.4 Source Code Examples

Throughout the book, we use a case study to illustrate various aspects of Ice. The case study implements a simple distributed hierarchical file system, which we progressively improve to take advantage of more sophisticated features as the book progresses. The source code for the case study in its various stages is provided with the distribution of this book. We encourage you to experiment with these code examples (as well as the many demonstration programs that ship with Ice).

1.5 Contacting the Authors

We would very much like to hear from you in case you find any bugs (however minor) in this book. We also would like to hear your opinion on the contents, and any suggestions as to how it might be improved. You can contact us via e-mail at mailto:icebook@zeroc.com.

1.6 Ice Support

If you have a question and you cannot find an answer in this manual, you can visit our developer forums at http://www.zeroc.com/forums to see if another developer has encountered the same issue. If you still need help, feel free to post your question on the forum, which ZeroC's developers monitor regularly. Note, however, that we can provide only limited free support in our forums. For guaranteed response and problem resolution times, we highly recommend purchasing commercial support.

Part I

Ice Overview

Chapter 2
Ice Overview

2.1 Chapter Overview

In this chapter, we present a high-level overview of the Ice architecture. Section 2.2 introduces fundamental concepts and terminology, and outlines how Slice definitions, language mappings, and the Ice run time and protocol work in concert to create clients and servers. Section 2.3 briefly presents the object services provided by Ice, and Section 2.4 outlines the benefits that result from the Ice architecture.

2.2 The Ice Architecture

2.2.1 Introduction

Ice is an object-oriented middleware platform. Fundamentally, this means that Ice provides tools, APIs, and library support for building object-oriented client–server applications. Ice applications are suitable for use in heterogeneous environments: client and server can be written in different programming languages, can run on different operating systems and machine architectures, and can communicate using a variety of networking technologies. The source code for these applications is portable regardless of the deployment environment.

2.2.2 Terminology

Every computing technology creates its own vocabulary as it evolves. Ice is no exception. However, the amount of new jargon used by Ice is minimal. Rather than inventing new terms, we have used existing terminology as much as possible. If you have used another middleware technology in the past, you will be familiar with much of what follows. (However, we suggest you at least skim the material because a few terms used by Ice *do* differ from the corresponding terms used by other middleware.)

Clients and Servers

The terms *client* and *server* are not firm designations for particular parts of an application; rather, they denote roles that are taken by parts of an application for the duration of a request:

- Clients are active entities. They issue requests for service to servers.
- Servers are passive entities. They provide services in response to client requests.

Frequently, servers are not "pure" servers, in the sense that they never issue requests and only respond to requests. Instead, servers often act as a server on behalf of some client but, in turn, act as a client to another server in order to satisfy their client's request.

Similarly, clients often are not "pure" clients, in the sense that they only request service from an object. Instead, clients are frequently client–server hybrids. For example, a client might start a long-running operation on a server; as part of starting the operation, the client can provide a *callback object* to the server that is used by the server to notify the client when the operation is complete. In that case, the client acts as a client when it starts the operation, and as a server when it is notified that the operation is complete.

Such role reversal is common in many systems, so, frequently, client–server systems could be more accurately described as *peer-to-peer* systems.

Ice Objects

An *Ice object* is a conceptual entity, or abstraction. An Ice object can be characterized by the following points:

- An Ice object is an entity in the local or a remote address space that can respond to client requests.

- A single Ice object can be instantiated in a single server or, redundantly, in multiple servers. If an object has multiple simultaneous instantiations, it is still a single Ice object.
- Each Ice object has one or more *interfaces*. An interface is a collection of named *operations* that are supported by an object. Clients issue requests by invoking operations.
- An operation has zero or more *parameters* as well as a *return value*. Parameters and return values have a specific *type*. Parameters are named and have a direction: in-parameters are initialized by the client and passed to the server; out-parameters are initialized by the server and passed to the client. (The return value is simply a special out-parameter.)
- An Ice object has a distinguished interface, known as its *main interface*. In addition, an Ice object can provide zero or more alternate interfaces, known as *facets*. Clients can select among the facets of an object to choose the interface they want to work with.
- Each Ice object has a unique *object identity*. An object's identity is an identifying value that distinguishes the object from all other objects. The Ice object model assumes that object identities are globally unique, that is, no two objects within an Ice communication domain can have the same object identity.

  In practice, you need not use object identities that are globally unique, such as UUIDs [14], only identities that do not clash with any other identity within your domain of interest. However, there are architectural advantages to using globally unique identifiers, which we explore in Chapter 34.

Proxies

For a client to be able to contact an Ice object, the client must hold a *proxy* for the Ice object. A proxy is an artifact that is local to the client's address space; it represents the (possibly remote) Ice object for the client. A proxy acts as the local ambassador for an Ice object: when the client invokes an operation on the proxy, the Ice run time:

1. Locates the Ice object
2. Activates the Ice object's server if it is not running
3. Activates the Ice object within the server
4. Transmits any in-parameters to the Ice object
5. Waits for the operation to complete

6. Returns any out-parameters and the return value to the client (or throws an exception in case of an error)

A proxy encapsulates all the necessary information for this sequence of steps to take place. In particular, a proxy contains:

- Addressing information that allows the client-side run time to contact the correct server
- An object identity that identifies which particular object in the server is the target of a request
- An optional facet identifier that determines which particular facet of an object the proxy refers to

Section 32.11 provides more information about proxies.

Stringified Proxies

The information in a proxy can be expressed as a string. For example, the string

```
SimplePrinter:default -p 10000
```

is a human-readable representation of a proxy. The Ice run time provides API calls that allow you to convert a proxy to its stringified form and vice versa. This is useful, for example, to store proxies in database tables or text files.

Provided that a client knows the identity of an Ice object and its addressing information, it can create a proxy "out of thin air" by supplying that information. In other words, no part of the information inside a proxy is considered opaque; a client needs to know only an object's identity, addressing information, and (to be able to invoke an operation) the object's type in order to contact the object.

Direct Proxies

A *direct proxy* is a proxy that embeds an object's identity, together with the *address* at which its server runs. The address is completely specified by:

- a protocol identifier (such TCP/IP or UDP)
- a protocol-specific address (such as a host name and port number)

To contact the object denoted by a direct proxy, the Ice run time uses the addressing information in the proxy to contact the server; the identity of the object is sent to the server with each request made by the client.

Indirect Proxies

An *indirect proxy* has two forms. It may provide only an object's identity, or it may specify an identity together with an *object adapter identifier*. An object that is accessible using only its identity is called a *well-known object*. For example, the string

```
SimplePrinter
```

is a valid proxy for a well-known object with the identity `SimplePrinter`.

An indirect proxy that includes an object adapter identifier has the stringified form

```
SimplePrinter@PrinterAdapter
```

Any object of the object adapter can be accessed using such a proxy, regardless of whether that object is also a well-known object.

Notice that an indirect proxy contains no addressing information. To determine the correct server, the client-side run time passes the proxy information to a location service (see Section 32.17). In turn, the location service uses the object identity or the object adapter identifier as the key in a lookup table that contains the address of the server and returns the current server address to the client. The client-side run time now knows how to contact the server and dispatches the client request as usual.

The entire process is similar to the mapping from Internet domain names to IP address by the Domain Name Service (DNS): when we use a domain name, such as www.zeroc.com, to look up a web page, the host name is first resolved to an IP address behind the scenes and, once the correct IP address is known, the IP address is used to connect to the server. With Ice, the mapping is from an object identity or object adapter identifier to a protocol–address pair, but otherwise very similar. The client-side run time knows how to contact the location service via configuration (just as web browsers know which DNS to use via configuration).

Direct Versus Indirect Binding

The process of resolving the information in a proxy to protocol–address pair is known as *binding*. Not surprisingly, *direct binding* is used for direct proxies, and *indirect binding* is used for indirect proxies.

The main advantage of indirect binding is that it allows us to move servers around (that is, change their address) without invalidating existing proxies that are held by clients. In other words, direct proxies avoid the extra lookup to locate the server but no longer work if a server is moved to a different machine. On the other hand, indirect proxies continue to work even if we move (or *migrate*) a server.

Fixed Proxies

A *fixed proxy* is a proxy that is bound to a particular connection: instead of containing addressing information or an adapter name, the proxy contains a connection handle. The connection handle stays valid only for as long as the connection stays open so, once the connection is closed, the proxy no longer works (and will never work again). Fixed proxies cannot be marshaled, that is, they cannot be passed as parameters on operation invocations. Fixed proxies are used to allow bidirectional communication, so a server can make callbacks to a client without having to open a new connection (see Section 36.7).

Routed Proxies

A *routed proxy* is a proxy that forwards all invocations to a specific target object, instead of sending invocations directly to the actual target. Routed proxies are useful to implement services such as Glacier2, which enables clients to communicate with servers that are behind a firewall (see Chapter 42).

Replication

In Ice, *replication* involves making object adapters (and their objects) available at multiple addresses. The goal of replication is usually to provide redundancy by running the same server on several computers. If one of the computers should happen to fail, a server still remains available on the others.

The use of replication implies that applications are designed for it. In particular, it means a client can access an object via one address and obtain the same result as from any other address. Either these objects are stateless, or their implementations are designed to synchronize with a database (or each other) in order to maintain a consistent view of each object's state.

Ice supports a limited form of replication when a proxy specifies multiple addresses for an object. The Ice run time selects one of the addresses at random for its initial connection attempt (see Section 32.11) and tries all of them in the case of a failure. For example, consider this proxy:

```
SimplePrinter:tcp -h server1 -p 10001:tcp -h server2 -p 10002
```

The proxy states that the object with identity `SimplePrinter` is available using TCP at two addresses, one on the host `server1` and another on the host `server2`. The burden falls to users or system administrators to ensure that the servers are actually running on these computers at the specified ports.

Replica Groups

In addition to the proxy-based replication described above, Ice supports a more useful form of replication known as *replica groups* that requires the use of a location service (see Section 32.17).

A replica group has a unique identifier and consists of any number of object adapters. An object adapter may be a member of at most one replica group; such an adapter is considered to be a *replicated object adapter*.

After a replica group has been established, its identifier can be used in an indirect proxy in place of an adapter identifier. For example, a replica group identified as `PrinterAdapters` can be used in a proxy as shown below:

```
SimplePrinter@PrinterAdapters
```

The replica group is treated by the location service as a "virtual object adapter." The behavior of the location service when resolving an indirect proxy containing a replica group id is an implementation detail. For example, the location service could decide to return the addresses of all object adapters in the group, in which case the client's Ice run time would select one of the addresses at random using the limited form of replication discussed earlier. Another possibility is for the location service to return only one address, which it decided upon using some heuristic.

Regardless of the way in which a location service resolves a replica group, the key benefit is indirection: the location service as a middleman can add more intelligence to the binding process.

Servants

As we mentioned on page 8, an Ice object is a conceptual entity that has a type, identity, and addressing information. However, client requests ultimately must end up with a concrete server-side processing entity that can provide the behavior for an operation invocation. To put this differently, a client request must ultimately end up executing code inside the server, with that code written in a specific programming language and executing on a specific processor.

The server-side artifact that provides behavior for operation invocations is known as a *servant*. A servant provides substance for (or *incarnates*) one or more Ice objects. In practice, a servant is simply an instance of a class that is written by the server developer and that is registered with the server-side run time as the servant for one or more Ice objects. Methods on the class correspond to the operations on the Ice object's interface and provide the behavior for the operations.

A single servant can incarnate a single Ice object at a time or several Ice objects simultaneously. If the former, the identity of the Ice object incarnated by the servant is implicit in the servant. If the latter, the servant is provided the identity of the Ice object with each request, so it can decide which object to incarnate for the duration of the request.

Conversely, a single Ice object can have multiple servants. For example, we might choose to create a proxy for an Ice object with two different addresses for different machines. In that case, we will have two servers, with each server containing a servant for the same Ice object. When a client invokes an operation on such an Ice object, the client-side run time sends the request to exactly one server. In other words, multiple servants for a single Ice object allow you to build redundant systems: the client-side run time attempts to send the request to one server and, if that attempt fails, sends the request to the second server. An error is reported back to the client-side application code only if that second attempt fails as well.

At-Most-Once Semantics

Ice requests have *at-most-once* semantics: the Ice run time does its best to deliver a request to the correct destination and, depending on the exact circumstances, may retry a failed request. Ice guarantees that it will either deliver the request, or, if it cannot deliver the request, inform the client with an appropriate exception; under no circumstances is a request delivered twice, that is, retries are attempted only if it is known that a previous attempt definitely failed.[1]

At-most-once semantics are important because they guarantee that operations that are not *idempotent* can be used safely. An idempotent operation is an operation that, if executed twice, has the same effect as if executed once. For example, `x = 1;` is an idempotent operation: if we execute the operation twice, the end result is the same as if we had executed it once. On the other hand, `x++;` is not idempotent: if we execute the operation twice, the end result is not the same as if we had executed it once.

Without at-most-once semantics, we can build distributed systems that are more robust in the presence of network failures. However, realistic systems require non-idempotent operations, so at-most-once semantics are a necessity, even though they make the system less robust in the presence of network failures.

1. One exception to this rule are datagram invocations over UDP transports. For these, duplicated UDP packets can lead to a violation of at-most-once semantics.

Ice permits you to mark individual operations as idempotent. For such operations, the Ice run time uses a more aggressive error recovery mechanism than for non-idempotent operations.

Synchronous Method Invocation

By default, the request dispatch model used by Ice is a synchronous remote procedure call: an operation invocation behaves like a local procedure call, that is, the client thread is suspended for the duration of the call and resumes when the call completes (and all its results are available).

Asynchronous Method Invocation

Ice also supports *asynchronous method invocation* (*AMI*): clients can invoke operations *asynchronously*, that is, the client uses a proxy as usual to invoke an operation but, in addition to passing the normal parameters, also passes a *callback object* and the client invocation returns immediately. Once the operation completes, the client-side run time invokes a method on the callback object passed initially, passing the results of the operation to the callback object (or, in case of failure, passing exception information).

The server cannot distinguish an asynchronous invocation from a synchronous one—either way, the server simply sees that a client has invoked an operation on an object.

Asynchronous Method Dispatch

Asynchronous method dispatch (*AMD*) is the server-side equivalent of AMI. For synchronous dispatch (the default), the server-side run time up-calls into the application code in the server in response to an operation invocation. While the operation is executing (or sleeping, for example, because it is waiting for data), a thread of execution is tied up in the server; that thread is released only when the operation completes.

With asynchronous method dispatch, the server-side application code is informed of the arrival of an operation invocation. However, instead of being forced to process the request immediately, the server-side application can choose to delay processing of the request and, in doing so, releases the execution thread for the request. The server-side application code is now free to do whatever it likes. Eventually, once the results of the operation are available, the server-side application code makes an API call to inform the server-side Ice run time that a request that was dispatched previously is now complete; at that point, the results of the operation are returned to the client.

Asynchronous method dispatch is useful if, for example, a server offers operations that block clients for an extended period of time. For example, the server may have an object with a `get` operation that returns data from an external, asynchronous data source and that blocks clients until the data becomes available. With synchronous dispatch, each client waiting for data to arrive ties up an execution thread in the server. Clearly, this approach does not scale beyond a few dozen clients. With asynchronous dispatch, hundreds or thousands of clients can be blocked in the same operation invocation without tying up any threads in the server.

Another way to use asynchronous method dispatch is to complete an operation, so the results of the operation are returned to the client, but to keep the execution thread of the operation beyond the duration of the operation invocation. This allows you to continue processing after results have been returned to the client, for example, to perform cleanup or write updates to persistent storage.

Synchronous and asynchronous method dispatch are transparent to the client, that is, the client cannot tell whether a server chose to process a request synchronously or asynchronously.

Oneway Method Invocation

Clients can invoke an operation as a *oneway* operation. A oneway invocation has "best effort" semantics. For a oneway invocation, the client-side run time hands the invocation to the local transport, and the invocation completes on the client side as soon as the local transport has buffered the invocation. The actual invocation is then sent asynchronously by the operating system. The server does not reply to oneway invocations, that is, traffic flows only from client to server, but not vice versa.

Oneway invocations are unreliable. For example, the target object may not exist, in which case the invocation is simply lost. Similarly, the operation may be dispatched to a servant in the server, but the operation may fail (for example, because parameter values are invalid); if so, the client receives no notification that something has gone wrong.

Oneway invocations are possible only on operations that do not have a return value, do not have out-parameters, and do not throw user exceptions (see Chapter 4).

To the application code on the server-side, oneway invocations are transparent, that is, there is no way to distinguish a twoway invocation from a oneway invocation.

Oneway invocations are available only if the target object offers a stream-oriented transport, such as TCP/IP or SSL.

Note that, even though oneway operations are sent over a stream-oriented transport, they may be processed out of order in the server. This can happen because each invocation may be dispatched in its own thread: even though the invocations are *initiated* in the order in which the invocations arrive at the server, this does not mean that they will be *processed* in that order—the vagaries of thread scheduling can result in a oneway invocation to complete before other oneway invocations that were received earlier.

Batched Oneway Method Invocation

Each oneway invocation sends a separate message to the server. For a series of short messages, the overhead of doing so is considerable: the client- and server-side run time each must switch between user mode and kernel mode for each message and, at the networking level, each message incurs the overheads of flow-control and acknowledgement.

Batched oneway invocations allow you to send a series of oneway invocations as a single message: every time you invoke a batched oneway operation, the invocation is buffered in the client-side run time. Once you have accumulated all the oneway invocations you want to send, you make a separate API call to send all the invocations at once. The client-side run time then sends all of the buffered invocations in a single message, and the server receives all of the invocations in a single message. This avoids the overhead of repeatedly trapping into the kernel for both client and server, and is much easier on the network between them because one large message can be transmitted more efficiently than many small ones.

The individual invocations in a batched oneway message are dispatched by a single thread in the order in which they were placed into the batch. This guarantees that the individual operations in a batched oneway message are processed in order in the server.

Batched oneway invocations are particularly useful for messaging services, such as IceStorm (see Chapter 44), and for fine-grained interfaces that offer `set` operations for small attributes.

Datagram Invocations

Datagram invocations have similar “best effort” semantics to oneway invocations. However, datagram invocations require the object to offer UDP as a transport (whereas oneway invocations require TCP/IP).

Like a oneway invocation, a datagram invocation can be made only if the operation does not have a return value, out-parameters, or user exceptions. A datagram invocation uses UDP to invoke the operation. The operation returns as soon as the local UDP stack has accepted the message; the actual operation invocation is sent asynchronously by the network stack behind the scenes.

Datagrams, like oneway invocations, are unreliable: the target object may not exist in the server, the server may not be running, or the operation may be invoked in the server but fail due to invalid parameters sent by the client. As for oneway invocations, the client receives no notification of such errors.

However, unlike oneway invocations, datagram invocations have a number of additional error scenarios:

- Individual invocations may simply be lost in the network.

  This is due to the unreliable delivery of UDP packets. For example, if you invoke three operations in sequence, the middle invocation may be lost. (The same thing cannot happen for oneway invocations—because they are delivered over a connection-oriented transport, individual invocations cannot be lost.)

- Individual invocations may arrive out of order.

  Again, this is due to the nature of UDP datagrams. Because each invocation is sent as a separate datagram, and individual datagrams can take different paths through the network, it can happen that invocations arrive in an order that differs from the order in which they were sent.

Datagram invocations are well suited for small messages on LANs, where the likelihood of loss is small. They are also suited to situations in which low latency is more important than reliability, such as for fast, interactive internet applications. Finally, datagram invocations can be used to multicast messages to multiple servers simultaneously.

Batched Datagram Invocations

As for batched oneway invocations, *batched datagram invocations* allow you to accumulate a number of invocations in a buffer and then send the entire buffer as a single datagram by making an API call to flush the buffer. Batched datagrams reduce the overhead of repeated system calls and allow the underlying network to operate more efficiently. However, batched datagram invocations are useful only for batched messages whose total size does not substantially exceed the PDU limit of the network: if the size of a batched datagram gets too large, UDP fragmentation makes it more likely that one or more fragments are lost, which results in the

loss of the entire batched message. However, you are guaranteed that either all invocations in a batch will be delivered, or none will be delivered. It is impossible for individual invocations within a batch to be lost.

Batched datagrams use a single thread in the server to dispatch the individual invocations in a batch. This guarantees that the invocations are made in the order in which they were queued—invocations cannot appear to be reordered in the server.

Run-Time Exceptions

Any operation invocation can raise a *run-time exception*. Run-time exceptions are pre-defined by the Ice run time and cover common error conditions, such as connection failure, connection timeout, or resource allocation failure. Run-time exceptions are presented to the application as native exceptions and so integrate neatly with the native exception handling capabilities of languages that support exception handling.

User Exceptions

User exceptions are used to indicate application-specific error conditions to clients. User exceptions can carry an arbitrary amount of complex data and can be arranged into inheritance hierarchies, which makes it easy for clients to handle categories of errors generically, by catching an exception that is further up the inheritance hierarchy. Like run-time exceptions, user exceptions map to native exceptions.

Properties

Much of the Ice run time is configurable via *properties*. Properties are name–value pairs, such as `Ice.Default.Protocol=tcp`. Properties are typically stored in text files and parsed by the Ice run time to configure various options, such as the thread pool size, the level of tracing, and various other configuration parameters.

2.2.3 Slice (Specification Language for Ice)

As mentioned on page 9, each Ice object has an interface with a number of operations. Interfaces, operations, and the types of data that are exchanged between client and server are defined using the *Slice language*. Slice allows you to define the client-server contract in a way that is independent of a specific programming language, such as C++, Java, or C#. The Slice definitions are compiled by a compiler into an API for a specific programming language, that is, the part of the

API that is specific to the interfaces and types you have defined consists of generated code.

2.2.4 Language Mappings

The rules that govern how each Slice construct is translated into a specific programming language are known as *language mappings*. For example, for the C++ mapping (see Chapter 6), a Slice sequence appears as an STL vector, whereas, for the Java mapping (see Chapter 10), a Slice sequence appears as a Java array. In order to determine what the API for a specific Slice construct looks like, you only need the Slice definition and knowledge of the language mapping rules. The rules are simple and regular enough to make it unnecessary to read the generated code to work out how to use the generated API.

Of course, you are free to peruse the generated code. However, as a rule, that is inefficient because the generated code is not necessarily suitable for human consumption. We recommend that you familiarize yourself with the language mapping rules; that way, you can mostly ignore the generated code and need to refer to it only when you are interested in some specific detail.

Currently, Ice provides language mappings for C++, Java, C#, Python, Objective-C, and, for the client side, PHP and Ruby.

2.2.5 Client and Server Structure

Ice clients and servers have the logical internal structure shown in Figure 2.1

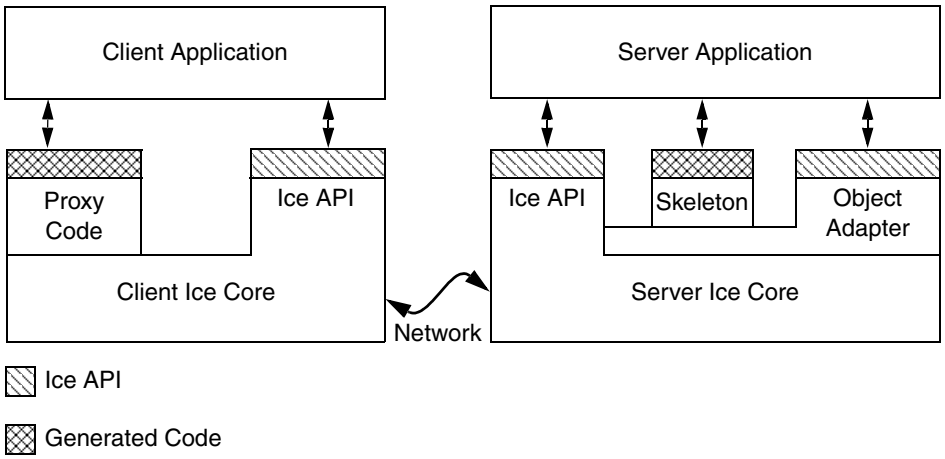


Figure 2.1. Ice Client and Server Structure

Both client and server consist of a mixture of application code, library code, and code generated from Slice definitions:

- The Ice core contains the client- and server-side run-time support for remote communication. Much of this code is concerned with the details of networking, threading, byte ordering, and many other networking-related issues that we want to keep away from application code. The Ice core is provided as a number of libraries that client and server use.
- The generic part of the Ice core (that is, the part that is independent of the specific types you have defined in Slice) is accessed through the Ice API. You use the Ice API to take care of administrative chores, such as initializing and finalizing the Ice run time. The Ice API is identical for clients and servers (although servers use a larger part of the API than clients).
- The proxy code is generated from your Slice definitions and, therefore, specific to the types of objects and data you have defined in Slice. The proxy code has two major functions:
  - It provides a down-call interface for the client. Calling a function in the generated proxy API ultimately ends up sending an RPC message to the server that invokes a corresponding function on the target object.

- It provides *marshaling* and *unmarshaling* code.

  Marshaling is the process of serializing a complex data structure, such as a sequence or a dictionary, for transmission on the wire. The marshaling code converts data into a form that is standardized for transmission and independent of the endian-ness and padding rules of the local machine.

  Unmarshaling is the reverse of marshaling, that is, deserializing data that arrives over the network and reconstructing a local representation of the data in types that are appropriate for the programming language in use.

- The skeleton code is also generated from your Slice definition and, therefore, specific to the types of objects and data you have defined in Slice. The skeleton code is the server-side equivalent of the client-side proxy code: it provides an up-call interface that permits the Ice run time to transfer the thread of control to the application code you write. The skeleton also contains marshaling and unmarshaling code, so the server can receive parameters sent by the client, and return parameters and exceptions to the client.

- The object adapter is a part of the Ice API that is specific to the server side: only servers use object adapters. An object adapter has several functions:

  - The object adapter maps incoming requests from clients to specific methods on programming-language objects. In other words, the object adapter tracks which servants with what object identity are in memory.

  - The object adapter is associated with one or more transport endpoints. If more than one transport endpoint is associated with an adapter, the servants incarnating objects within the adapter can be reached via multiple transports. For example, you can associate both a TCP/IP and a UDP endpoint with an adapter, to provide alternate quality-of-service and performance characteristics.

  - The object adapter is responsible for the creation of proxies that can be passed to clients. The object adapter knows about the type, identity, and transport details of each of its objects and embeds the correct details when the server-side application code requests the creation of a proxy.

Note that, as far as the process view is concerned, there are only two processes involved: the client and the server. All the run time support for distributed communication is provided by the Ice libraries and the code that is generated from Slice definitions. (For indirect proxies, a third process, IceGrid, is required to resolve proxies to transport endpoints.)

2.2.6 The Ice Protocol

Ice provides an RPC protocol that can use either TCP/IP or UDP as an underlying transport. In addition, Ice also allows you to use SSL as a transport, so all communication between client and server is encrypted.

The Ice protocol defines:

- a number of message types, such as request and reply message types,
- a protocol state machine that determines in what sequence different message types are exchanged by client and server, together with the associated connection establishment and tear-down semantics for TCP/IP,
- encoding rules that determine how each type of data is represented on the wire,
- a header for each message type that contains details such as the message type, the message size, and the protocol and encoding version in use.

Ice also supports compression on the wire: by setting a configuration parameter, you can arrange for all network traffic to be compressed to conserve bandwidth. This is useful if your application exchanges large amounts of data between client and server.

The Ice protocol is suitable for building highly-efficient event forwarding mechanisms because it permits forwarding of a message without knowledge of the details of the information inside a message. This means that messaging switches need not do any unmarshaling and remarshaling of messages—they can forward a message by simply treating it as an opaque buffer of bytes.

The Ice protocol also supports bidirectional operation: if a server wants to send a message to a callback object provided by the client, the callback can be made over the connection that was originally created by the client. This feature is especially important when the client is behind a firewall that permits outgoing connections, but not incoming connections.

2.3 Ice Services

The Ice core provides a sophisticated client–server platform for distributed application development. However, realistic applications usually require more than just a remoting capability: typically, you also need a way to start servers on demand, distribute proxies to clients, distribute asynchronous events, configure your application, distribute patches for an application, and so on.

Ice ships with a number of services that provide these and other features. The services are implemented as Ice servers to which your application acts as a client. None of the services use Ice-internal features that are hidden from application developers so, in theory, you could develop equivalent services yourself. However, having these services available as part of the platform allows you to focus on application development instead of having to build a lot of infrastructure first. Moreover, building such services is not a trivial effort, so it pays to know what is available and use it instead of reinventing your own wheel.

2.3.1 Freeze and FreezeScript

Ice has a built-in object persistence service, known as *Freeze*. Freeze makes it easy to store object state in a database: you define the state stored by your objects in Slice, and the Freeze compiler generates code that stores and retrieves object state to and from a database. Freeze uses Berkeley DB [18] as its database. We discuss Freeze in detail in Chapter 39.

Ice also offers a tool called FreezeScript that makes it easier to maintain databases and to migrate the contents of existing databases to a new schema if the type definitions of objects change. We discuss FreezeScript in Chapter 40.

2.3.2 IceGrid

IceGrid is an implementation of an Ice location service that resolves the symbolic information in an indirect proxy to a protocol–address pair for indirect binding. A location service is only the beginning of IceGrid's capabilities:

- IceGrid allows you to register servers for automatic start-up: instead of requiring a server to be running at the time a client issues a request, IceGrid starts servers on demand, when the first client request arrives.
- IceGrid provides tools that make it easy to configure complex applications containing several servers.
- IceGrid supports replication and load-balancing.
- IceGrid automates the distribution and patching of server executables and dependent files.
- IceGrid provides a simple query service that allows clients to obtain proxies for objects they are interested in.

2.3.3 IceBox

IceBox is a simple application server that can orchestrate the starting and stopping of a number of application components. Application components can be deployed as a dynamic library instead of as a process. This reduces overall system load, for example, by allowing you to run several application components in a single Java virtual machine instead of having multiple processes, each with its own virtual machine.

2.3.4 IceStorm

IceStorm is a publish–subscribe service that decouples clients and servers. Fundamentally, IceStorm acts as a distribution switch for events. Publishers send events to the service, which, in turn, passes the events to subscribers. In this way, a single event published by a publisher can be sent to multiple subscribers. Events are categorized by topic, and subscribers specify the topics they are interested in. Only events that match a subscriber's topic are sent to that subscriber. The service permits selection of a number of quality-of-service criteria to allow applications to choose the appropriate trade-off between reliability and performance.

IceStorm is particularly useful if you have a need to distribute information to large numbers of application components. (A typical example is a stock ticker application with a large number of subscribers.) IceStorm decouples the publishers of information from subscribers and takes care of the redistribution of the published events. In addition, IceStorm can be run as a *federated* service, that is, multiple instances of the service can be run on different machines to spread the processing load over a number of CPUs.

2.3.5 IcePatch2

IcePatch2[2] is a software patching service. It allows you to easily distribute software updates to clients. Clients simply connect to the IcePatch2 server and request updates for a particular application. The service automatically checks the version of the client's software and downloads any updated application components in a compressed format to conserve bandwidth. Software patches can be secured using the Glacier2 service, so only authorized clients can download software updates.

2. IcePatch2 supersedes IcePatch, which was a previous version of this service.

2.3.6 Glacier2

Glacier2[3] is the Ice firewall traversal service: it allows clients and servers to securely communicate through a firewall without compromising security. Client-server traffic is SSL-encrypted using public key certificates and is bidirectional. Glacier2 offers support for mutual authentication as well as secure session management.

2.4 Architectural Benefits of Ice

The Ice architecture provides a number of benefits to application developers:

- Object-oriented semantics

  Ice fully preserves the object-oriented paradigm "across the wire." All operation invocations use late binding, so the implementation of an operation is chosen depending on the actual run-time (not static) type of an object.

- Support for synchronous and asynchronous messaging

  Ice provides both synchronous and asynchronous operation invocation and dispatch, as well as publish–subscribe messaging via IceStorm. This allows you to choose a communication model according to the needs of your application instead of having to shoe-horn the application to fit a single model.

- Support for multiple interfaces

  With facets, objects can provide multiple, unrelated interfaces while retaining a single object identity across these interfaces. This provides great flexibility, particularly as an application evolves but needs to remain compatible with older, already deployed clients.

- Machine independence

  Clients and servers are shielded form idiosyncrasies of the underlying machine architecture. Issues such as byte ordering and padding are hidden from application code.

- Language independence

  Client and server can be developed independently and in different programming languages. The Slice definition used by both client and server estab-

3. Glacier2 supersedes Glacier, which was a previous version of this service.

lishes the interface contract between them and is the only thing they need to agree on.

- Implementation independence

  Clients are unaware of how servers implement their objects. This means that the implementation of a server can be changed after clients are deployed, for example, to use a different persistence mechanism or even a different programming language.

- Operating system independence

  The Ice APIs are fully portable, so the same source code compiles and runs under both Windows and Unix.

- Threading support

  The Ice run time is fully threaded and APIs are thread-safe. No effort (beyond synchronizing access to shared data) is required on part of the application developer to develop threaded, high-performance clients and servers.

- Transport independence

  Ice currently offers both TCP/IP and UDP as transport protocols. Neither client nor server code are aware of the underlying transport. (The desired transport can be chosen by a configuration parameter.)

- Location and server transparency

  The Ice run time takes care of locating objects and managing the underlying transport mechanism, such as opening and closing connections. Interactions between client and server appear connection-less. Via IceGrid, you can arrange for servers to be started on demand if they are not running at the time a client invokes an operation. Servers can be migrated to different physical addresses without breaking proxies held by clients, and clients are completely unaware how object implementations are distributed over server processes.

- Security

  Communications between client and server can be fully secured with strong encryption over SSL, so applications can use unsecured public networks to communicate securely. Via Glacier2, you can implement secure forwarding of requests through a firewall, with full support for callbacks.

- Built-in persistence

  With Freeze, creating persistent object implementations becomes trivial. Ice comes with built-in support for Berkeley DB [18], which is a high-performance database.

- Source code availability

  The source code for Ice is available. While it is not necessary to have access to the source code to use the platform, it allows you to see how things are implemented or port the code to a new operating system.

Overall, Ice provides a state-of-the art development and deployment environment for distributed computing that is more complete than any other platform we are aware of.

Chapter 3
A Hello World Application

3.1 Chapter Overview

In this chapter, we will see how to create a very simple client–server application in C++ (Section 3.3), Java (Section 3.4), C# (Section 3.5), Visual Basic (Section 3.6), Objective-C (Section 3.7), Python (Section 3.8), Ruby (Section 3.9), and PHP (Section 3.10). Rather than reading the entire chapter, we suggest that you read Section 3.2 and then skip to the section that deals with the programming language of your choice.

The application enables remote printing: a client sends the text to be printed to a server, which in turn sends that text to a printer. For simplicity (and because we do not want to concern ourselves with the idiosyncrasies of print spoolers for various platforms), our printer will simply print to a terminal instead of a real printer. This is no great loss: the purpose of the exercise is to show how a client can communicate with a server; once the thread of control has reached the server application code, that code can of course do anything it likes (including sending the text to a real printer). How to do this is independent of Ice and therefore not relevant here.

Note that much of the detail of the source code will remain unexplained for now. The intent is to show you how to get started and give you a feel for what the development environment looks like; we will provide all the detail throughout the remainder of this book.

3.2 Writing a Slice Definition

The first step in writing any Ice application is to write a Slice definition containing the interfaces that are used by the application. For our minimal printing application, we write the following Slice definition:

```
module Demo {
    interface Printer {
        void printString(string s);
    };
};
```

We save this text in a file called `Printer.ice`.

Our Slice definitions consist of the module `Demo` containing a single interface called `Printer`. For now, the interface is very simple and provides only a single operation, called `printString`. The `printString` operation accepts a string as its sole input parameter; the text of that string is what appears on the (possibly remote) printer.

3.3 Writing an Ice Application with C++

This section shows how to create an Ice application with C++.

Compiling a Slice Definition for C++

The first step in creating our C++ application is to compile our Slice definition to generate C++ proxies and skeletons. Under Unix, you can compile the definition as follows:

```
$ slice2cpp Printer.ice
```

The **`slice2cpp`** compiler produces two C++ source files from this definition, `Printer.h` and `Printer.cpp`.

- `Printer.h`

  The `Printer.h` header file contains C++ type definitions that correspond to the Slice definitions for our `Printer` interface. This header file must be included in both the client and the server source code.

- `Printer.cpp`

  The `Printer.cpp` file contains the source code for our `Printer` interface. The generated source contains type-specific run-time support for both clients

and servers. For example, it contains code that marshals parameter data (the string passed to the `printString` operation) on the client side and unmarshals that data on the server side.

The `Printer.cpp` file must be compiled and linked into both client and server.

Writing and Compiling a Server

The source code for the server takes only a few lines and is shown in full here:

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

class PrinterI : public Printer {
public:
    virtual void printString(const string& s,
                             const Ice::Current&);
};

void
PrinterI::
printString(const string& s, const Ice::Current&)
{
    cout << s << endl;
}

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter
            = ic->createObjectAdapterWithEndpoints(
                "SimplePrinterAdapter", "default -p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object,
                     ic->stringToIdentity("SimplePrinter"));
        adapter->activate();
        ic->waitForShutdown();
    } catch (const Ice::Exception& e) {
```

```
        cerr << e << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const Ice::Exception& e) {
            cerr << e << endl;
            status = 1;
        }
    }
    return status;
}
```

There appears to be a lot of code here for something as simple as a server that just prints a string. Do not be concerned by this: most of the preceding code is boiler plate that never changes. For this very simple server, the code is dominated by this boiler plate.

Every Ice source file starts with an include directive for `Ice.h`, which contains the definitions for the Ice run time. We also include `Printer.h`, which was generated by the Slice compiler and contains the C++ definitions for our printer interface, and we import the contents of the `std` and `Demo` namespaces for brevity in the code that follows:

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;
```

Our server implements a single printer servant, of type `PrinterI`. Looking at the generated code in `Printer.h`, we find the following (tidied up a little to get rid of irrelevant detail):

```
namespace Demo {
    class Printer : virtual public Ice::Object {
    public:
        virtual void printString(const std::string&,
                                 const Ice::Current&
```

```
                                    = Ice::Current()
                                  ) = 0;
    };
};
```

The `Printer` skeleton class definition is generated by the Slice compiler. (Note that the `printString` method is pure virtual so the skeleton class cannot be instantiated.) Our servant class inherits from the skeleton class to provide an implementation of the pure virtual `printString` method. (By convention, we use an `I`-suffix to indicate that the class implements an interface.)

```
class PrinterI : public Printer {
public:
    virtual void printString(const string& s,
                             const Ice::Current&);
};
```

The implementation of the `printString` method is trivial: it simply writes its string argument to `stdout`:

```
void
PrinterI::
printString(const string& s, const Ice::Current&)
{
    cout << s << endl;
}
```

Note that `printString` has a second parameter of type `Ice::Current`. As you can see from the definition of `Printer::printString`, the Slice compiler generates a default argument for this parameter, so we can leave it unused in our implementation. (We will examine the purpose of the `Ice::Current` parameter in Section 32.6.)

What follows is the server main program. Note the general structure of the code:

```
int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {

        // Server implementation here...

    } catch (const Ice::Exception& e) {
        cerr << e << endl;
```

```
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const Ice::Exception& e) {
            cerr << e << endl;
            status = 1;
        }
    }
    return status;
}
```

The body of `main` contains the declaration of two variables, `status` and `ic`. The `status` variable contains the exit status of the program and the `ic` variable, of type `Ice::CommunicatorPtr`, contains the main handle to the Ice run time.

Following these declarations is a `try` block in which we place all the server code, followed by two `catch` handlers. The first handler catches all exceptions that may be thrown by the Ice run time; the intent is that, if the code encounters an unexpected Ice run-time exception anywhere, the stack is unwound all the way back to `main`, which prints the exception and then returns failure to the operating system. The second handler catches string constants; the intent is that, if we encounter a fatal error condition somewhere in our code, we can simply throw a string literal with an error message. Again, this unwinds the stack all the way back to `main`, which prints the error message and then returns failure to the operating system.

Following the `try` block, we see a bit of cleanup code that calls the `destroy` method on the communicator (provided that the communicator was initialized). The cleanup call is outside the first `try` block for a reason: we must ensure that the Ice run time is finalized whether the code terminates normally or terminates due to an exception.[1]

The body of the first `try` block contains the actual server code:

1. Failure to call `destroy` on the communicator before the program exits results in undefined behavior.

```
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter
            = ic->createObjectAdapterWithEndpoints(
                "SimplePrinterAdapter", "default -p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object, ic->stringToIdentity("SimplePrinter")
);
        adapter->activate();
        ic->waitForShutdown();
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice::initialize`. (We pass `argc` and `argv` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns a smart pointer to an `Ice::Communicator` object, which is the main handle to the Ice run time.

2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.

3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.

4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the servant. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)

5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.) The server starts to process incoming requests from clients as soon as the adapter is activated.

6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice ships with such a helper class, called `Ice::Application`—see Section 8.3.1.) As far as actual application code is concerned, the server contains only a few lines: six lines for the definition of the `PrinterI` class, plus three[2] lines to instantiate a `PrinterI` object and register it with the object adapter.

Assuming that we have the server code in a file called `Server.cpp`, we can compile it as follows:

```
$ c++ -I. -I$ICE_HOME/include -c Printer.cpp Server.cpp
```

This compiles both our application code and the code that was generated by the Slice compiler. We assume that the **`ICE_HOME`** environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in `/opt/Ice`, set **`ICE_HOME`** to that path.) Depending on your platform, you may have to add additional include directives or other options to the compiler (such as an include directive for the STLport headers, or to control template instantiation); please see the demo programs that ship with Ice for the details.

Finally, we need to link the server into an executable:

```
$ c++ -o server Printer.o Server.o \
-L$ICE_HOME/lib -lIce -lIceUtil
```

Again, depending on the platform, the actual list of libraries you need to link against may be longer. The demo programs that ship with Ice contain all the detail. The important point to note here is that the Ice run time is shipped in two libraries, `libIce` and `libIceUtil`.

Writing and Compiling a Client

The client code looks very similar to the server. Here it is in full:

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

int
```

2. Well, two lines, really: printing space limitations force us to break source lines more often than you would in your actual source files.

```
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectPrx base = ic->stringToProxy(
                                "SimplePrinter:default -p 10000");
        PrinterPrx printer = PrinterPrx::checkedCast(base);
        if (!printer)
            throw "Invalid proxy";

        printer->printString("Hello World!");
    } catch (const Ice::Exception& ex) {
        cerr << ex << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}
```

Note that the overall code layout is the same as for the server: we include the headers for the Ice run time and the header generated by the Slice compiler, and we use the same `try` block and `catch` handlers to deal with errors.

The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice::initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 38.)
3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Print-`

`erPrx::checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy to a `Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns a null proxy.

4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.

5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling and linking the client looks much the same as for the server:

```
$ c++ -I. -I$ICE_HOME/include -c Printer.cpp Client.cpp
$ c++ -o client Printer.o Client.o -L$ICE_HOME/lib -lIce -lIceUtil
```

Running Client and Server

To run client and server, we first start the server in a separate window:

```
$ ./server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ ./client
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in Section 8.3.1.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get:

```
Network.cpp:471: Ice::ConnectFailedException:
connect failed: Connection refused
```

Note that, to successfully run client and server, you will have to set some platform-dependent environment variables. For example, under Linux, you need to add the Ice library directory to your **`LD_LIBRARY_PATH`**. Please have a look at the demo applications that ship with Ice for the details for your platform.

3.4 Writing an Ice Application with Java

This section shows how to create an Ice application with Java.

Compiling a Slice Definition for Java

The first step in creating our Java application is to compile our Slice definition to generate Java proxies and skeletons. Under Unix, you can compile the definition as follows:[3]

```
$ mkdir generated
$ slice2java --output-dir generated Printer.ice
```

The **`--output-dir`** option instructs the compiler to place the generated files into the `generated` directory. This avoids cluttering the working directory with the generated files. The **`slice2java`** compiler produces a number of Java source files from this definition. The exact contents of these files do not concern us for now—they contain the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing and Compiling a Server

To implement our `Printer` interface, we must create a servant class. By convention, servant classes use the name of their interface with an `I`-suffix, so our servant class is called `PrinterI` and placed into a source file `PrinterI.java`:

```
public class PrinterI extends Demo._PrinterDisp {
    public void
    printString(String s, Ice.Current current)
    {
        System.out.println(s);
    }
}
```

The `PrinterI` class inherits from a base class called `_PrinterDisp`, which is generated by the **`slice2java`** compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter. We will see its purpose in detail in Section 32.6.) Our

3. Whenever we show Unix commands in this book, we assume a Bourne or Bash shell. The commands for Windows are essentially identical and therefore not shown.

implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code is in a source file called `Server.java`, shown in full here:

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            Ice.ObjectAdapter adapter
                = ic.createObjectAdapterWithEndpoints(
                    "SimplePrinterAdapter", "default -p 10000");
            Ice.Object object = new PrinterI();
            adapter.add(
                    object,
                    ic.stringToIdentity("SimplePrinter"));
            adapter.activate();
            ic.waitForShutdown();
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                status = 1;
            }
        }
        System.exit(status);
    }
}
```

Note the general structure of the code:

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            Ice.ObjectAdapter adapter
                = ic.createObjectAdapterWithEndpoints(
                    "SimplePrinterAdapter", "default -p 10000");
            Ice.Object object = new PrinterI();
            adapter.add(
                    object,
                    ic.stringToIdentity("SimplePrinter"));
            adapter.activate();
            ic.waitForShutdown();
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                status = 1;
            }
        }
        System.exit(status);
    }
}
```

The body of `main` contains a `try` block in which we place all the server code, followed by two `catch` blocks. The first block catches all exceptions that may be thrown by the Ice run time; the intent is that, if the code encounters an unexpected Ice run-time exception anywhere, the stack is unwound all the way back to `main`, which prints the exception and then returns failure to the operating system. The second block catches `Exception` exceptions; the intent is that, if we encounter a fatal error condition somewhere in our code, we can simply throw an exception

with an error message. Again, this unwinds the stack all the way back to `main`, which prints the error message and then returns failure to the operating system.

Before the code exits, it destroys the communicator (if one was created successfully). Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results.

The body of our `try` block contains the actual server code:

```
ic = Ice.Util.initialize(args);
Ice.ObjectAdapter adapter
    = ic.createObjectAdapterWithEndpoints(
        "SimplePrinterAdapter", "default -p 10000");
Ice.Object object = new PrinterI();
adapter.add(
        object,
        ic.stringToIdentity("SimplePrinter"));
adapter.activate();
ic.waitForShutdown();
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.Util.initialize`. (We pass `args` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns an `Ice::Communicator` reference, which is the main handle to the Ice run time.

2. We create an object adapter by calling `createObjectAdapterWith-Endpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.

3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.

4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the servant. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)

5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants

that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)

6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice ships with such a helper class, called `Ice.Application`—see Section 12.3.1.) As far as actual application code is concerned, the server contains only a few lines: seven lines for the definition of the `PrinterI` class, plus four[4] lines to instantiate a `PrinterI` object and register it with the object adapter.

We can compile the server code as follows:

```
$ mkdir classes
$ javac -d classes -classpath classes:$ICEJ_HOME/lib/Ice.jar\
-source 1.4 Server.java PrinterI.java generated/Demo/*.java
```

This compiles both our application code and the code that was generated by the Slice compiler. We assume that the **`ICEJ_HOME`** environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in `/opt/Icej`, set **`ICEJ_HOME`** to that path.) Note that Ice for Java uses the **`ant`** build environment to control building of source code. (**`ant`** is similar to **`make`**, but more flexible for Java applications.) You can have a look at the demo code that ships with Ice to see how to use this tool.

Writing and Compiling a Client

The client code, in `Client.java`, looks very similar to the server. Here it is in full:

```
public class Client {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
```

4. Well, two lines, really: printing space limitations force us to break source lines more often than you would in your actual source files.

```
        try {
            ic = Ice.Util.initialize(args);
            Ice.ObjectPrx base = ic.stringToProxy(
                    "SimplePrinter:default -p 10000");
            Demo.PrinterPrx printer
                = Demo.PrinterPrxHelper.checkedCast(base);
            if (printer == null)
                throw new Error("Invalid proxy");

            printer.printString("Hello World!");
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                status = 1;
            }
        }
        System.exit(status);
    }
}
```

Note that the overall code layout is the same as for the server: we use the same `try` and `catch` blocks to deal with errors. The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.Util.initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 38.)

3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrxHelper.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy of type `Demo::Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns null.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling the client looks much the same as for the server:

```
$ javac -d classes -classpath classes:$ICEJ_HOME/lib/Ice.jar\
-source 1.4 Client.java PrinterI.java generated/Demo/*.java
```

Running Client and Server

To run client and server, we first start the server in a separate window:

```
$ java Server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ java Client
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in Chapter 12.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Ice.ConnectFailedException
        at IceInternal.Network.doConnect(Network.java:201)
        at IceInternal.TcpConnector.connect(TcpConnector.java:26)
        at
IceInternal.OutgoingConnectionFactory.create(OutgoingConnectionFac
tory.java:80)
```

```
        at Ice._ObjectDelM.setup(_ObjectDelM.java:251)
        at Ice.ObjectPrxHelper.__getDelegate(ObjectPrxHelper.java:
642)
        at Ice.ObjectPrxHelper.ice_isA(ObjectPrxHelper.java:41)
        at Ice.ObjectPrxHelper.ice_isA(ObjectPrxHelper.java:30)
        at Demo.PrinterPrxHelper.checkedCast(Unknown Source)
        at Client.main(Unknown Source)
Caused by: java.net.ConnectException: Connection refused
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method
)
        at
sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:
518)
        at IceInternal.Network.doConnect(Network.java:173)
        ... 8 more
```

Note that, to successfully run client and server, your **CLASSPATH** must include the Ice library and the classes directory, for example:

```
$ export CLASSPATH=$CLASSPATH:./classes:$ICEJ_HOME/lib/Ice.jar
```

Please have a look at the demo applications that ship with Ice for the details for your platform.

3.5 Writing an Ice Application with C#

This section shows how to create an Ice application with C#.

Compiling a Slice Definition for C#

The first step in creating our C# application is to compile our Slice definition to generate C# proxies and skeletons. You can compile the definition as follows:

```
$ mkdir generated
$ slice2cs --output-dir generated Printer.ice
```

The **--output-dir** option instructs the compiler to place the generated files into the `generated` directory. This avoids cluttering the working directory with the generated files. The **slice2cs** compiler produces a single source file, `Printer.cs`, from this definition. The exact contents of this file do not concern us for now—it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing and Compiling a Server

To implement our `Printer` interface, we must create a servant class. By convention, servant classes use the name of their interface with an `I`-suffix, so our servant class is called `PrinterI` and placed into a source file `Server.cs`:

```
using System;

public class PrinterI : Demo.PrinterDisp_
{
   public override void printString(string s, Ice.Current current)
   {
      Console.WriteLine(s);
   }
}
```

The `PrinterI` class inherits from a base class called `_PrinterDisp`, which is generated by the **`slice2cs`** compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter. We will see its purpose in detail in Section 32.6.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code follows in `Server.cs` and is shown in full here:

```
public class Server
{
    public static void Main(string[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(ref args);
            Ice.ObjectAdapter adapter
                = ic.createObjectAdapterWithEndpoints(
                    "SimplePrinterAdapter", "default -p 10000");
            Ice.Object obj = new PrinterI();
            adapter.add(
                    obj,
                    ic.stringToIdentity("SimplePrinter"));
            adapter.activate();
            ic.waitForShutdown();
        } catch (Exception e) {
            Console.Error.WriteLine(e);
            status = 1;
        }
```

```
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                Console.Error.WriteLine(e);
                status = 1;
            }
        }
        Environment.Exit(status);
    }
}
```

Note the general structure of the code:

```
public class Server
{
    public static void Main(string[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {

            // Server implementation here...

        } catch (Exception e) {
            Console.Error.WriteLine(e);
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                Console.Error.WriteLine(e);
                status = 1;
            }
        }
        Environment.Exit(status);
    }
}
```

The body of `Main` contains a `try` block in which we place all the server code, followed by a `catch` block. The catch block catches all exceptions that may be

thrown by the code; the intent is that, if the code encounters an unexpected run-time exception anywhere, the stack is unwound all the way back to `Main`, which prints the exception and then returns failure to the operating system.

Before the code exits, it destroys the communicator (if one was created successfully). Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results.

The body of our `try` block contains the actual server code:

```
ic = Ice.Util.initialize(ref args);
Ice.ObjectAdapter adapter
    = ic.createObjectAdapterWithEndpoints(
        "SimplePrinterAdapter", "default -p 10000");
Ice.Object obj = new PrinterI();
adapter.add(
        obj,
        ic.stringToIdentity("SimplePrinter"));
adapter.activate();
ic.waitForShutdown();
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.Util.initialize`. (We pass `args` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns an `Ice::Communicator` reference, which is the main handle to the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWith-Endpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the servant. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants

that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)

6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice ships with such a helper class, called `Ice.Application`—see Section 16.3.1.) As far as actual application code is concerned, the server contains only a few lines: seven lines for the definition of the `PrinterI` class, plus four[5] lines to instantiate a `PrinterI` object and register it with the object adapter.

We can compile the server code as follows:

```
$ csc /reference:Ice.dll /lib:%ICE_HOME%\bin Server.cs \
generated\Printer.cs
```

This compiles both our application code and the code that was generated by the Slice compiler. We assume that the **`ICE_HOME`** environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in `C:\opt\Ice`, set **`ICE_HOME`** to that path.)

Writing and Compiling a Client

The client code, in `Client.cs`, looks very similar to the server. Here it is in full:

```
using System;
using Demo;

public class Client
{
    public static void Main(string[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(ref args);
            Ice.ObjectPrx obj = ic.stringToProxy(
```

5. Well, two lines, really: printing space limitations force us to break source lines more often than you would in your actual source files.

```
                "SimplePrinter:default -p 10000");
            PrinterPrx printer
                = PrinterPrxHelper.checkedCast(obj);
            if (printer == null)
                throw new ApplicationException("Invalid proxy");

            printer.printString("Hello World!");
        } catch (Exception e) {
            Console.Error.WriteLine(e);
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                Console.Error.WriteLine(e);
                status = 1;
            }
        }
        Environment.Exit(status);
    }
}
```

Note that the overall code layout is the same as for the server: we use the same `try` and `catch` blocks to deal with errors. The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.Util.initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 38.)
3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrxHelper.checkedCast`. A checked cast sends a message to the server,

effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy of type `Demo::Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns null.

4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling the client looks much the same as for the server:

```
$ csc /reference:Ice.dll /lib:%ICE_HOME%\bin Client.cs \
generated\Printer.cs
```

Running Client and Server

To run client and server, we first start the server in a separate window:

```
$ server.exe
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ client.exe
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in Chapter 16.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Ice.ConnectionRefusedException
    error = 0
   at IceInternal.ProxyFactory.checkRetryAfterException(LocalExcep
tion ex, Reference ref, Int32 cnt) in c:\cygwin\home\m
ichi\src\ice\cs\src\Ice\ProxyFactory.cs:line 167
   at Ice.ObjectPrxHelperBase.handleException__(ObjectDel_ delegat
e, LocalException ex, Int32 cnt) in c:\cygwin\home\mic
hi\src\ice\cs\src\Ice\Proxy.cs:line 970
   at Ice.ObjectPrxHelperBase.ice_isA(String id__, Dictionary`2 co
ntext__, Boolean explicitContext__) in c:\cygwin\home\
michi\src\ice\cs\src\Ice\Proxy.cs:line 201
   at Ice.ObjectPrxHelperBase.ice_isA(String id__) in c:\cygwin\ho
```

```
me\michi\src\ice\cs\src\Ice\Proxy.cs:line 170
   at Demo.PrinterPrxHelper.checkedCast(ObjectPrx b) in C:\cygwin\
home\michi\src\ice\cs\demo\book\printer\generated\Prin
ter.cs:line 140
   at Client.Main(String[] args) in C:\cygwin\home\michi\src\ice\c
s\demo\book\printer\Client.cs:line 23
Caused by: System.ComponentModel.Win32Exception: No connection cou
ld be made because the target machine actively refused
 it
```

Note that, to successfully run client and server, the C# run time must be able to locate the `Ice.dll` library. (Under Windows, one way to ensure this is to copy the library into the current directory. Please consult the documentation for your C# run time to see how it locates libraries.)

3.6 Writing an Ice Application with Visual Basic

This section shows how to create an Ice application with Visual Basic.

Overview

As of version 3.3, Ice no longer includes a separate compiler to create Visual Basic source code from Slice definitions. Instead, you need to use the Slice-to-C# compiler **`slice2cs`** to create C# source code and compile the generated C# source code with a C# compiler into a DLL that contains the compiled generated code for your Slice definitions. Your Visual Basic application then links with this DLL and the Ice-for-.NET DLL (`Ice.dll`).[6]

6. This approach works not only with Visual Basic, but with any language that targets the .NET run time. However, ZeroC does not provide support for languages other than C# and Visual Basic.

Figure 3.1 illustrates this development process.

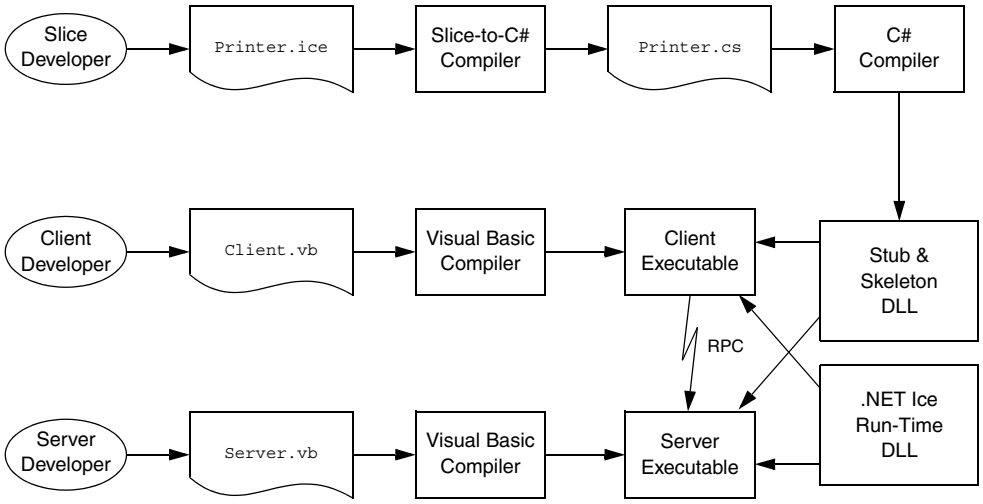


Figure 3.1. Developing a Visual Basic application with Ice.

Compiling a Slice Definition for Visual Basic

The first step in creating our VB application is to compile our Slice definition to generate proxies and skeletons. You can compile the definition as follows:

```
$ mkdir generated
$ slice2cs --output-dir generated Printer.ice
```

The **`--output-dir`** option instructs the compiler to place the generated files into the `generated` directory. This avoids cluttering the working directory with the generated files. The **`slice2cs`** compiler produces a single source file, `Printer.cs`, from this definition. The exact contents of this file do not concern us for now—it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

We now need to compile this generated code into a DLL:

```
$ csc /reference:Ice.dll /lib:%ICE_HOME%\bin /t:library
/out:Printer.dll generated\Printer.cs
```

This creates a DLL called `Printer.dll` that contains the code we generated from the Slice definitions.

Writing and Compiling a Server

To implement our `Printer` interface, we must create a servant class. By convention, servant classes use the name of their interface with an `I`-suffix, so our servant class is called `PrinterI` and placed into a source file `Server.vb`:

```
Imports System
Imports Demo

Public Class PrinterI
    Inherits PrinterDisp_

    Public Overloads Overrides Sub printString( _
                                ByVal s As String, _
                                ByVal current As Ice.Current)
        Console.WriteLine(s)
    End Sub

End Class
```

The `PrinterI` class inherits from a base class called `_PrinterDisp`, which is generated by the **`slice2cs`** compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter. We will see its purpose in detail in Section 32.6.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code follows in `Server.vb` and is shown in full here:

```
Module Server

    Public Sub Main(ByVal args() As String)

        Dim status As Integer = 0
        Dim ic As Ice.Communicator = Nothing
        Try
            ic = Ice.Util.initialize(args)
            Dim adapter As Ice.ObjectAdapter = _
                ic.createObjectAdapterWithEndpoints( _
                    "SimplePrinterAdapter", "default -p 10000")
            Dim obj As Ice.Object = New PrinterI
            adapter.add(obj, ic.stringToIdentity( _
                                        "SimplePrinter"))
            adapter.activate()
            ic.waitForShutdown()
```

```
        Catch e As Exception
            Console.Error.WriteLine(e)
            status = 1
        End Try
        If Not ic Is Nothing Then
            ' Clean up
            '
            Try
                ic.destroy()
            Catch e As Exception
                Console.Error.WriteLine(e)
                status = 1
            End Try
        End If
        Environment.Exit(status)
    End Sub

End module
```

Note the general structure of the code:

```
Module Server

    Public Sub Main(ByVal args() As String)

        Dim status As Integer = 0
        Dim ic As Ice.Communicator = Nothing
        Try

            ' Server implementation here...

        Catch e As Exception
            Console.Error.WriteLine(e)
            status = 1
        End Try
        If Not ic Is Nothing Then
            ' Clean up
            '
            Try
                ic.destroy()
            Catch e As Exception
                Console.Error.WriteLine(e)
                status = 1
            End Try
        End If
```

```
        Environment.Exit(status)
    End Sub

End module
```

The body of `Main` contains a `Try` block in which we place all the server code, followed by a `Catch` block. The catch block catches all exceptions that may be thrown by the code; the intent is that, if the code encounters an unexpected run-time exception anywhere, the stack is unwound all the way back to `Main`, which prints the exception and then returns failure to the operating system.

Before the code exits, it destroys the communicator (if one was created successfully). Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results.

The body of our `Try` block contains the actual server code:

```
ic = Ice.Util.initialize(args)
Dim adapter As Ice.ObjectAdapter = _
    ic.createObjectAdapterWithEndpoints( _
        "SimplePrinterAdapter", "default -p 10000")
Dim obj As Ice.Object = New PrinterI
adapter.add(obj, ic.stringToIdentity( _
                            "SimplePrinter"))
adapter.activate()
ic.waitForShutdown()
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.Util.initialize`. (We pass `args` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns an `Ice::Communicator` reference, which is the main handle to the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWith-Endpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated,

plus an identifier. In this case, the string `"SimplePrinter"` is the name of the servant. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)

5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice ships with such a helper class, called `Ice.Application`—see Section 16.3.1.) As far as actual application code is concerned, the server contains only a few lines: ten lines for the definition of the `PrinterI` class, plus three[7] lines to instantiate a `PrinterI` object and register it with the object adapter.

We can compile the server code as follows:

```
$ vbc /reference:Ice.dll /libpath:%ICE_HOME%\bin
/reference:Printer.dll /out:server.exe Server.vb
```

This compiles our application code and links it with the Ice-for-.NET run time and the DLL we generated earlier. We assume that the **`ICE_HOME`** environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in `C:\opt\Ice`, set **`ICE_HOME`** to that path.)

Writing and Compiling a Client

The client code, in `Client.vb`, looks very similar to the server. Here it is in full:

```
Imports System
Imports Demo

Module Client
```

7. Well, two lines, really: printing space limitations force us to break source lines more often than you would in your actual source files.

```
    Public Sub Main(ByVal args() As String)
        Dim status As Integer = 0
        Dim ic As Ice.Communicator = Nothing
        Try
            ic = Ice.Util.initialize(args)
            Dim obj As Ice.ObjectPrx = ic.stringToProxy( _
                                "SimplePrinter:default -p 10000")
            Dim printer As PrinterPrx = _
                                PrinterPrxHelper.checkedCast(obj)
            If printer Is Nothing Then
                Throw New ApplicationException("Invalid proxy")
            End If

            printer.printString("Hello World!")
        Catch e As Exception
            Console.Error.WriteLine(e)
            status = 1
        End Try
        If Not ic Is Nothing Then
            ' Clean up
            '
            Try
                ic.destroy()
            Catch e As Exception
                Console.Error.WriteLine(e)
                status = 1
            End Try
        End If
        Environment.Exit(status)
    End Sub

End Module
```

Note that the overall code layout is the same as for the server: we use the same `Try` and `Catch` blocks to deal with errors. The code in the `Try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.Util.initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a

bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 38.)

3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `PrinterPrxHelper.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy of type `Demo::Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns null.

4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.

5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling the client looks much the same as for the server:

```
$ vbc /reference:Ice.dll /libpath:%ICE_HOME%\bin
/reference:Printer.dll /out:client.exe Client.vb
```

Running Client and Server

To run client and server, we first start the server in a separate window:

```
$ server.exe
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ client.exe
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in Chapter 16.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Ice.ConnectionRefusedException
    error = 0
   at IceInternal.ProxyFactory.checkRetryAfterException(LocalExcep
tion ex, Reference ref, Int32 cnt) in c:\cygwin\home\m
ichi\src\ice\cs\src\Ice\ProxyFactory.cs:line 167
   at Ice.ObjectPrxHelperBase.handleException__(ObjectDel_ delegat
e, LocalException ex, Int32 cnt) in c:\cygwin\home\mic
hi\src\ice\cs\src\Ice\Proxy.cs:line 970
   at Ice.ObjectPrxHelperBase.ice_isA(String id__, Dictionary`2 co
ntext__, Boolean explicitContext__) in c:\cygwin\home\
michi\src\ice\cs\src\Ice\Proxy.cs:line 201
   at Ice.ObjectPrxHelperBase.ice_isA(String id__) in c:\cygwin\ho
me\michi\src\ice\cs\src\Ice\Proxy.cs:line 170
   at Demo.PrinterPrxHelper.checkedCast(ObjectPrx b) in C:\cygwin\
home\michi\src\ice\cs\demo\book\printer\generated\Prin
ter.cs:line 140
   at Client.Main(String[] args) in C:\cygwin\home\michi\src\ice\c
s\demo\book\printer\Client.cs:line 23
Caused by: System.ComponentModel.Win32Exception: No connection cou
ld be made because the target machine actively refused
 it
```

Note that, to successfully run client and server, the VB run time must be able to locate the `Ice.dll` library. (Under Windows, one way to ensure this is to copy the library into the current directory. Please consult the documentation for your VB run time to see how it locates libraries.)

3.7 Writing an Ice Application with Objective-C

This section shows how to create an Ice application with Objective-C.

Compiling a Slice Definition for Objective-C

The first step in creating our Objective-C application is to compile our Slice definition to generate Objective-C proxies and skeletons. Under Unix, you can compile the definition as follows:

```
$ slice2objc Printer.ice
```

The **`slice2objc`** compiler produces two Objective-C source files from this definition, `Printer.h` and `Printer.m`.

- `Printer.h`

  The `Printer.h` header file contains Objective-C type definitions that correspond to the Slice definitions for our `Printer` interface. This header file must be included in both the client and the server source code.

- `Printer.m`

  The `Printer.m` file contains the source code for our `Printer` interface. The generated source contains type-specific run-time support for both clients and servers. For example, it contains code that marshals parameter data (the string passed to the `printString` operation) on the client side and unmarshals that data on the server side.

  The `Printer.m` file must be compiled and linked into both client and server.

Writing and Compiling a Server

The source code for the server takes only a few lines and is shown in full here:

```
#import <Ice/Ice.h>
#import <Printer.h>

#import <Foundation/NSAutoreleasePool.h>
#import <stdio.h>

@interface PrinterI : DemoPrinter <DemoPrinter>
@end

@implementation PrinterI
-(void) printString:(NSMutableString *)s
                    current:(ICECurrent *)current
{
    printf("%s\n", [s UTF8String]);
}
@end

int
main(int argc, char* argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int status = 1;
    id<ICECommunicator> communicator = nil;
    @try {
        communicator =
            [ICEUtil createCommunicator:&argc argv:argv];
```

```
        id<ICEObjectAdapter> adapter =
            [communicator createObjectAdapterWithEndpoints:
                                    @"SimplePrinterAdapter"
                                    endpoints:@"default -p 10000"];

        ICEObject *object = [[[PrinterI alloc] init] autorelease];
        [adapter add:object identity:[communicator
                                stringToIdentity:@"SimplePrinter"]];
        [adapter activate];

        [communicator waitForShutdown];

        status = 0;
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    @try {
        [communicator destroy];
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    [pool release];
    return status;
}
```

There appears to be a lot of code here for something as simple as a server that just prints a string. Do not be concerned by this: most of the preceding code is boiler plate that never changes. For this very simple server, the code is dominated by this boiler plate.

Every Ice source file starts with an include directive for `Ice.h`, which contains the definitions for the Ice run time. We also include `Printer.h`, which was generated by the Slice compiler and contains the Objective-C definitions for our printer interface. We also import headers to allow us to use an autorelease pool and to produce output:

```
#import <Ice/Ice.h>
#import <Printer.h>

#import <Foundation/NSAutoreleasePool.h>
#import <stdio.h>
```

Our server implements a single printer servant, of type `PrinterI`. Looking at the generated code in `Printer.h`, we find the following (tidied up a little to get rid of irrelevant detail):

```
@protocol DemoPrinter <ICEObject>
-(void) printString:(NSMutableString *)s
                    current:(ICECurrent *)current;
@end

@interface DemoPrinter : ICEObject
// ...
@end
```

The `DemoPrinter` protocol and class definitions are generated by the Slice compiler. The protocol defines the `printString` method, which we must implement in our servant. The `DemoPrinter` class contains methods that are internal to the mapping, so we are not concerned with these. However, our servant must derive from this skeleton class:

```
@interface PrinterI : DemoPrinter <DemoPrinter>
@end

@implementation PrinterI
-(void) printString:(NSMutableString *)s
                    current:(ICECurrent *)current
{
    printf("%s\n", [s UTF8String]);
}
@end
```

As you can see, the implementation of the `printString` method is trivial: it simply writes its string argument to `stdout`.

Note that `printString` has a second parameter of type `ICECurrent`. The Ice run time passes additional information about an incoming request to the servant in this parameter. For now, we will ignore it. (See Section 32.6 for more information about this parameter.)

What follows is the server main program. Note the general structure of the code:

```
int
main(int argc, char* argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int status = 1;
```

```
    id<ICECommunicator> communicator = nil;
    @try {
        communicator =
            [ICEUtil createCommunicator:&argc argv:argv];

        // Server implementation here...

        status = 0;
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    @try {
        [communicator destroy];
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    [pool release];
    return status;
}
```

The body of `main` instantiates an autorelease pool, which it releases before returning to ensure that the program does not leak memory. `main` contains the declaration of two variables, `status` and `communicator`. The `status` variable contains the exit status of the program and the `communicator` variable, of type `id<ICECommunicator>`, contains the main handle to the Ice run time.

Following these declarations is a `try` block in which we place all the server code, followed by a `catch` handler that logs any unhandled exceptions.

Before returning, `main` executes a bit of cleanup code that calls the `destroy` method on the communicator. The cleanup call is outside the first `try` block for a reason: we must ensure that the Ice run time is finalized whether the code terminates normally or terminates due to an exception.[8]

The body of the first `try` block contains the actual server code:

```
        communicator =
            [ICEUtil createCommunicator:&argc argv:argv];

        id<ICEObjectAdapter> adapter =
```

8. Failure to call `destroy` on the communicator before the program exits results in undefined behavior.

```
    [communicator createObjectAdapterWithEndpoints:
                            @"SimplePrinterAdapter"
                            endpoints:@"default -p 10000"];

ICEObject *object = [[[PrinterI alloc] init] autorelease];
[adapter add:object identity:[communicator
                        stringToIdentity:@"SimplePrinter"]];
[adapter activate];

[communicator waitForShutdown];
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `createCommunicator`. (We pass `argc` and `argv` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `createCommunicator` returns a pointer to an `Ice::Communicator` object, which is the main handle to the Ice run time.
2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.
4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the servant. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.) The server starts to process incoming requests from clients as soon as the adapter is activated.
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. As far as actual application code is concerned, the server contains only a few lines: nine lines for the definition of the `PrinterI` class, plus three[9] lines to instantiate a `PrinterI` object and register it with the object adapter.

Assuming that we have the server code in a file called `Server.m`, we can compile it as follows:

```
$ cc -c -I. -I$ICE_HOME/include Printer.m Server.m
```

This compiles both our application code and the code that was generated by the Slice compiler. We assume that the **`ICE_HOME`** environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in `/opt/Ice`, set **`ICE_HOME`** to that path.) Depending on your platform, you may have to add additional include directives or other options to the compiler; please see the demo programs that ship with Ice for the details.

Finally, we need to link the server into an executable:

```
$ c++ Printer.o Server.o -o server \
-L$ICE_HOME/lib -lIceObjC -framework Foundation
```

Again, depending on the platform, the actual list of libraries you need to link against may be longer. The demo programs that ship with Ice contain all the detail.

Writing and Compiling a Client

The client code looks very similar to the server. Here it is in full:

```
#import <Ice/Ice.h>
#import <Printer.h>

#import <Foundation/NSAutoreleasePool.h>
#import <stdio.h>

int
main(int argc, char* argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

9. Well, fewer lines, really: printing space limitations force us to break source lines more often than you would in your actual source files.

```
    int status = 1;
    id<ICECommunicator> communicator = nil;
    @try {
        communicator =
            [ICEUtil createCommunicator:&argc argv:argv];
        id<ICEObjectPrx> base = [communicator
                stringToProxy:@"SimplePrinter:default -p 10000"];
        id<DemoPrinterPrx> printer =
            [DemoPrinterPrx checkedCast:base];
        if(!printer)
            [NSException raise:@"Invalid proxy" format:nil];

        [printer printString:@"Hello World!"];

        status = 0;
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    @try {
        [communicator destroy];
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    [pool release];
    return status;
}
```

Note that the overall code layout is the same as for the server: we include the headers for the Ice run time and the header generated by the Slice compiler, and we use the same `try` block and `catch` handlers to deal with errors.

The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `createCommunicator`.

2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 38.)

3. The proxy returned by `stringToProxy` is of type `id<ICEObjectPrx>`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling the `checkedCast` class method on the `DemoPrinterPrx` class. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Printer` interface?" If so, the call returns a proxy to a `Printer`; otherwise, if the proxy denotes an interface of some other type, the call returns a null proxy.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Compiling and linking the client looks much the same as for the server:

```
$ cc -c -I. -I$ICE_HOME/include Printer.m Client.m
```

```
$ c++ Printer.o Client.o -o client \
-L$ICE_HOME/lib -lIceObjC -framework Foundation
```

Running Client and Server

To run client and server, we first start the server in a separate window:

```
$ ./server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ ./client
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now.

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get:

```
Network.cpp:1218: Ice::ConnectionRefusedException:
connection refused: Connection refused
```

Note that, to successfully run client and server, you may have to set **`DYLD_LIBRARY_PATH`** to include the Ice library directory. Please see the installation instructions and the demo applications that ship with Ice Touch for details.

3.8 Writing an Ice Application with Python

This section shows how to create an Ice application with Python.

Compiling a Slice Definition for Python

The first step in creating our Python application is to compile our Slice definition to generate Python proxies and skeletons. You can compile the definition as follows:[10]

```
$ slice2py Printer.ice
```

The **slice2py** compiler produces a single source file, `Printer_ice.py`, from this definition. The compiler also creates a Python package for the `Demo` module, resulting in a subdirectory named `Demo`. The exact contents of the source file do not concern us for now—it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing a Server

To implement our `Printer` interface, we must create a servant class. By convention, servant classes use the name of their interface with an `I`-suffix, so our servant class is called `PrinterI`:

```
class PrinterI(Demo.Printer):
    def printString(self, s, current=None):
        print s
```

The `PrinterI` class inherits from a base class called `Demo.Printer`, which is generated by the **slice2py** compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.Current` parameter. We will see its purpose in detail in Section 32.6.) Our implementation of the `printString` method simply writes its argument to the terminal.

The remainder of the server code, in `Server.py`, follows our servant class and is shown in full here:

10. Whenever we show Unix commands in this book, we assume a Bourne or Bash shell. The commands for Windows are essentially identical and therefore not shown.

```
import sys, traceback, Ice
import Demo

class PrinterI(Demo.Printer):
    def printString(self, s, current=None):
        print s

status = 0
ic = None
try:
    ic = Ice.initialize(sys.argv)
    adapter = ic.createObjectAdapterWithEndpoints(
        "SimplePrinterAdapter", "default -p 10000")
    object = PrinterI()
    adapter.add(object, ic.stringToIdentity("SimplePrinter"))
    adapter.activate()
    ic.waitForShutdown()
except:
    traceback.print_exc()
    status = 1

if ic:
    # Clean up
    try:
        ic.destroy()
    except:
        traceback.print_exc()
        status = 1

sys.exit(status)
```

Note the general structure of the code:

```
status = 0
ic = None
try:

    # Server implementation here...

except:
    traceback.print_exc()
    status = 1

if ic:
    # Clean up
    try:
```

```
        ic.destroy()
    except:
        traceback.print_exc()
        status = 1

sys.exit(status)
```

The body of the main program contains a `try` block in which we place all the server code, followed by an `except` block. The `except` block catches all exceptions that may be thrown by the code; the intent is that, if the code encounters an unexpected run-time exception anywhere, the stack is unwound all the way back to the main program, which prints the exception and then returns failure to the operating system.

Before the code exits, it destroys the communicator (if one was created successfully). Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results.

The body of our `try` block contains the actual server code:

```
ic = Ice.initialize(sys.argv)
adapter = ic.createObjectAdapterWithEndpoints(
    "SimplePrinterAdapter", "default -p 10000")
object = PrinterI()
adapter.add(object, ic.stringToIdentity("SimplePrinter"))
adapter.activate()
ic.waitForShutdown()
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.initialize`. (We pass `sys.argv` to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to `initialize` returns an `Ice::Communicator` reference, which is the main handle to the Ice run time.

2. We create an object adapter by calling `createObjectAdapterWithEndpoints` on the `Communicator` instance. The arguments we pass are `"SimplePrinterAdapter"` (which is the name of the adapter) and `"default -p 10000"`, which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.

3. At this point, the server-side run time is initialized and we create a servant for our `Printer` interface by instantiating a `PrinterI` object.

4. We inform the object adapter of the presence of a new servant by calling `add` on the adapter; the arguments to `add` are the servant we have just instantiated, plus an identifier. In this case, the string `"SimplePrinter"` is the name of the servant. (If we had multiple printers, each would have a different name or, more correctly, a different *object identity*.)
5. Next, we activate the adapter by calling its `activate` method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
6. Finally, we call `waitForShutdown`. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice ships with such a helper class, called `Ice.Application`—see Section 24.3.1.) As far as actual application code is concerned, the server contains only a few lines: three lines for the definition of the `PrinterI` class, plus two lines to instantiate a `PrinterI` object and register it with the object adapter.

Writing a Client

The client code, in `Client.py`, looks very similar to the server. Here it is in full:

```
import sys, traceback, Ice
import Demo

status = 0
ic = None
try:
    ic = Ice.initialize(sys.argv)
    base = ic.stringToProxy("SimplePrinter:default -p 10000")
    printer = Demo.PrinterPrx.checkedCast(base)
    if not printer:
        raise RuntimeError("Invalid proxy")

    printer.printString("Hello World!")
except:
    traceback.print_exc()
    status = 1
```

```
if ic:
    # Clean up
    try:
        ic.destroy()
    except:
        traceback.print_exc()
        status = 1

sys.exit(status)
```

Note that the overall code layout is the same as for the server: we use the same `try` and `except` blocks to deal with errors. The code in the `try` block does the following:

1. As for the server, we initialize the Ice run time by calling `Ice.initialize`.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 38.)
3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo.PrinterPrx.checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy of type `Demo.PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns `None`.
4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Running Client and Server

To run client and server, we first start the server in a separate window:

```
$ python Server.py
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ python Client.py
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in Chapter 24.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
Traceback (most recent call last):
  File "Client.py", line 10, in ?
    printer = Demo.PrinterPrx.checkedCast(base)
  File "Printer_ice.py", line 43, in checkedCast
    return Demo.PrinterPrx.ice_checkedCast(proxy, '::Demo::Printer
', facet)
ConnectionRefusedException: Ice.ConnectionRefusedException:
Connection refused
```

Note that, to successfully run the client and server, the Python interpreter must be able to locate the Ice extension for Python. See the Ice for Python installation instructions for more information.

3.9 Writing an Ice Application with Ruby

This section shows how to create an Ice client application with Ruby.

Compiling a Slice Definition for Ruby

The first step in creating our Ruby application is to compile our Slice definition to generate Ruby proxies. You can compile the definition as follows:[11]

```
$ slice2rb Printer.ice
```

11. Whenever we show Unix commands in this book, we assume a Bourne or Bash shell. The commands for Windows are essentially identical and therefore not shown.

The **slice2rb** compiler produces a single source file, `Printer.rb`, from this definition. The exact contents of the source file do not concern us for now—it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing a Client

The client code, in `Client.rb`, is shown below in full:

```
require 'Printer.rb'

status = 0
ic = nil
begin
    ic = Ice::initialize(ARGV)
    base = ic.stringToProxy("SimplePrinter:default -p 10000")
    printer = Demo::PrinterPrx::checkedCast(base)
    if not printer
        raise "Invalid proxy"
    end

    printer.printString("Hello World!")
rescue
    puts $!
    puts $!.backtrace.join("\n")
    status = 1
end

if ic
    # Clean up
    begin
        ic.destroy()
    rescue
        puts $!
        puts $!.backtrace.join("\n")
        status = 1
    end
end

exit(status)
```

The program begins with a `require` statement, which loads the Ruby code we generated from our Slice definition in the previous section. It is not necessary for the client to explicitly load the `Ice` module because `Printer.rb` already does that.

The body of the main program contains a `begin` block in which we place all the client code, followed by a `rescue` block. The `rescue` block catches all exceptions that may be thrown by the code; the intent is that, if the code encounters an unexpected run-time exception anywhere, the stack is unwound all the way back to the main program, which prints the exception and then returns failure to the operating system.

The body of our `begin` block goes through the following steps:

1. We initialize the Ice run time by calling `Ice::initialize`. (We pass `ARGV` to this call because the client may have command-line arguments that are of interest to the run time; for this example, the client does not require any command-line arguments.) The call to `initialize` returns an `Ice::Communicator` reference, which is the main handle to the Ice run time.

2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 38.)

3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo::PrinterPrx::checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy of type `Demo::PrinterPrx`; otherwise, if the proxy denotes an interface of some other type, the call returns `nil`.

4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.

5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Before the code exits, it destroys the communicator (if one was created successfully). Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results.

Running the Client

The server must be started before the client. Since Ice for Ruby does not support server-side behavior, we need to use a server from another language mapping. In this case, we will use the C++ server (see Chapter 9):

```
$ server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ ruby Client.rb
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in Chapter 24.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
exception ::Ice::ConnectionRefusedException
{
    error = 111
}
```

Note that, to successfully run the client, the Ruby interpreter must be able to locate the Ice extension for Ruby. See the Ice for Ruby installation instructions for more information.

3.10 Writing an Ice Application with PHP

This section shows how to create an Ice client application with PHP.

Compiling a Slice Definition for PHP

The first step in creating our PHP application is to compile our Slice definition to generate PHP code. You can compile the definition as follows:[12]

12. Whenever we show Unix commands in this book, we assume a Bourne or Bash shell. The commands for Windows are essentially identical and therefore not shown.

```
$ slice2php Printer.ice
```

The **slice2php** compiler produces a single source file, `Printer.php`, from this definition. The exact contents of the source file do not concern us for now—it contains the generated code that corresponds to the `Printer` interface we defined in `Printer.ice`.

Writing a Client

The client code, in `Client.php`, is shown below in full:

```
<?php
require 'Ice.php';
require 'Printer.php';

$ic = null;
try
{
    $ic = Ice_initialize();
    $base = $ic->stringToProxy("SimplePrinter:default -p 10000");
    $printer = Demo_PrinterPrxHelper::checkedCast($base);
    if(!$printer)
        throw new RuntimeException("Invalid proxy");

    $printer->printString("Hello World!");
}
catch(Exception $ex)
{
    echo $ex;
}

if($ic)
{
    // Clean up
    try
    {
        $ic->destroy();
    }
    catch(Exception $ex)
    {
        echo $ex;
    }
}
?>
```

The program begins with `require` statements to load the Ice run-time definitions (`Ice.php`) and the code we generated from our Slice definition in the previous section (`Printer.php`).

The body of the main program contains a `try` block in which we place all the client code, followed by a `catch` block. The `catch` block catches all exceptions that may be thrown by the code; the intent is that, if the code encounters an unexpected run-time exception anywhere, the stack is unwound all the way back to the main program, which prints the exception and then returns failure to the operating system.

The body of our `try` block goes through the following steps:

1. We initialize the Ice run time by calling `Ice_initialize`. The call to `initialize` returns an `Ice::Communicator` reference, which is the main handle to the Ice run time.
2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling `stringToProxy` on the communicator, with the string `"SimplePrinter:default -p 10000"`. Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this in Chapter 38.)
3. The proxy returned by `stringToProxy` is of type `Ice::ObjectPrx`, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a `Demo::Printer` interface, not an `Object` interface. To do this, we need to do a down-cast by calling `Demo_PrinterPrxHelper::checkedCast`. A checked cast sends a message to the server, effectively asking "is this a proxy for a `Demo::Printer` interface?" If so, the call returns a proxy narrowed to the `Printer` interface; otherwise, if the proxy denotes an interface of some other type, the call returns `null`.
4. We test that the down-cast succeeded and, if not, throw an exception that terminates the client.
5. We now have a live proxy in our address space and can call the `printString` method, passing it the time-honored `"Hello World!"` string. The server prints that string on its terminal.

Before the code exits, it destroys the communicator (if one was created successfully). Doing this is essential in order to correctly finalize the Ice run time. If a script neglects to destroy the communicator, Ice destroys it automatically.

Running the Client

The server must be started before the client. Since Ice for PHP does not support server-side behavior, we need to use a server from another language mapping. In this case, we will use the C++ server (see Chapter 9):

```
$ server
```

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window using PHP's command-line interpreter:

```
$ php -f Client.php
$
```

The client runs and exits without producing any output; however, in the server window, we see the `"Hello World!"` that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in Chapter 24.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
exception ::Ice::ConnectionRefusedException
{
    error = 111
}
```

Note that, to successfully run the client, the PHP interpreter must be able to locate the Ice extension for PHP. See the Ice for PHP installation instructions for more information.

3.11 Summary

This chapter presented a very simple (but complete) client and server. As we saw, writing an Ice application involves the following steps:

1. Write a Slice definition and compile it.
2. Write a server and compile it.
3. Write a client and compile it.

If someone else has written the server already and you are only writing a client, you do not need to write the Slice definition, only compile it (and, obviously, you do not need to write the server in that case).

Do not be concerned if, at this point, much appears unclear. The intent of this chapter is to give you an idea of the development process, not to explain the Ice APIs in intricate detail. We will cover all the detail throughout the remainder of this book.

Part II

Slice

Chapter 4
The Slice Language

4.1 Chapter Overview

In this chapter we present the Slice language. We start by discussing the role and purpose of Slice, explaining how language-independent specifications are compiled for particular implementation languages to create actual implementations. Sections 4.10 and 4.11 cover the core Slice concepts of interfaces, operations, exceptions, and inheritance. These concepts have profound influence on the behavior of a distributed system and should be read in detail.

This chapter also presents **`slice2docbook`**, which you can use to automate generation of documentation for Slice definitions.

4.2 Introduction

Slice[1] (Specification Language for Ice) is the fundamental abstraction mechanism for separating object interfaces from their implementations. Slice establishes a contract between client and server that describes the types and object interfaces used by an application. This description is independent of the implementation

1. Even though Slice is an acronym, it is pronounced as single syllable, like a slice of bread.

language, so it does not matter whether the client is written in the same language as the server.

Slice definitions are compiled for a particular implementation language by a compiler. The compiler translates the language-independent definitions into language-specific type definitions and APIs. These types and APIs are used by the developer to provide application functionality and to interact with Ice. The translation algorithms for various implementation languages are known as *language mappings*. Currently, Ice defines language mappings for C++, Java, C#, Python, Objective-C, Ruby, and PHP.

Because Slice describes interfaces and types (but not implementations), it is a purely declarative language; there is no way to write executable statements in Slice.

Slice definitions focus on object interfaces, the operations supported by those interfaces, and exceptions that may be raised by operations. In addition, Slice offers features for object persistence (see Chapter 39). This requires quite a bit of supporting machinery; in particular, quite a bit of Slice is concerned with the definition of data types. This is because data can be exchanged between client and server only if their types are defined in Slice. You cannot exchange arbitrary C++ data between client and server because it would destroy the language independence of Ice. However, you can always create a Slice type definition that corresponds to the C++ data you want to send, and then you can transmit the Slice type.

We present the full syntax and semantics of Slice here. Because much of Slice is based on C++ and Java, we focus on those areas where Slice differs from C++ or Java or constrains the equivalent C++ or Java feature in some way. Slice features that are identical to C++ and Java are mentioned mostly by example.

4.3 Compilation

A Slice compiler produces source files that must be combined with application code to produce client and server executables.

The outcome of the development process is a client executable and a server executable. These executables can be deployed anywhere, whether the target environments use the same or different operating systems and whether the executables are implemented using the same or different languages. The only constraint is that the host machines must provide the necessary run-time environment, such as any required dynamic libraries, and that connectivity can be established between them.

4.3.1 Single Development Environment for Client and Server

Figure 4.1 shows the situation when both client and server are developed in C++. The Slice compiler generates two files from a Slice definition in a source file `Printer.ice`: a header file (`Printer.h`) and a source file (`Printer.cpp`).

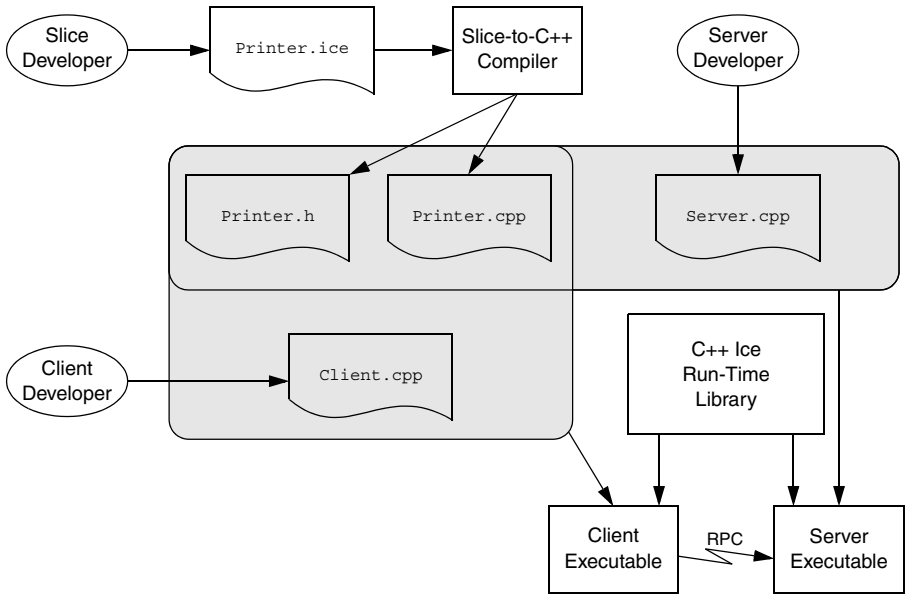


Figure 4.1. Development process if client and server share the same development environment.

- The `Printer.h` header file contains definitions that correspond to the types used in the Slice definition. It is included in the source code of both client and server to ensure that client and server agree about the types and interfaces used by the application.
- The `Printer.cpp` source file provides an API to the client for sending messages to remote objects. The client source code (`Client.cpp`, written by the client developer) contains the client-side application logic. The generated source code and the client code are compiled and linked into the client executable.

  The `Printer.cpp` source file also contains source code that provides an upcall interface from the Ice run time into the server code written by the developer and provides the connection between the networking layer of Ice and the

application code. The server implementation file (`Server.cpp`, written by the server developer) contains the server-side application logic (the object implementations, properly termed *servants*). The generated source code and the implementation source code are compiled and linked into the server executable.

Both client and server also link with an Ice library that provides the necessary run-time support.

You are not limited to a single implementation of a client or server. For example, you can build multiple servers, each of which implements the same interfaces but uses different implementations (for example, with different performance characteristics). Multiple such server implementations can coexist in the same system. This arrangement provides one fundamental scalability mechanism in Ice: if you find that a server process starts to bog down as the number of objects increases, you can run an additional server for the same interfaces on a different machine. Such *federated* servers provide a single logical service that is distributed over a number of processes on different machines. Each server in the federation implements the same interfaces but hosts different object instances. (Of course, federated servers must somehow ensure consistency of any databases they share across the federation.)

Ice also provides support for *replicated* servers. Replication permits multiple servers to each implement the same set of object instances. This improves performance and scalability (because client load can be shared over a number of servers) as well as redundancy (because each object is implemented in more than one server).

4.3.2 Different Development Environments for Client and Server

Client and server cannot share any source or binary components if they are developed in different languages. For example, a client written in Java cannot include a C++ header file.

Figure 4.2 shows the situation when a client written in Java and the corresponding server is written in C++. In this case, the client and server developers are completely independent, and each uses his or her own development environment

and language mapping. The only link between client and server developers is the Slice definition each one uses.

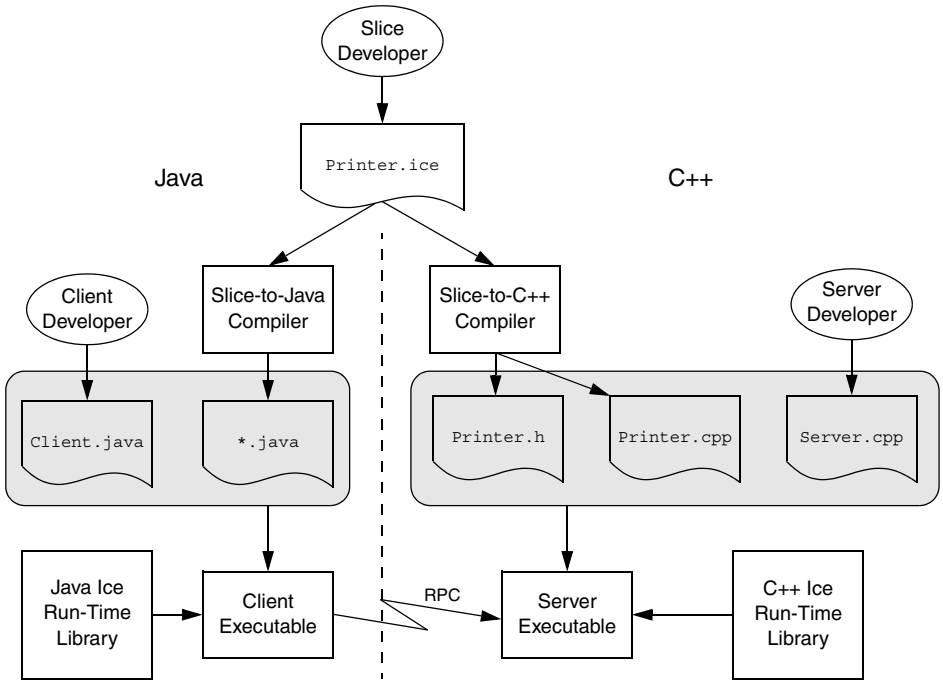


Figure 4.2. Development process for different development environments.

For Java, the slice compiler creates a number of files whose names depend on the names of various Slice constructs. (These files are collectively referred to as `*.java` in Figure 4.2.)

4.4 Source Files

Slice defines a number of rules for the naming and contents of Slice source files.

4.4.1 File Naming

Files containing Slice definitions must end in a `.ice` file extension, for example, `Clock.ice` is a valid file name. Other file extensions are rejected by the compilers.

For case-insensitive file systems (such as DOS), the file extension may be written as uppercase or lowercase, so `Clock.ICE` is legal. For case-sensitive file systems (such as Unix), `Clock.ICE` is illegal. (The extension must be in lowercase.)

4.4.2 File Format

Slice is a free-form language so you can use spaces, horizontal and vertical tab stops, form feeds, and newline characters to lay out your code in any way you wish. (White space characters are token separators). Slice does not attach semantics to the layout of a definition. You may wish to follow the style we have used for the Slice examples throughout this book.

Slice files can be ASCII text files or use the UTF-8 character encoding with a byte order marker (BOM) at the beginning of each file. However, Slice identifiers are limited to ASCII letters and digits; non-ASCII letters can appear only in comments.

4.4.3 Preprocessing

Slice is preprocessed by the C++ preprocessor, so you can use the usual preprocessor directives, such as `#include` and macro definitions. However, Slice permits `#include` directives only at the beginning of a file, before any Slice definitions.

If you use `#include` directives, it is a good idea to protect them with guards to prevent double inclusion of a file:

```
// File Clock.ice
#ifndef _CLOCK_ICE
#define _CLOCK_ICE

// #include directives here...
// Definitions here...

#endif _CLOCK_ICE
```

`#include` directives permit a Slice definition to use types defined in a different source file. The Slice compilers parse all of the code in a source file, including the

code in `#included` files. However, the compilers generate code only for the top-level file(s) nominated on the command line. You must separately compile `#included` files to obtain generated code for all the files that make up your Slice definition.

Note that you should avoid `#include` with double quotes:

```
#include "Clock.ice" // Not recommended!
```

While double quotes will work, the directory in which the preprocessor tries to locate the file can vary depending on the operating system, so the included file may not always be found where you expect it. Instead, use angle brackets (<>); you can control which directories are searched for the file with the **-I** option of the Slice compiler (see page 169).

Also note that, if you include a path separator in a `#include` directive, you must use a forward slash:

```
#include <SliceDefs/Clock.ice>  // OK
```

You cannot use a backslash in `#include` directives:

```
#include <SliceDefs\Clock.ice>  // Illegal
```

4.4.4 Definition Order

Slice constructs, such as modules, interfaces, or type definitions, can appear in any order you prefer. However, identifiers must be declared before they can be used.

4.5 Lexical Rules

Slice's lexical rules are very similar to those of C++ and Java, except for some differences for identifiers.

4.5.1 Comments

Slice definitions permit both the C and the C++ style of writing comments:

```
/*
 * C-style comment.
 */

// C++-style comment extending to the end of this line.
```

4.5.2 Keywords

Slice uses a number of keywords, which must be spelled in lowercase. For example, `class` and `dictionary` are keywords and must be spelled as shown. There are two exceptions to this lowercase rule: `Object` and `LocalObject` are keywords and must be capitalized as shown. You can find a full list of Slice keywords in Appendix A.

4.5.3 Identifiers

Identifiers begin with an alphabetic character followed by any number of alphabetic characters or digits. Underscores are also permitted in identifiers with the following limitations:

- an identifier cannot begin or end with an underscore
- an identifier cannot contain multiple consecutive underscores

Given these rules, the identifier `get_account_name` is legal but not `_account`, `account_`, or `get__account`.

Slice identifiers are restricted to the ASCII range of alphabetic characters and cannot contain non-English letters, such as Å. (Supporting non-ASCII identifiers would make it very difficult to map Slice to target languages that lack support for this feature.)

Case Sensitivity

Identifiers are case-insensitive but must be capitalized consistently. For example, `TimeOfDay` and `TIMEOFDAY` are considered the same identifier within a naming scope. However, Slice enforces consistent capitalization. After you have introduced an identifier, you must capitalize it consistently throughout; otherwise, the compiler will reject it as illegal. This rule exists to permit mappings of Slice to languages that ignore case in identifiers as well as to languages that treat differently capitalized identifiers as distinct.

Identifiers That Are Keywords

You can define Slice identifiers that are keywords in one or more implementation languages. For example, `switch` is a perfectly good Slice identifier but is a C++ and Java keyword. Each language mapping defines rules for dealing with such identifiers. The solution typically involves using a prefix to map away from the keyword. For example, the Slice identifier `switch` is mapped to `_cpp_switch` in C++ and `_switch` in Java.

The rules for dealing with keywords can result in hard-to-read source code. Identifiers such as `native`, `throw`, or `export` will clash with C++ or Java keywords (or both). To make life easier for yourself and others, try to avoid Slice identifiers that are implementation language keywords. Keep in mind that mappings for new languages may be added to Ice in the future. While it is not reasonable to expect you to compile a list of all keywords in all popular programming languages, you should make an attempt to avoid at least common keywords. Slice identifiers such as `self`, `import`, and `while` are definitely not a good idea.

Escaped Identifiers

It is possible to use a Slice keyword as an identifier by prefixing the keyword with a backslash, for example:

```
struct dictionary {     // Error!
    // ...
};

struct \dictionary {    // OK
    // ...
};

struct \foo {           // Legal, same as "struct foo"
    // ...
};
```

The backslash escapes the usual meaning of a keyword; in the preceding example, `\dictionary` is treated as the identifier `dictionary`. The escape mechanism exists to permit keywords to be added to the Slice language over time with minimal disruption to existing specifications: if a pre-existing specification happens to use a newly-introduced keyword, that specification can be fixed by simply prepending a backslash to the new keyword. Note that, as a matter of style, you should avoid using Slice keywords as identifiers (even though the backslash escapes allow you to do this).

It is legal (though redundant) to precede an identifier that is not a keyword with a backslash—the backslash is ignored in that case.

Reserved Identifiers

Slice reserves the identifier `Ice` and all identifiers beginning with `Ice` (in any capitalization) for the Ice implementation. For example, if you try to define a type named `Icecream`, the Slice compiler will issue an error message.[2]

Slice identifiers ending in any of the suffixes `Helper`, `Holder`, `Prx`, and `Ptr` are also reserved. These endings are used by the various language mappings and are reserved to prevent name clashes in the generated code.

4.6 Modules

A common problem in large systems is pollution of the global namespace: over time, as isolated systems are integrated, name clashes become quite likely. Slice provides the `module` construct to alleviate this problem:

```
module ZeroC {
    module Client {
        // Definitions here...
    };
    module Server {
        // Definitions here...
    };
};
```

A module can contain any legal Slice construct, including other module definitions. Using modules to group related definitions together avoids polluting the global namespace and makes accidental name clashes quite unlikely. (You can use a well-known name, such as a company or product name, as the name of the outermost module.)

Slice requires all definitions to be nested inside a module, that is, you cannot define anything other than a module at global scope. For example, the following is illegal:

```
interface I {   // Error: only modules can appear at global scope
    // ...
};
```

Definitions at global scope are prohibited because they cause problems with some implementation languages (such as Python, which does not have a true global scope).

2. You can suppress this behavior by using the `--ice` compiler option, which enables definition of identifiers beginning with `Ice`. However, do not use this option unless you are compiling the Slice definitions for the Ice run time itself.

NOTE: Throughout the remainder of this book, you will occasionally see Slice definitions that are not nested inside a module. This is to keep the examples short and free of clutter. Whenever you see such a definition, assume that it is nested in a module.

Modules can be reopened:

```
module ZeroC {
    // Definitions here...
};

// Possibly in a different source file:

module ZeroC {  // OK, reopened module
    // More definitions here...
};
```

Reopened modules are useful for larger projects: they allow you to split the contents of a module over several different source files. The advantage of doing this is that, when a developer makes a change to one part of the module, only files dependent on the changed part need be recompiled (instead of having to recompile all files that use the module).

Modules map to a corresponding scoping construct in each programming language. (For example, for C++ and C#, Slice modules map to namespaces whereas, for Java, they map to packages.) This allows you to use an appropriate C++ `using` or Java `import` declaration to avoid excessively long identifiers in the source code.

4.7 The `Ice` Module

APIs for the Ice run time, apart from a small number of language-specific calls that cannot be expressed in Ice, are defined in the `Ice` module. In other words, most of the Ice API is actually expressed as Slice definitions. The advantage of doing this is that a single Slice definition is sufficient to define the API for the Ice run time for all supported languages. The respective language mapping rules then determine the exact shape of each Ice API for each implementation language.

We will incrementally explore the contents of the `Ice` module throughout the remainder of this book.

4.8 Basic Slice Types

Slice provides a number of built-in basic types, shown in Table 4.1.

Table 4.1. Slice basic types.

Type	Range of Mapped Type	Size of Mapped Type
`bool`	`false` or `true`	≥ 1bit
`byte`	–128–127 or 0–255[a]	≥ 8 bits
`short`	-2^{15} to $2^{15}-1$	≥ 16 bits
`int`	-2^{31} to $2^{31}-1$	≥ 32 bits
`long`	-2^{63} to $2^{63}-1$	≥ 64 bits
`float`	IEEE single-precision	≥ 32 bits
`double`	IEEE double-precision	≥ 64 bits
`string`	All Unicode characters, excluding the character with all bits zero.	Variable-length

a. The range depends on whether `byte` maps to a signed or an unsigned type.

All the basic types (except `byte`) are subject to changes in representation as they are transmitted between clients and servers. For example, a `long` value is byte-swapped when sent from a little-endian to a big-endian machine. Similarly, strings undergo translation in representation if they are sent, for example, from an EBCDIC to an ASCII implementation, and the characters of a string may also change in size. (Not all architectures use 8-bit characters). However, these changes are transparent to the programmer and do exactly what is required.

4.8.1 Integer Types

Slice provides integer types `short`, `int`, and `long`, with 16-bit, 32-bit, and 64-bit ranges, respectively. Note that, on some architectures, any of these types may be mapped to a native type that is wider. Also note that no unsigned types are provided. (This choice was made because unsigned types are difficult to map into languages without native unsigned types, such as Java. In addition, the unsigned integers add little value to a language. See [9] for a good treatment of the topic.)

4.8.2 Floating-Point Types

These types follow the IEEE specification for single- and double-precision floating-point representation [6]. If an implementation cannot support IEEE format floating-point values, the Ice run time converts values into the native floating-point representation (possibly at a loss of precision or even magnitude, depending on the capabilities of the native floating-point format).

4.8.3 Strings

Slice strings use the Unicode character set. The only character that cannot appear inside a string is the zero character.[3]

The Slice data model does *not* have the concept of a null string (in the sense of a C++ null pointer). This decision was made because null strings are difficult to map to languages without direct support for this concept (such as Python). Do not design interfaces that depend on a null string to indicate "not there" semantics. If you need the notion of an optional string, use a class (see Section 4.11), a sequence of strings (see Section 4.9.3), or use an empty string to represent the idea of a null string. (Of course, the latter assumes that the empty string is not otherwise used as a legitimate string value by your application.)

4.8.4 Booleans

Boolean values can have only the values `false` and `true`. Language mappings use the corresponding native boolean type if one is available.

4.8.5 Bytes

The Slice type `byte` is an (at least) 8-bit type that is guaranteed not to undergo any changes in representation as it is transmitted between address spaces. This guarantee permits exchange of binary data such that it is not tampered with in transit. All other Slice types are subject to changes in representation during transmission.

3. This decision was made as a concession to C++, with which it becomes impossibly difficult to manipulate strings with embedded zero characters using standard library routines, such as `strlen` or `strcat`.

4.9 User-Defined Types

In addition to providing the built-in basic types, Slice allows you to define complex types: enumerations, structures, sequences, and dictionaries.

4.9.1 Enumerations

A Slice enumerated type definition looks like the C++ version:

```
enum Fruit { Apple, Pear, Orange };
```

This definition introduces a type named `Fruit` that becomes a new type in its own right. Slice does not define how ordinal values are assigned to enumerators. For example, you cannot assume that the enumerator `Orange` will have the value 2 in different implementation languages. Slice guarantees only that the ordinal values of enumerators increase from left to right, so `Apple` compares less than `Pear` in all implementation languages.

Unlike C++, Slice does not permit you to control the ordinal values of enumerators (because many implementation languages do not support such a feature):

```
enum Fruit { Apple = 0, Pear = 7, Orange = 2 }; // Syntax error
```

In practice, you do not care about the values used for enumerators as long as you do not transmit the *ordinal value* of an enumerator between address spaces. For example, sending the value 0 to a server to mean `Apple` can cause problems because the server may not use 0 to represent `Apple`. Instead, simply send the value `Apple` itself. If `Apple` is represented by a different ordinal value in the receiving address space, that value will be appropriately translated by the Ice run time.

As with C++, Slice enumerators enter the enclosing namespace, so the following is illegal:

```
enum Fruit { Apple, Pear, Orange };
enum ComputerBrands { Apple, IBM, Sun, HP };    // Apple redefined
```

Slice does not permit empty enumerations.

4.9.2 Structures

Slice supports structures containing one or more named members of arbitrary type, including user-defined complex types. For example:

```
struct TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};
```

As in C++, this definition introduces a new type called `TimeOfDay`. Structure definitions form a namespace, so the names of the structure members need to be unique only within their enclosing structure.

Data member definitions using a named type are the only construct that can appear inside a structure. It is impossible to, for example, define a structure inside a structure:

```
struct TwoPoints {
    struct Point {      // Illegal!
                short x;
                short y;
    };
    Point       coord1;
    Point       coord2;
};
```

This rule applies to Slice in general: type definitions cannot be nested (except for modules, which do support nesting—see Section 4.6). The reason for this rule is that nested type definitions can be difficult to implement for some target languages and, even if implementable, greatly complicate the scope resolution rules. For a specification language, such as Slice, nested type definitions are unnecessary—you can always write the above definitions as follows (which is stylistically cleaner as well):

```
struct Point {
    short x;
    short y;
};

struct TwoPoints {      // Legal (and cleaner!)
    Point coord1;
    Point coord2;
};
```

You can specify a default value for a data member that has one of the following types:

- An integral type (`bool`, `byte`, `short`, `int`, `long`, or an enumerated type)
- `float` or `double`

- `string`

For example:

```
struct Location {
    string name;
    Point pt;
    bool display = true;
    string source = "GPS";
};
```

The legal syntax for literal values is the same as for Slice constants (see Section 4.9.5). The language mapping guarantees that data members are initialized to their declared default values using a language-specific mechanism.

4.9.3 Sequences

Sequences are variable-length collections of elements:

```
sequence<Fruit> FruitPlatter;
```

A sequence can be empty—that is, it can contain no elements, or it can hold any number of elements up to the memory limits of your platform.

Sequences can contain elements that are themselves sequences. This arrangement allows you to create lists of lists:

```
sequence<FruitPlatter> FruitBanquet;
```

Sequences are used to model a variety of collections, such as vectors, lists, queues, sets, bags, or trees. (It is up to the application to decide whether or not order is important; by discarding order, a sequence serves as a set or bag.)

One particular use of sequences has become idiomatic, namely, the use of a sequence to indicate an optional value. For example, we might have a `Part` structure that records the details of the parts that go into a car. The structure could record things such as the name of the part, a description, weight, price, and other details. Spare parts commonly have a serial number, which we can model as a `long` value. However, some parts, such as simple screws, often do not have a serial number, so what are we supposed to put into the serial number field of a screw? There are a number of options for dealing with this situation:

- Use a sentinel value, such as zero, to indicate the “no serial number” condition.

  This approach is workable, provided that a sentinel value is actually available. While it may seem unlikely that anyone would use a serial number of zero for

a part, it is not impossible. And, for other values, such as a temperature value, all values in the range of their type can be legal, so no sentinel value is available.

- Change the type of the serial number from `long` to `string`.

  Strings come with their own built-in sentinel value, namely, the empty string so we can use an empty string to indicate the "no serial number" case. This is workable, but leaves a bad taste in most people's mouth: we should not have to change the natural data type of something to `string` just so we get a sentinel value.

- Add an indicator as to whether the contents of the serial number are valid:

```
struct Part {
    string name;
    string description;
    // ...
    bool   serialIsValid;  // true if part has serial number
    long   serialNumber;
};
```

  This is distasteful to most people and guaranteed to get you into trouble eventually: sooner or later, some programmer will forget to check whether the serial number is valid before using it and create havoc.

- Use a sequence to model the optional field.

  This technique uses the following convention:

```
sequence<long> SerialOpt;

struct Part {
    string    name;
    string    description;
    // ...
    SerialOpt serialNumber; // optional: zero or one element
};
```

  By convention, the `Opt` suffix is used to indicate that the sequence is used to model an optional value. If the sequence is empty, the value is obviously not there; if it contains a single element, that element is the value. The obvious drawback of this scheme is that someone could put more than one element into the sequence. This could be rectified by adding a special-purpose Slice construct for optional values. However, optional values are not used frequently enough to justify the complexity of adding a dedicated language

feature. (As we will see in Section 4.11, you can also use class hierarchies to model optional fields.)

4.9.4 Dictionaries

A dictionary is a mapping from a key type to a value type. For example:

```
struct Employee {
    long   number;
    string firstName;
    string lastName;
};

dictionary<long, Employee> EmployeeMap;
```

This definition creates a dictionary named `EmployeeMap` that maps from an employee number to a structure containing the details for an employee. Whether or not the key type (the employee number, of type `long` in this example) is also part of the value type (the `Employee` structure in this example) is up to you—as far as Slice is concerned, there is no need to include the key as part of the value.

Dictionaries can be used to implement sparse arrays, or any lookup data structure with non-integral key type. Even though a sequence of structures containing key–value pairs could be used to model the same thing, a dictionary is more appropriate:

- A dictionary clearly signals the intent of the designer, namely, to provide a mapping from a domain of values to a range of values. (A sequence of structures of key–value pairs does not signal that same intent as clearly.)
- At the programming language level, sequences are implemented as vectors (or possibly lists), that is, they are not well suited to model sparsely populated domains and require a linear search to locate an element with a particular value. On the other hand, dictionaries are implemented as a data structure (typically a hash table or red-black tree) that supports efficient searching in $O(\log n)$ average time or better.

The key type of a dictionary need not be an integral type. For example, we could use the following definition to translate the names of the days of the week:

```
dictionary<string, string> WeekdaysEnglishToGerman;
```

The server implementation would take care of initializing this map with the key–value pairs `Monday-Montag`, `Tuesday-Dienstag`, and so on.

The value type of a dictionary can be any user-defined type. However, the key type of a dictionary is limited to one of the following types:

- Integral types (`byte`, `short`, `int`, `long`, `bool`, and enumerated types)
- `string`
- structures containing only data members of integral type or `string`

Complex nested types, such as nested structures, sequences, or dictionaries, and floating-point types (`float` and `double`) cannot be used as the key type. Complex nested types are disallowed because they complicate the language mappings for dictionaries, and floating-point types are disallowed because representational changes of values as they cross machine boundaries can lead to ill-defined semantics for equality.

4.9.5 Constant Definitions and Literals

Slice allows you to define constants. Constant definitions must be of one of the following types:

- An integral type (`bool`, `byte`, `short`, `int`, `long`, or an enumerated type)
- `float` or `double`
- `string`

Here are a few examples:

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

The syntax for literals is the same as for C++ and Java (with a few minor exceptions):

- Boolean constants can only be initialized with the keywords `false` and `true`. (You cannot use `0` and `1` to represent `false` and `true`.)
- As for C++, integer literals can be specified in decimal, octal, or hexadecimal notation. For example:

```
const byte TheAnswer = 42;
const byte TheAnswerInOctal = 052;
const byte TheAnswerInHex = 0x2A;       // or 0x2a
```

Be aware that, if you interpret `byte` as a number instead of a bit pattern, you may get different results in different languages. For example, for C++, `byte` maps to `unsigned char` whereas, for Java, `byte` maps to `byte`, which is a signed type.

Note that suffixes to indicate long and unsigned constants (`l`, `L`, `u`, `U`, used by C++) are illegal:

```
const long Wrong = 0u;          // Syntax error
const long WrongToo = 1000000L; // Syntax error
```

The value of an integer literal must be within the range of its constant type, as shown in Table 4.1 on page 96; otherwise the compiler will issue a diagnostic.

- Floating-point literals use C++ syntax, except that you cannot use an `l` or `L` suffix to indicate an extended floating-point constant; however, `f` and `F` are legal (but are ignored). Here are a few examples:

```
const float P1 = -3.14f;    // Integer & fraction, with suffix
const float P2 = +3.1e-3;   // Integer, fraction, and exponent
const float P3 = .1;        // Fraction part only
const float P4 = 1.;        // Integer part only
const float P5 = .9E5;      // Fraction part and exponent
const float P6 = 5e2;       // Integer part and exponent
```

Floating-point literals must be within the range of the constant type (`float` or `double`); otherwise, the compiler will issue a diagnostic.

- String literals support the same escape sequences as C++. Here are some examples:

```
const string AnOrdinaryString = "Hello World!";

const string DoubleQuote =      "\"";
const string TwoSingleQuotes =  "'\'";     // ' and \' are OK
const string Newline =          "\n";
const string CarriageReturn =   "\r";
const string HorizontalTab =    "\t";
const string VerticalTab =      "\v";
const string FormFeed =         "\f";
const string Alert =            "\a";
const string Backspace =        "\b";
const string QuestionMark =     "\?";
```

```
const string Backslash =        "\\";

const string OctalEscape =      "\007";    // Same as \a
const string HexEscape =        "\x07";    // Ditto

const string UniversalCharName = "\u03A9"; // Greek Omega
```

As for C++, adjacent string literals are concatenated:

```
const string MSG1 = "Hello World!";
const string MSG2 = "Hello" " " "World!";       // Same message

/*
 * Escape sequences are processed before concatenation,
 * so the string below contains two characters,
 * '\xa' and 'c'.
 */

const string S = "\xa" "c";
```

Note that Slice has no concept of a null string:

```
const string nullString = 0;    // Illegal!
```

Null strings simply do not exist in Slice and, therefore, do not exist as a legal value for a string anywhere in the Ice platform. The reason for this decision is that null strings do not exist in many programming languages.[4]

4.10 Interfaces, Operations, and Exceptions

The central focus of Slice is on defining interfaces, for example:

```
struct TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};
```

4. Many languages other than C and C++ use a byte array as the internal string representation. Null strings do not exist (and would be very difficult to map) in such languages.

```
interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};
```

This definition defines an interface type called `Clock`. The interface supports two operations: `getTime` and `setTime`. Clients access an object supporting the `Clock` interface by invoking an operation on the proxy for the object: to read the current time, the client invokes the `getTime` operation; to set the current time, the client invokes the `setTime` operation, passing an argument of type `TimeOfDay`.

Invoking an operation on a proxy instructs the Ice run time to send a message to the target object. The target object can be in another address space or can be collocated (in the same process) as the caller—the location of the target object is transparent to the client. If the target object is in another (possibly remote) address space, the Ice run time invokes the operation via a remote procedure call; if the target is collocated with the client, the Ice run time uses an ordinary function call instead, to avoid the overhead of marshaling.

You can think of an interface definition as the equivalent of the public part of a C++ class definition or as the equivalent of a Java interface, and of operation definitions as (virtual) member functions. Note that nothing but operation definitions are allowed to appear inside an interface definition. In particular, you cannot define a type, an exception, or a data member inside an interface. This does not mean that your object implementation cannot contain state—it can, but how that state is implemented (in the form of data members or otherwise) is hidden from the client and, therefore, need not appear in the object's interface definition.

An Ice object has exactly one (most derived) Slice interface type (or class type—see Section 4.11). Of course, you can create multiple Ice objects that have the same type; to draw the analogy with C++, a Slice interface corresponds to a C++ class *definition*, whereas an Ice object corresponds to a C++ class *instance* (but Ice objects can be implemented in multiple different address spaces).

Ice also provides multiple interfaces via a feature called *facets*. We discuss facets in detail in Chapter 33.

A Slice interface defines the smallest grain of distribution in Ice: each Ice object has a unique identity (encapsulated in its proxy) that distinguishes it from all other Ice objects; for communication to take place, you must invoke operations on an object's proxy. There is no other notion of an addressable entity in Ice. You cannot, for example, instantiate a Slice structure and have clients manipulate that structure remotely. To make the structure accessible, you must create an interface that allows clients to access the structure.

The partition of an application into interfaces therefore has profound influence on the overall architecture. Distribution boundaries must follow interface (or class) boundaries; you can spread the implementation of interfaces over multiple address spaces (and you can implement multiple interfaces in the same address space), but you cannot implement parts of interfaces in different address spaces.

4.10.1 Parameters and Return Values

An operation definition must contain a return type and zero or more parameter definitions. For example, the `getTime` operation on page 105 has a return type of `TimeOfDay` and the `setTime` operation has a return type of `void`. You must use `void` to indicate that an operation returns no value—there is no default return type for Slice operations.

An operation can have one or more input parameters. For example, `setTime` accepts a single input parameter of type `TimeOfDay` called `time`. Of course, you can use multiple input parameters, for example:

```
interface CircadianRhythm {
    void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);
    // ...
};
```

Note that the parameter name (as for Java) is mandatory. You cannot omit the parameter name, so the following is in error:

```
interface CircadianRhythm {
    void setSleepPeriod(TimeOfDay, TimeOfDay);  // Error!
    // ...
};
```

By default, parameters are sent from the client to the server, that is, they are input parameters. To pass a value from the server to the client, you can use an output parameter, indicated by the `out` keyword. For example, an alternative way to define the `getTime` operation on page 105 would be:

```
void getTime(out TimeOfDay time);
```

This achieves the same thing but uses an output parameter instead of the return value. As with input parameters, you can use multiple output parameters:

```
interface CircadianRhythm {
    void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);
    void getSleepPeriod(out TimeOfDay startTime,
                        out TimeOfDay stopTime);
    // ...
};
```

If you have both input and output parameters for an operation, the output parameters must follow the input parameters:

```
void changeSleepPeriod(    TimeOfDay startTime,     // OK
                           TimeOfDay stopTime,
                       out TimeOfDay prevStartTime,
                       out TimeOfDay prevStopTime);
void changeSleepPeriod(out TimeOfDay prevStartTime,
                       out TimeOfDay prevStopTime,
                           TimeOfDay startTime,     // Error
                           TimeOfDay stopTime);
```

Slice does not support parameters that are both input and output parameters (call by reference). The reason is that, for remote calls, reference parameters do not result in the same savings that one can obtain for call by reference in programming languages. (Data still needs to be copied in both directions and any gains in marshaling efficiency are negligible.) Also, reference (or input–output) parameters result in more complex language mappings, with concomitant increases in code size.

Style of Operation Definition

As you would expect, language mappings follow the style of operation definition you use in Slice: Slice return types map to programming language return types, and Slice parameters map to programming language parameters.

For operations that return only a single value, it is common to return the value from the operation instead of using an out-parameter. This style maps naturally into all programming languages. Note that, if you use an out-parameter instead, you impose a different API style on the client: most programming languages permit the return value of a function to be ignored whereas it is typically not possible to ignore an output parameter.

For operations that return multiple values, it is common to return all values as out-parameters and to use a return type of `void`. However, the rule is not all that clear-cut because operations with multiple output values can have one particular value that is considered more “important” than the remainder. A common example of this is an iterator operation that returns items from a collection one-by-one:

```
bool next(out RecordType r);
```

The `next` operation returns two values: the record that was retrieved and a Boolean to indicate the end-of-collection condition. (If the return value is `false`, the end of the collection has been reached and the parameter `r` has an undefined value.) This style of definition can be useful because it naturally fits into the way programmers write control structures. For example:

```
while (next(record))
    // Process record...

if (next(record))
    // Got a valid record...
```

Overloading

Slice does not support any form of overloading of operations. For example:

```
interface CircadianRhythm {
    void modify(TimeOfDay startTime,
                TimeOfDay endTime);
    void modify(    TimeOfDay startTime,        // Error
                    TimeOfDay endTime,
                out timeOfDay prevStartTime,
                out TimeOfDay prevEndTime);
};
```

Operations in the same interface must have different names, regardless of what type and number of parameters they have. This restriction exists because overloaded functions cannot sensibly be mapped to languages without built-in support for overloading.[5]

Idempotent Operations

Some operations, such as `getTime` on page 105, do not modify the state of the object they operate on. They are the conceptual equivalent of C++ `const` member functions. Similary, `setTime` does modify the state of the object, but is idempotent. You can indicate this in Slice as follows:

5. Name mangling is not an option in this case: while it works fine for compilers, it is unacceptable to humans.

```
interface Clock {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};
```

This marks the `getTime` and `setTime` operations as idempotent. An operation is idempotent if two successive invocations of the operation have the same effect as a single invocation. For example, `x = 1;` is an idempotent operation because it does not matter whether it is executed once or twice—either way, `x` ends up with the value 1. On the other hand, `x += 1;` is not an idempotent operation because executing it twice results in a different value for `x` than executing it once. Obviously, any read-only operation is idempotent.

The `idempotent` keyword is useful because it allows the Ice run time to attempt more aggressive error recovery. Specifically, Ice guarantees *at-most-once* semantics for operation invocations:

- For normal (not idempotent) operations, the Ice run time has to be conservative about how it deals with errors. For example, if a client sends an operation invocation to a server and then loses connectivity, there is no way for the client-side run time to find out whether the request it sent actually made it to the server. This means that the run time cannot attempt to recover from the error by re-establishing a connection and sending the request a second time because that could cause the operation to be invoked a second time and violate at-most-once semantics; the run time has no option but to report the error to the application.
- For `idempotent` operations, on the other hand, the client-side run time can attempt to re-establish a connection to the server and safely send the failed request a second time. If the server can be reached on the second attempt, everything is fine and the application never notices the (temporary) failure. Only if the second attempt fails need the run time report the error back to the application. (The number of retries can be increased with an Ice configuration parameter.)

4.10.2 User Exceptions

Looking at the `setTime` operation on page 105, we find a potential problem: given that the `TimeOfDay` structure uses `short` as the type of each field, what will happen if a client invokes the `setTime` operation and passes a `TimeOfDay` value with meaningless field values, such as `-199` for the minute field, or `42` for the hour? Obviously, it would be nice to provide some indication to the caller that this

is meaningless. Slice allows you to define user exceptions to indicate error conditions to the client. For example:

```
exception Error {}; // Empty exceptions are legal

exception RangeError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};
```

A user exception is much like a structure in that it contains a number of data members. However, unlike structures, exceptions can have zero data members, that is, be empty.

You can specify a default value for an exception data member that has one of the following types:

- An integral type (`bool`, `byte`, `short`, `int`, `long`, or an enumerated type)
- `float` or `double`
- `string`

For example:

```
exception RangeError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
    string reason = "out of range";
};
```

The legal syntax for literal values is the same as for Slice constants (see Section 4.9.5). The language mapping guarantees that data members are initialized to their declared default values using a language-specific mechanism.

Exceptions allow you to return an arbitrary amount of error information to the client if an error condition arises in the implementation of an operation. Operations use an exception specification to indicate the exceptions that may be returned to the client:

```
interface Clock {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time)
                        throws RangeError, Error;
};
```

This definition indicates that the `setTime` operation may throw either a `RangeError` or an `Error` user exception (and no other type of exception). If the client receives a `RangeError` exception, the exception contains the `TimeOfDay` value that was passed to `setTime` and caused the error (in the `errorTime` member), as well as the minimum and maximum time values that can be used (in the `minTime` and `maxTime` members). If `setTime` failed because of an error not caused by an illegal parameter value, it throws `Error`. Obviously, because `Error` does not have data members, the client will have no idea what exactly it was that went wrong—it simply knows that the operation did not work.

An operation can throw only those user exceptions that are listed in its exception specification. If, at run time, the implementation of an operation throws an exception that is not listed in its exception specification, the client receives a run-time exception (see Section 4.10.4) to indicate that the operation did something illegal. To indicate that an operation does not throw any user exception, simply omit the exception specification. (There is no empty exception specification in Slice.)

Exceptions are not first-class data types and first-class data types are not exceptions:

- You cannot pass an exception as a parameter value.
- You cannot use an exception as the type of a data member.
- You cannot use an exception as the element type of a sequence.
- You cannot use an exception as the key or value type of a dictionary.
- You cannot throw a value of non-exception type (such as a value of type `int` or `string`).

The reason for these restrictions is that some implementation languages use a specific and separate type for exceptions (in the same way as Slice does). For such languages, it would be difficult to map exceptions if they could be used as an ordinary data type. (C++ is somewhat unusual among programming languages by allowing arbitrary types to be used as exceptions.)

4.10.3 Exception Inheritance

Exceptions support inheritance. For example:

```
exception ErrorBase {
    string reason;
};
```

```
enum RTError {
    DivideByZero, NegativeRoot, IllegalNull /* ... */
};

exception RuntimeError extends ErrorBase {
    RTError err;
};

enum LError { ValueOutOfRange, ValuesInconsistent, /* ... */ };

exception LogicError extends ErrorBase {
    LError err;
};

exception RangeError extends LogicError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};
```

These definitions set up a simple exception hierarchy:

- `ErrorBase` is at the root of the tree and contains a string explaining the cause of the error.
- Derived from `ErrorBase` are `RuntimeError` and `LogicError`. Each of these exceptions contains an enumerated value that further categorizes the error.
- Finally, `RangeError` is derived from `LogicError` and reports the details of the specific error.

Setting up exception hierarchies such as this not only helps to create a more readable specification because errors are categorized, but also can be used at the language level to good advantage. For example, the Slice C++ mapping preserves the exception hierarchy so you can catch exceptions generically as a base exception, or set up exception handlers to deal with specific exceptions.

Looking at the exception hierarchy on page 112, it is not clear whether, at run time, the application will only throw most derived exceptions, such as `RangeError`, or if it will also throw base exceptions, such as `LogicError`, `RuntimeError`, and `ErrorBase`. If you want to indicate that a base exception, interface, or class is abstract (will not be instantiated), you can add a comment to that effect.

Note that, if the exception specification of an operation indicates a specific exception type, at run time, the implementation of the operation may also throw more derived exceptions. For example:

```
exception Base {
    // ...
};

exception Derived extends Base {
    // ...
};

interface Example {
    void op() throws Base;      // May throw Base or Derived
};
```

In this example, `op` may throw a `Base` or a `Derived` exception, that is, any exception that is compatible with the exception types listed in the exception specification can be thrown at run time.

As a system evolves, it is quite common for new, derived exceptions to be added to an existing hierarchy. Assume that we initially construct clients and server with the following definitions:

```
exception Error {
    // ...
};

interface Application {
    void doSomething() throws Error;
};
```

Also assume that a large number of clients are deployed in field, that is, when you upgrade the system, you cannot easily upgrade all the clients. As the application evolves, a new exception is added to the system and the server is redeployed with the new definition:

```
exception Error {
    // ...
};

exception FatalApplicationError extends Error {
    // ...
};

interface Application {
    void doSomething() throws Error;
};
```

This raises the question of what should happen if the server throws a `FatalApplicationError` from `doSomething`. The answer depends whether the client was built using the old or the updated definition:

- If the client was built using the same definition as the server, it simply receives a `FatalApplicationError`.
- If the client was built with the original definition, that client has no knowledge that `FatalApplicationError` even exists. In this case, the Ice run time automatically slices the exception to the most-derived type that is understood by the receiver (`Error`, in this case) and discards the information that is specific to the derived part of the exception. (This is exactly analogous to catching C++ exceptions by value—the exception is sliced to the type used in the `catch`-clause.)

Exceptions support single inheritance only. (Multiple inheritance would be difficult to map into many programming languages.)

4.10.4 Ice Run-Time Exceptions

As mentioned in Section 2.2.2, in addition to any user exceptions that are listed in an operation's exception specification, an operation can also throw Ice *run-time exceptions*. Run-time exceptions are predefined exceptions that indicate platform-related run-time errors. For example, if a networking error interrupts communication between client and server, the client is informed of this by a run-time exception, such as `ConnectTimeoutException` or `SocketException`.

The exception specification of an operation must not list any run-time exceptions. (It is understood that all operations can raise run-time exceptions and you are not allowed to restate that.)

Inheritance Hierarchy for Exceptions

All the Ice run-time and user exceptions are arranged in an inheritance hierarchy, as shown in Figure 4.3.

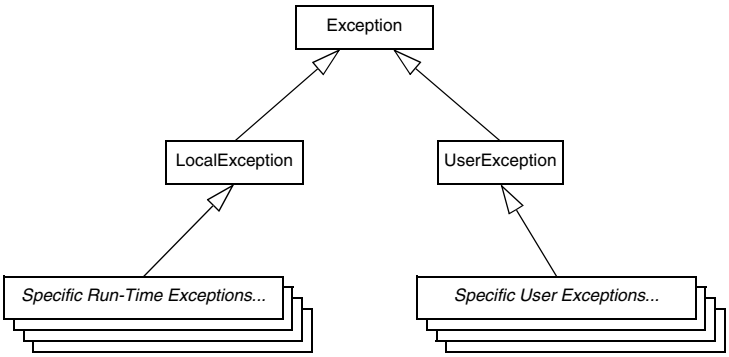


Figure 4.3. Inheritance structure for exceptions.

`Ice::Exception` is at the root of the inheritance hierarchy. Derived from that are the (abstract) types `Ice::LocalException` and `Ice::UserException`. In turn, all run-time exceptions are derived from `Ice::LocalException`, and all user exceptions are derived from `Ice::UserException`.

Figure 4.4 shows the complete hierarchy of the Ice run-time exceptions.[6]

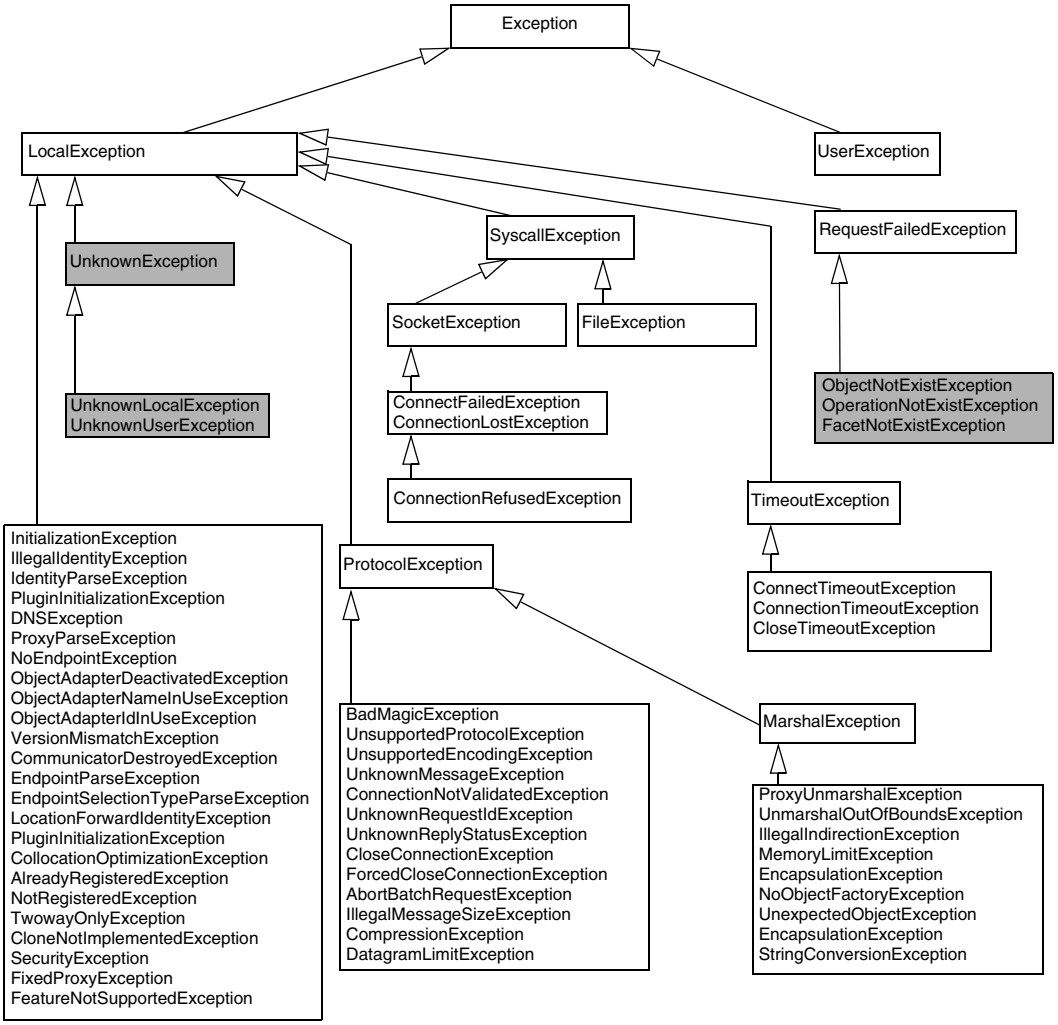


Figure 4.4. Ice run-time exception hierarchy. (Shaded exceptions can be sent by the server.)

6. We use the Unified Modeling Language (UML) for the object model diagrams in this book (see [1] and [13] for details).

Note that Figure 4.4 groups several exceptions into a single box to save space (which, strictly, is incorrect UML syntax). Also note that some run-time exceptions have data members, which, for brevity, we have omitted in Figure 4.4. These data members provide additional information about the precise cause of an error.

Many of the run-time exceptions have self-explanatory names, such as `MemoryLimitException`. Others indicate problems in the Ice run time, such as `EncapsulationException`. Still others can arise only through application programming errors, such as `TwowayOnlyException`. In practice, you will likely never see most of these exceptions. However, there are a few run-time exceptions you will encounter and whose meaning you should know.

Local Versus Remote Exceptions

Most error conditions are detected on the client side. For example, if an attempt to contact a server fails, the client-side run time raises a `ConnectTimeoutException`. However, there are three specific error conditions (shaded in Figure 4.4) that are detected by the server and made known explicitly to the client-side run time via the Ice protocol:

- `ObjectNotExistException`

  This exception indicates that a request was delivered to the server but the server could not locate a servant with the identity that is embedded in the proxy. In other words, the server could not find an object to dispatch the request to.

  An `ObjectNotExistException` is a death certificate: it indicates that the target object in the server does not exist.[7] Most likely, this is the case because the object existed some time in the past and has since been destroyed, but the same exception is also raised if a client uses a proxy with the identity of an object that has never been created. If you receive this exception, you are expected to clean up whatever resources you might have allocated that relate to the specific object for which you receive this exception.

- `FacetNotExistException`

  The client attempted to contact a non-existent facet of an object, that is, the server has at least one servant with the given identity, but no servant with a matching facet name. (See Chapter 33 for a discussion of facets.)

7. The Ice run time raises `ObjectNotExistException` only if there are no facets in existence with a matching identity; otherwise, it raises `FacetNotExistException` (see Chapter 33).

- `OperationNotExistException`

  This exception is raised if the server could locate an object with the correct identity but, on attempting to dispatch the client's operation invocation, the server found that the target object does not have such an operation. You will see this exception in only two cases:

  - You have used an unchecked down-cast on a proxy of the incorrect type. (See page 208 and page 345 for unchecked down-casts.)
  - Client and server have been built with Slice definitions for an interface that disagree with each other, that is, the client was built with an interface definition for the object that indicates that an operation exists, but the server was built with a different version of the interface definition in which the operation is absent.

Any error condition on the server side that is not described by one of the three preceding exceptions is made known to the client as one of three generic exceptions (shaded in Figure 4.4):

- `UnknownUserException`

  This exception indicates that an operation implementation has thrown a Slice exception that is not declared in the operation's exception specification (and is not derived from one of the exceptions in the operation's exception specification).

- `UnknownLocalException`

  If an operation implementation raises a run-time exception other than `ObjectNotExistException`, `FacetNotExistException`, or `OperationNotExistException` (such as a `NotRegisteredException`), the client receives an `UnknownLocalException`. In other words, the Ice protocol does not transmit the exact exception that was encountered in the server, but simply returns a bit to the client in the reply to indicate that the server encountered a run-time exception.

  A common cause for a client receiving an `UnknownLocalException` is failure to catch and handle all exceptions in the server. For example, if the implementation of an operation encounters an exception it does not handle, the exception propagates all the way up the call stack until the stack is unwound to the point where the Ice run time invoked the operation. The Ice run time catches all Ice exceptions that "escape" from an operation invocation and returns them to the client as an `UnknownLocalException`.

- `UnknownException`

  An operation has thrown a non-Ice exception. For example, if the operation in the server throws a C++ exception, such as a `char *`, or a Java exception, such as a `ClassCastException`, the client receives an `UnknownException`.

All other run-time exceptions (not shaded in Figure 4.4) are detected by the client-side run time and are raised locally.

It is possible for the implementation of an operation to throw Ice run-time exceptions (as well as user exceptions). For example, if a client holds a proxy to an object that no longer exists in the server, your server application code is required to throw an `ObjectNotExistException`. If you do throw run-time exceptions from your application code, you should take care to throw a run-time exception only if appropriate, that is, do not use run-time exceptions to indicate something that really should be a user exception. Doing so can be very confusing to the client: if the application "hijacks" some run-time exceptions for its own purposes, the client can no longer decide whether the exception was thrown by the Ice run time or by the server application code. This can make debugging very difficult.

4.10.5 Interface Semantics and Proxies

Building on the `Clock` example, we can create definitions for a world-time server:

```
exception GenericError {
    string reason;
};

struct TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

exception BadTimeVal extends GenericError {};

interface Clock {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time) throws BadTimeVal;
};

dictionary<string, Clock*> TimeMap; // Time zone name to clock map
```

```
exception BadZoneName extends GenericError {};

interface WorldTime {
    idempotent void addZone(string zoneName, Clock* zoneClock);
    void removeZone(string zoneName) throws BadZoneName;
    idempotent Clock* findZone(string zoneName)
                                    throws BadZoneName;
    idempotent TimeMap listZones();
    idempotent void setZones(TimeMap zones);
};
```

The `WorldTime` interface acts as a collection manager for clocks, one for each time zone. In other words, the `WorldTime` interface manages a collection of pairs. The first member of each pair is a time zone name; the second member of the pair is the clock that provides the time for that zone. The interface contains operations that permit you to add or remove a clock from the map (`addZone` and `remove-Zone`), to search for a particular time zone by name (`findZone`), and to read or write the entire map (`listZones` and `setZones`).

The `WorldTime` example illustrates an important Slice concept: note that `addZone` accepts a parameter of type `Clock*` and `findZone` returns a parameter of type `Clock*`. In other words, interfaces are types in their own right and can be passed as parameters. The `*` operator is known as the *proxy operator*. Its left-hand argument must be an interface (or class—see Section 4.11) and its return type is a proxy. A proxy is like a pointer that can denote an object. The semantics of proxies are very much like those of C++ class instance pointers:

- A proxy can be null (see page 126).
- A proxy can dangle (point at an object that is no longer there)
- Operations dispatched via a proxy use late binding: if the actual run-time type of the object denoted by the proxy is more derived than the proxy's type, the implementation of the most-derived interface will be invoked.

When a client passes a `Clock` proxy to the `addZone` operation, the proxy denotes an actual `Clock` object in a server. The `Clock` *Ice object* denoted by that proxy may be implemented in the same server process as the `WorldTime` interface, or in a different server process. Where the `Clock` object is physically implemented matters neither to the client nor to the server implementing the `WorldTime` interface; if either invokes an operation on a particular clock, such as `getTime`, an RPC call is sent to whatever server implements that particular clock. In other words, a proxy acts as a local "ambassador" for the remote object; invoking an operation on the proxy forwards the invocation to the actual object implementation. If the

object implementation is in a different address space, this results in a remote procedure call; if the object implementation is collocated in the same address space, the Ice run time uses an ordinary local function call from the proxy to the object implementation.

Note that proxies also act very much like pointers in their sharing semantics: if two clients have a proxy to the same object, a state change made by one client (such as setting the time) will be visible to the other client.

Proxies are strongly typed (at least for statically typed languages, such as C++ and Java). This means that you cannot pass something other than a `Clock` proxy to the `addZone` operation; attempts to do so are rejected at compile time.

4.10.6 Interface Inheritance

Interfaces support inheritance. For example, we could extend our world-time server to support the concept of an alarm clock:

```
interface AlarmClock extends Clock {
    idempotent TimeOfDay getAlarmTime();
    idempotent void      setAlarmTime(TimeOfDay alarmTime)
                                      throws BadTimeVal;
};
```

The semantics of this are the same as for C++ or Java: `AlarmClock` is a subtype of `Clock` and an `AlarmClock` proxy can be substituted wherever a `Clock` proxy is expected. Obviously, an `AlarmClock` supports the same `getTime` and `setTime` operations as a `Clock` but also supports the `getAlarmTime` and `setAlarmTime` operations.

Multiple interface inheritance is also possible. For example, we can construct a radio alarm clock as follows:

```
interface Radio {
    void setFrequency(long hertz) throws GenericError;
    void setVolume(long dB) throws GenericError;
};

enum AlarmMode { RadioAlarm, BeepAlarm };

interface RadioClock extends Radio, AlarmClock {
    void      setMode(AlarmMode mode);
    AlarmMode getMode();
};
```

`RadioClock` extends both `Radio` and `AlarmClock` and can therefore be passed where a `Radio`, an `AlarmClock`, or a `Clock` is expected. The inheritance diagram for this definition looks as follows:

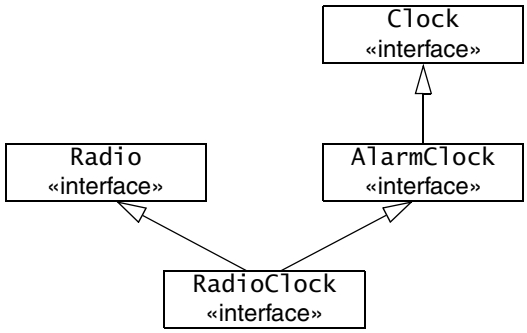


Figure 4.5. Inheritance diagram for `RadioClock`.

Interfaces that inherit from more than one base interface may share a common base interface. For example, the following definition is legal:

```
interface B { /* ... */ };
interface I1 extends B { /* ... */ };
interface I2 extends B { /* ... */ };
interface D extends I1, I2 { /* ... */ };
```

This definition results in the familiar diamond shape:

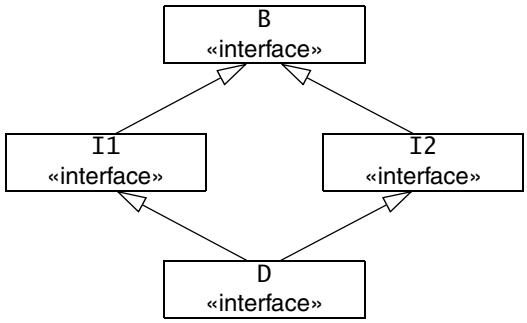


Figure 4.6. Diamond-shaped inheritance.

Interface Inheritance Limitations

If an interface uses multiple inheritance, it must not inherit the same operation name from more than one base interface. For example, the following definition is illegal:

```
interface Clock {
    void set(TimeOfDay time);                   // set time
};

interface Radio {
    void set(long hertz);                       // set frequency
};

interface RadioClock extends Radio, Clock {     // Illegal!
    // ...
};
```

This definition is illegal because `RadioClock` inherits two `set` operations, `Radio::set` and `Clock::set`. The Slice compiler makes this illegal because (unlike C++) many programming languages do not have a built-in facility for disambiguating the different operations. In Slice, the simple rule is that all inherited operations must have unique names. (In practice, this is rarely a problem because inheritance is rarely added to an interface hierarchy "after the fact". To avoid accidental clashes, we suggest that you use descriptive operation names, such as `setTime` and `setFrequency`. This makes accidental name clashes less likely.)

Implicit Inheritance from Object

All Slice interfaces are ultimately derived from `Object`. For example, the inheritance hierarchy from Figure 4.5 would be shown more correctly as in Figure 4.7.

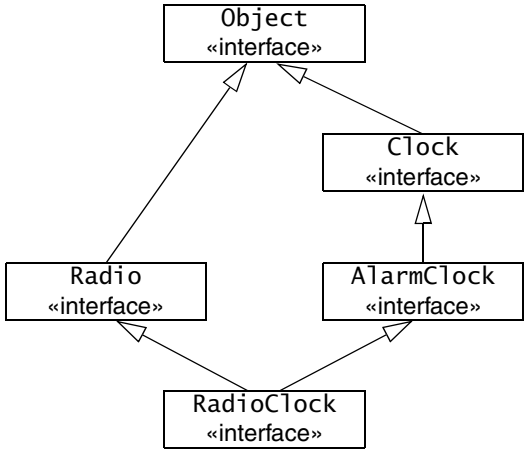


Figure 4.7. Implicit inheritance from `Object`.

Because all interfaces have a common base interface, we can pass any type of interface as that type. For example:

```
interface ProxyStore {
    idempotent  void    putProxy(string name, Object* o);
    idempotent Object* getProxy(string name);
};
```

`Object` is a Slice keyword (note the capitalization) that denotes the root type of the inheritance hierarchy. The `ProxyStore` interface is a generic proxy storage facility: the client can call `putProxy` to add a proxy of any type under a given name and later retrieve that proxy again by calling `getProxy` and supplying that name. The ability to generically store proxies in this fashion allows us to build general-purpose facilities, such as a naming service that can store proxies and deliver them to clients. Such a service, in turn, allows us to avoid hard-coding proxy details into clients and servers (see Chapter 38).

Inheritance from type `Object` is always implicit. For example, the following Slice definition is illegal:

```
interface MyInterface extends Object { /* ... */ }; // Error!
```

It is understood that all interfaces inherit from type `Object`; you are not allowed to restate that.

Type `Object` is mapped to an abstract type by the various language mappings, so you cannot instantiate an Ice object of that type.

Null Proxies

Looking at the `ProxyStore` interface once more, we notice that `getProxy` does not have an exception specification. The question then is what should happen if a client calls `getProxy` with a name under which no proxy is stored? Obviously, we could add an exception to indicate this condition to `getProxy`. However, another option is to return a *null proxy*. Ice has the built-in notion of a null proxy, which is a proxy that "points nowhere". When such a proxy is returned to the client, the client can test the value of the returned proxy to check whether it is null or denotes a valid object.

A more interesting question is: "which approach is more appropriate, throwing an exception or returning a null proxy?" The answer depends on the expected usage pattern of an interface. For example, if, in normal operation, you do not expect clients to call `getProxy` with a non-existent name, it is better to throw an exception. (This is probably the case for our `ProxyStore` interface: the fact that there is no `list` operation makes it clear that clients are expected to know which names are in use.)

On the other hand, if you expect that clients will occasionally try to look up something that is not there, it is better to return a null proxy. The reason is that throwing an exception breaks the normal flow of control in the client and requires special handling code. This means that you should throw exceptions only in exceptional circumstances. For example, throwing an exception if a database lookup returns an empty result set is wrong; it is expected and normal that a result set is occasionally empty.

It is worth paying attention to such design issues: well-designed interfaces that get these details right are easier to use and easier to understand. Not only do such interfaces make life easier for client developers, they also make it less likely that latent bugs cause problems later.

Self-Referential Interfaces

Proxies have pointer semantics, so we can define self-referential interfaces. For example:

```
interface Link {
    idempotent SomeType getValue();
    idempotent Link*    next();
};
```

The `Link` interface contains a `next` operation that returns a proxy to a `Link` interface. Obviously, this can be used to create a chain of interfaces; the final link in the chain returns a null proxy from its `next` operation.

Empty Interfaces

The following Slice definition is legal:

```
interface Empty {};
```

The Slice compiler will compile this definition without complaint. An interesting question is: "why would I need an empty interface?" In most cases, empty interfaces are an indication of design errors. Here is one example:

```
interface ThingBase {};

interface Thing1 extends ThingBase {
    // Operations here...
};

interface Thing2 extends ThingBase {
    // Operations here...
};
```

Looking at this definition, we can make two observations:

- `Thing1` and `Thing2` have a common base and are therefore related.
- Whatever is common to `Thing1` and `Thing2` can be found in interface `ThingBase`.

Of course, looking at `ThingBase`, we find that `Thing1` and `Thing2` do not share any operations at all because `ThingBase` is empty. Given that we are using an object-oriented paradigm, this is definitely strange: in the object-oriented model, the *only* way to communicate with an object is to send a message to the object. But, to send a message, we need an operation. Given that `ThingBase` has no operations, we cannot send a message to it, and it follows that `Thing1` and `Thing2` are *not* related because they have no common operations. But of course, seeing that `Thing1` and `Thing2` have a common base, we conclude that they *are* related, otherwise the common base would not exist. At this point, most programmers begin to scratch their head and wonder what is going on here.

One common use of the above design is a desire to treat `Thing1` and `Thing2` polymorphically. For example, we might continue the previous definition as follows:

```
interface ThingUser {
    void putThing(ThingBase* thing);
};
```

Now the purpose of having the common base becomes clear: we want to be able to pass both `Thing1` and `Thing2` proxies to `putThing`. Does this justify the empty base interface? To answer this question, we need to think about what happens in the implementation of `putThing`. Obviously, `putThing` cannot possibly invoke an operation on a `ThingBase` because there are no operations. This means that `putThing` can do one of two things:

1. `putThing` can simply remember the value of thing.
2. `putThing` can try to down-cast to either `Thing1` or `Thing2` and then invoke an operation. The pseudo-code for the implementation of `putThing` would look something like this:

   ```
   void putThing(ThingBase thing)
   {
       if (is_a(Thing1, thing)) {
           // Do something with Thing1...
       } else if (is_a(Thing2, thing)) {
           // Do something with Thing2...
       } else {
           // Might be a ThingBase?
           // ...
       }
   }
   ```

   The implementation tries to down-cast its argument to each possible type in turn until it has found the actual run-time type of the argument. Of course, any object-oriented text book worth its price will tell you that this is an abuse of inheritance and leads to maintenance problems.

If you find yourself writing operations such as `putThing` that rely on artificial base interfaces, ask yourself whether you really need to do things this way. For example, a more appropriate design might be:

```
interface Thing1 {
    // Operations here...
};
```

```
interface Thing2 {
    // Operations here...
};

interface ThingUser {
    void putThing1(Thing1* thing);
    void putThing2(Thing2* thing);
};
```

With this design, `Thing1` and `Thing2` are not related, and `ThingUser` offers a separate operation for each type of proxy. The implementation of these operations does not need to use any down-casts, and all is well in our object-oriented world.

Another common use of empty base interfaces is the following:

```
interface PersistentObject {};

interface Thing1 extends PersistentObject {
    // Operations here...
};

interface Thing2 extends PersistentObject {
    // Operations here...
};
```

Clearly, the intent of this design is to place persistence functionality into the `PersistentObject` base *implementation* and require objects that want to have persistent state to inherit from `PersistentObject`. On the face of things, this is reasonable: after all, using inheritance in this way is a well-established design pattern, so what can possibly be wrong with it? As it turns out, there are a number of things that are wrong with this design:

- The above inheritance hierarchy is used to add *behavior* to `Thing1` and `Thing2`. However, in a strict OO model, behavior can be invoked only by sending messages. But, because `PersistentObject` has no operations, no messages can be sent.

  This raises the question of how the implementation of `PersistentObject` actually goes about doing its job; presumably, it knows something about the implementation (that is, the internal state) of `Thing1` and `Thing2`, so it can write that state into a database. But, if so, `PersistentObject`, `Thing1`, and `Thing2` can no longer be implemented in different address spaces because, in

that case, `PersistentObject` can no longer get at the state of `Thing1` and `Thing2`.

Alternatively, `Thing1` and `Thing2` use some functionality provided by `PersistentObject` in order to make their internal state persistent. But `PersistentObject` does not have any operations, so how would `Thing1` and `Thing2` actually go about achieving this? Again, the only way that can work is if `PersistentObject`, `Thing1`, and `Thing2` are implemented in a single address space and share implementation state behind the scenes, meaning that they cannot be implemented in different address spaces.

- The above inheritance hierarchy splits the world into two halves, one containing persistent objects and one containing non-persistent ones. This has far-reaching ramifications:
  - Suppose you have an existing application with already implemented, non-persistent objects. Requirements change over time and you find that you now would like to make some of your objects persistent. With the above design, you cannot do this unless you change the type of your objects because they now must inherit from `PersistentObject`. Of course, this is extremely bad news: not only do you have to change the implementation of your objects in the server, you also need to locate and update all the clients that are currently using your objects because they suddenly have a completely new type. What is worse, there is no way to keep things backward compatible: either all clients change with the server, or none of them do. It is impossible for some clients to remain "unupgraded".
  - The design does not scale to multiple features. Imagine that we have a number of additional behaviors that objects can inherit, such as serialization, fault-tolerance, persistence, and the ability to be searched by a search engine. We quickly end up in a mess of multiple inheritance. What is worse, each possible combination of features creates a completely separate type hierarchy. This means that you can no longer write operations that generically operate on a number of object types. For example, you cannot pass a persistent object to something that expects a non-persistent object, *even if the receiver of the object does not care about the persistence aspects of the object*. This quickly leads to fragmented and hard-to-maintain type systems. Before long, you will either find yourself rewriting your application or end up with something that is both difficult to use and difficult to maintain.

The foregoing discussion will hopefully serve as a warning: Slice is an *interface* definition language that has nothing to do with *implementation* (but empty inter-

faces almost always indicate that implementation state is shared via mechanisms other than defined interfaces). If you find yourself writing an empty interface definition, at least step back and think about the problem at hand; there may be a more appropriate design that expresses your intent more cleanly. If you do decide to go ahead with an empty interface regardless, be aware that, almost certainly, you will lose the ability to later change the distribution of the object model over physical server processes because you cannot place an address space boundary between interfaces that share hidden state.

Interface Versus Implementation Inheritance

Keep in mind that Slice interface inheritance applies only to *interfaces*. In particular, if two interfaces are in an inheritance relationship, this in no way implies that the implementations of those interfaces must also inherit from each other. You can choose to use implementation inheritance when you implement your interfaces, but you can also make the implementations independent of each other. (To C++ programmers, this often comes as a surprise because C++ uses implementation inheritance by default, and interface inheritance requires extra effort to implement.)

In summary, Slice inheritance simply establishes type compatibility. It says nothing about how interfaces are implemented and, therefore, keeps implementation choices open to whatever is most appropriate for your application.

4.11 Classes

In addition to interfaces, Slice permits the definition of classes. Classes are like interfaces in that they can have operations and are like structures in that they can have data members. This leads to hybrid objects that can be treated as interfaces and passed by reference, or can be treated as values and passed by value. Classes provide much architectural flexibility. For example, classes allow behavior to be implemented on the client side, whereas interfaces allow behavior to be implemented only on the server side.

Classes support inheritance and are therefore polymorphic: at run time, you can pass a class instance to an operation as long as the actual class type is derived from the formal parameter type in the operation's signature. This also permits classes to be used as type-safe unions, similarly to Pascal's discriminated variant records.

4.11.1 Simple Classes

A Slice class definition is similar to a structure definition, but uses the `class` keyword. For example:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};
```

Apart from the keyword `class`, this definition is identical to the structure definition we saw on page 99. You can use a Slice class wherever you can use a Slice structure (but, as we will see shortly, for performance reasons, you should not use a class where a structure is sufficient). Unlike structures, classes can be empty:

```
class EmptyClass {};    // OK
struct EmptyStruct {};  // Error
```

Much the same design considerations as for empty interfaces (see page 127) apply to empty classes: you should at least stop and rethink your approach before committing yourself to an empty class.

You can specify a default value for a class data member that has one of the following types:

- An integral type (`bool`, `byte`, `short`, `int`, `long`, or an enumerated type)
- `float` or `double`
- `string`

For example:

```
class Location {
    string name;
    Point pt;
    bool display = true;
    string source = "GPS";
};
```

The legal syntax for literal values is the same as for Slice constants (see Section 4.9.5). The language mapping guarantees that data members are initialized to their declared default values using a language-specific mechanism.

4.11.2 Class Inheritance

Unlike structures, classes support inheritance. For example:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

class DateTime extends TimeOfDay {
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
};
```

This example illustrates one major reason for using a class: a class can be extended by inheritance, whereas a structure is not extensible. The previous example defines `DateTime` to extend the `TimeOfDay` class with a date.[8]

Classes only support single inheritance. The following is illegal:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

class Date {
    short day;
    short month;
    short year;
};

class DateTime extends TimeOfDay, Date {   // Error!
    // ...
};
```

A derived class also cannot redefine a data member of its base class:

8. If you are puzzled by the comment about the year 1753, search the Web for “1752 date change”. The intricacies of calendars for various countries prior to that year can keep you occupied for months…

```
class Base {
    int integer;
};

class Derived extends Base {
    int integer;                // Error, integer redefined
};
```

4.11.3 Class Inheritance Semantics

Classes use the same pass-by-value semantics as structures. If you pass a class instance to an operation, the class and all its members are passed. The usual type compatibility rules apply: you can pass a derived instance where a base instance is expected. If the receiver has static type knowledge of the actual derived run-time type, it receives the derived instance; otherwise, if the receiver does not have static type knowledge of the derived type, the instance is sliced to the base type. For an example, suppose we have the following definitions:

```
// In file Clock.ice:

class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};


// In file DateTime.ice:

#include <Clock.ice>

class DateTime extends TimeOfDay {
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
};
```

Because `DateTime` is a sub-class of `TimeOfDay`, the server can return a `DateTime` instance from `getTime`, and the client can pass a `DateTime` instance to `setTime`.

In this case, if both client and server are linked to include the code generated for both `Clock.ice` and `DateTime.ice`, they each receive the actual derived `DateTime` instance, that is, the actual run-time type of the instance is preserved.

Contrast this with the case where the server is linked to include the code generated for both `Clock.ice` and `DateTime.ice`, but the client is linked only with the code generated for `Clock.ice`. In other words, the server understands the type `DateTime` and can return a `DateTime` instance from `getTime`, but the client only understands `TimeOfDay`. In this case, the derived `DateTime` instance returned by the server is sliced to its `TimeOfDay` base type in the client. (The information in the derived part of the instance is simply lost to the client.)

Class hierarchies are useful if you need polymorphic *values* (instead of polymorphic *interfaces*). For example:

```
class Shape {
    // Definitions for shapes, such as size, center, etc.
};

class Circle extends Shape {
    // Definitions for circles, such as radius...
};

class Rectangle extends Shape {
    // Definitions for rectangles, such as width and length...
};

sequence<Shape> ShapeSeq;

interface ShapeProcessor {
    void processShapes(ShapeSeq ss);
};
```

Note the definition of `ShapeSeq` and its use as a parameter to the `processShapes` operation: the class hierarchy allows us to pass a polymorphic sequence of shapes (instead of having to define a separate operation for each type of shape).

The receiver of a `ShapeSeq` can iterate over the elements of the sequence and down-cast each element to its actual run-time type. (The receiver can also ask each element for its type ID to determine its type—see Section 6.14.1 and Section 10.11.2.)

4.11.4 Classes as Unions

Slice does not offer a dedicated union construct because it is redundant. By deriving classes from a common base class, you can create the same effect as with a union:

```
interface ShapeShifter {
    Shape translate(Shape s, long xDistance, long yDistance);
};
```

The parameter `s` of the `translate` operation can be viewed as a union of two members: a `Circle` and a `Rectangle`. The receiver of a `Shape` instance can use the type ID (see Section 4.13) of the instance to decide whether it received a `Circle` or a `Rectangle`. Alternatively, if you want something more along the lines of a conventional discriminated union, you can use the following approach:

```
class UnionDiscriminator {
    int d;
};

class Member1 extends UnionDiscriminator {
    // d == 1
    string s;
    float f;
};

class Member2 extends UnionDiscriminator {
    // d == 2
    byte b;
    int i;
};
```

With this approach, the `UnionDiscriminator` class provides a discriminator value. The "members" of the union are the classes that are derived from `UnionDiscriminator`. For each derived class, the discriminator takes on a distinct value. The receiver of such a union uses the discriminator value in a `switch` statement to select the active union member.

4.11.5 Self-Referential Classes

Classes can be self-referential. For example:

```
class Link {
    SomeType value;
    Link next;
};
```

This looks very similar to the self-referential interface example on page 127, but the semantics are very different. Note that `value` and `next` are data members, not operations, and that the type of `next` is `Link` (*not* `Link*`). As you would expect, this forms the same linked list arrangement as the `Link` interface on page 127: each instance of a `Link` class contains a `next` member that points at the next link in the chain; the final link's `next` member contains a null value. So, what looks like a class including itself really expresses pointer semantics: the `next` data member contains a pointer to the next link in the chain.

You may be wondering at this point what the difference is then between the `Link` interface on page 127 and the `Link` class on page 136. The difference is that classes have *value* semantics, whereas proxies have *reference* semantics. To illustrate this, consider the `Link` *interface* from page 127 once more:

```
interface Link {
    idempotent SomeType getValue();
    idempotent Link*    next();
};
```

Here, `getValue` and `next` are both operations and the return value of `next` is `Link*`, that is, next returns a *proxy*. A proxy has *reference* semantics, that is, it denotes an object somewhere. If you invoke the `getValue` operation on a `Link` proxy, a message is sent to the (possibly remote) servant for that proxy. In other words, for proxies, the object stays put in its server process and we access the state of the object via remote procedure calls. Compare this with the definition of our `Link` *class*:

```
class Link {
    SomeType value;
    Link next;
};
```

Here, `value` and `next` are data members and the type of next is `Link`, which has *value* semantics. In particular, while `next` looks and feels like a pointer, *it cannot denote an instance in a different address space*. This means that if we have a chain of `Link` instances, all of the instances are in our local address space and, when we read or write a value data member, we are performing local address space operations. This means that an operation that returns a `Link` instance, such as `getHead`,

does not just return the head of the chain, *but the entire chain*, as shown in Figure 4.8.

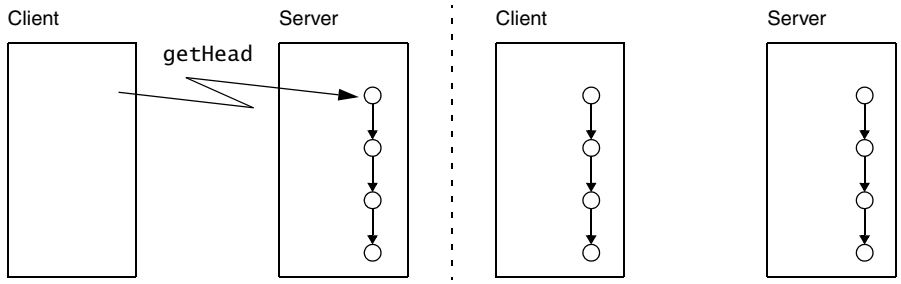


Figure 4.8. Class version of `Link` before and after calling `getHead`.

On the other hand, for the interface version of `Link`, we do not know where all the links are physically implemented. For example, a chain of four links could have each object instance in its own physical server process; those server processes could be each in a different continent. If you have a proxy to the head of this four-link chain and traverse the chain by invoking the `next` operation on each link, you will be sending four remote procedure calls, one to each object

Self-referential classes are particularly useful to model graphs. For example, we can create a simple expression tree along the following lines:

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {};

class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
    Node operand2;
};

class Operand extends Node {
    long val;
};
```

The expression tree consists of leaf nodes of type `Operand`, and interior nodes of type `UnaryOperator` and `BinaryOperator`, with one or two descendants, respectively. All three of these classes are derived from a common base class `Node`. Note that `Node` is an empty class. This is one of the few cases where an empty base class is justified. (See the discussion on page 127; once we add operations to this class hierarchy (see Section 4.11.7), the base class is no longer empty.)

If we write an operation that, for example, accepts a `Node` parameter, passing that parameter results in transmission of the entire tree to the server:

```
interface Evaluator {
    long eval(Node expression); // Send entire tree for evaluation
};
```

Self-referential classes are not limited to acyclic graphs; the Ice run time permits loops: it ensures that no resources are leaked and that infinite loops are avoided during marshaling.

4.11.6 Classes Versus Structures

One obvious question to ask is: why does Ice provide structures as well as classes, when classes obviously can be used to model structures? The answer has to do with the cost of implementation: classes provide a number of features that are absent for structures:

- Classes support inheritance.
- Classes can be self-referential.
- Classes can have operations (see Section 4.11.7).
- Classes can implement interfaces (see Section 4.11.9).

Obviously, an implementation cost is associated with the additional features of classes, both in terms of the size of the generated code and the amount of memory and CPU cycles consumed at run time. On the other hand, structures are simple collections of values ("plain old structs") and are implemented using very efficient mechanisms. This means that, if you use structures, you can expect better performance and smaller memory footprint than if you would use classes (especially for languages with direct support for "plain old structures", such as C++ and C#). Use a class only if you need at least one of its more powerful features.

4.11.7 Classes with Operations

Classes, in addition to data members, can have operations. The syntax for operation definitions in classes is identical to the syntax for operations in interfaces. For example, we can modify the expression tree from Section 4.11.5 as follows:

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {
    idempotent long eval();
};

class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
    Node operand2;
};

class Operand {
    long val;
};
```

The only change compared to the version in Section 4.11.5 is that the `Node` class now has an `eval` operation. The semantics of this are as for a virtual member function in C++: each derived class inherits the operation from its base class and can choose to override the operation's definition. For our expression tree, the `Operand` class provides an implementation that simply returns the value of its `val` member, and the `UnaryOperator` and `BinaryOperator` classes provide implementations that compute the value of their respective subtrees. If we call `eval` on the root node of an expression tree, it returns the value of that tree, regardless of whether we have a complex expression or a tree that consists of only a single `Operand` node.

Operations on classes are normally executed in the caller's address space, that is, operations on classes are *local* operations that do not result in a remote procedure call.[9] Of course, this immediately raises an interesting question: what happens if a client receives a class instance with operations from a server, but client and server are implemented in different languages? Classes with operations

require the receiver to supply a factory for instances of the class. The Ice run time only marshals the data members of the class. If a class has operations, the receiver of the class must provide a class factory that can instantiate the class in the receiver's address space, and the receiver is responsible for providing an implementation of the class's operations.

Therefore, if you use classes with operations, it is understood that client and server each have access to an implementation of the class's operations. No code is shipped over the wire (which, in an environment of heterogeneous nodes using different operating systems and languages is infeasible).

4.11.8 Architectural Implications of Classes

Classes have a number of architectural implications that are worth exploring in some detail.

Classes without Operations

Classes that do not use inheritance and only have data members (whether self-referential or not) pose no architectural problems: they simply are values that are marshaled like any other value, such as a sequence, structure, or dictionary. Classes using derivation also pose no problems: if the receiver of a derived instance has knowledge of the derived type, it simply receives the derived type; otherwise, the instance is sliced to the most-derived type that is understood by the receiver. This makes class inheritance useful as a system is extended over time: you can create derived class without having to upgrade all parts of the system at once.

Classes with Operations

Classes with operations require additional thought. Here is an example: suppose that you are creating an Ice application. Also assume that the Slice definitions use quite a few classes with operations. You sell your clients and servers (both written in Java) and end up with thousands of deployed systems.

As time passes and requirements change, you notice a demand for clients written in C++. For commercial reasons, you would like to leave the development

9. It is possible to invoke an operation on a remote class instance—see the relevant language mapping chapter for details.

of C++ clients to customers or a third party but, at this point, you discover a glitch: your application has lots of classes with operations along the following lines:

```
class ComplexThingForExpertsOnly {
    // Lots of arcane data members here...
    MysteriousThing mysteriousOperation(/* parameters */);
    ArcaneThing arcaneOperation(/* parameters */);
    ComplexThing complexOperation(/* parameters */);
    // etc...
};
```

It does not matter what exactly these operations do. (Presumably, you decided to off-load some of the processing for your application onto the client side for performance reasons.) Now that you would like other developers to write C++ clients, it turns out that your application will work only if these developers provide implementations of all the client-side operations and, moreover, if the semantics of these operations exactly match the semantics of your Java implementations. Depending on what these operations do, providing exact semantic equivalents in a different language may not be trivial, so you decide to supply the C++ implementations yourself. But now, you discover another problem: the C++ clients need to be supported for a variety of operating systems that use a variety of different C++ compilers. Suddenly, your task has become quite daunting: you really need to supply implementations for all the combinations of operating systems and compiler versions that are used by clients. Given the different state of compliance with the ISO C++ standard of the various compilers, and the idiosyncrasies of different operating systems, you may find yourself facing a development task that is much larger than anticipated. And, of course, the same scenario will arise again should you need client implementations in yet another language.

The moral of this story is not that classes with operations should be avoided; they can provide significant performance gains and are not necessarily bad. But, keep in mind that, once you use classes with operations, you are, in effect, using client-side native code and, therefore, you can no longer enjoy the implementation transparencies that are provided by interfaces. This means that classes with operations should be used only if you can tightly control the deployment environment of clients. If not, you are better off using interfaces and classes without operations. That way, all the processing stays on the server and the contract between client and server is provided solely by the Slice definitions, not by the semantics of the additional client-side code that is required for classes with operations.

Classes for Persistence

Ice also provides a built-in persistence mechanism that allows you to store the state of a class in a database with very little implementation effort. To get access to these persistence features, you must define a Slice class whose members store the state of the class. We discuss the persistence features of Slice in detail in Chapter 39.

4.11.9 Classes Implementing Interfaces

A Slice class can also be used as a servant in a server, that is, an instance of a class can be used to provide the behavior for an interface, for example:

```
interface Time {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};

class Clock implements Time {
    TimeOfDay time;
};
```

The `implements` keyword indicates that the class `Clock` provides an *implementation* of the `Time` interface. The class can provide data members and operations of its own; in the preceding example, the `Clock` class stores the current time that is accessed via the `Time` interface. A class can implement several interfaces, for example:

```
interface Time {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};

interface Radio {
    idempotent void setFrequency(long hertz);
    idempotent void setVolume(long dB);
};

class RadioClock implements Time, Radio {
    TimeOfDay time;
    long hertz;
};
```

The class `RadioClock` implements both `Time` and `Radio` interfaces.

A class, in addition to implementing an interface, can also extend another class:

```
interface Time {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};

class Clock implements Time {
    TimeOfDay time;
};

interface AlarmClock extends Time {
    idempotent TimeOfDay getAlarmTime();
    idempotent void setAlarmTime(TimeOfDay alarmTime);
};

interface Radio {
    idempotent void setFrequency(long hertz);
    idempotent void setVolume(long dB);
};

class RadioAlarmClock extends Clock
                      implements AlarmClock, Radio {
    TimeOfDay alarmTime;
    long hertz;
};
```

These definitions result in the inheritance graph shown in Figure 4.9:

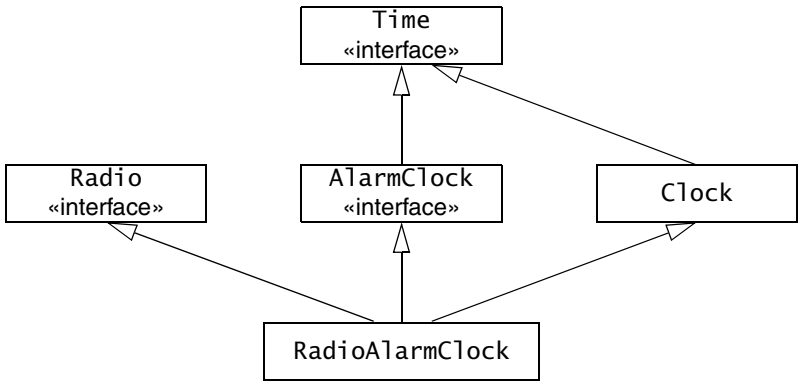


Figure 4.9. A Class using implementation and interface inheritance.

For this definition, `Radio` and `AlarmClock` are abstract interfaces, and `Clock` and `RadioAlarmClock` are concrete classes. As for Java, a class can implement multiple interfaces, but can extend at most one class.

4.11.10 Class Inheritance Limitations

As for interface inheritance, a class cannot redefine an operation or data member that it inherits from a base interface or class. For example:

```
interface BaseInterface {
    void op();
};

class BaseClass {
    int member;
};

class DerivedClass extends BaseClass implements BaseInterface {
    void someOperation();       // OK
    int op();                   // Error!
    int  someMember;            // OK
    long member;                // Error!
};
```

4.11.11 Pass-by-Value Versus Pass-by-Reference

As we saw in Section 4.11.5, classes naturally support pass-by-value semantics: passing a class transmits the data members of the class to the receiver. Any changes made to these data members by the receiver affect only the receiver's copy of the class; the data members of the sender's class are not affected by the changes made by the receiver.

In addition to passing a class by value, you can pass a class by reference. For example:

```
class TimeOfDay {
    short hour;
    short minute;
    short second;
    string format();
};
```

```
interface Example {
     TimeOfDay* get();  // Note: returns a proxy!
};
```

Note that the `get` operation returns a *proxy* to a `TimeOfDay` class and not a `TimeOfDay` instance itself. The semantics of this are as follows:

- When the client receives a `TimeOfDay` proxy from the `get` call, it holds a proxy that differs in no way from an ordinary proxy for an interface.
- The client can invoke operations via the proxy, but *cannot* access the data members. This is because proxies do not have the concept of data members, but represent interfaces: even though the `TimeOfDay` class has data members, only its *operations* can be accessed via a the proxy.

The net effect is that, in the preceding example, the server holds an instance of the `TimeOfDay` class. A proxy for that instance was passed to the client. The only thing the client can do with this proxy is to invoke the `format` operation. The implementation of that operation is provided by the server and, when the client invokes `format`, it sends an RPC message to the server just as it does when it invokes an operation on an interface. The implementation of the `format` operation is entirely up to the server. (Presumably, the server will use the data members of the `TimeOfDay` instance it holds to return a string containing the time to the client.)

The preceding example looks somewhat contrived for classes only. However, it makes perfect sense if classes implement interfaces: parts of your application can exchange class instances (and, therefore, state) by value, whereas other parts of the system can treat these instances as remote interfaces. For example:

```
interface Time {
    string format();
    // ...
};

class TimeOfDay implements Time {
    short hour;
    short minute;
    short second;
};

interface I1 {
     TimeOfDay get();           // Pass by value
     void put(TimeOfDay time);  // Pass by value
};
```

```
interface I2 {
    Time* get();                // Pass by reference
};
```

In this example, clients dealing with interface `I1` are aware of the `TimeOfDay` class and pass it by value whereas clients dealing with interface `I2` deal only with the `Time` interface. However, the actual implementation of the `Time` interface in the server uses `TimeOfDay` instances.

Be careful when designing systems that use such mixed pass-by-value and pass-by-reference semantics. Unless you are clear about what parts of the system deal with the interface (pass by reference) aspects and the class (pass by value) aspects, you can end up with something that is more confusing than helpful.

A good example of putting this feature to use can be found in Freeze (see Chapter 39), which allows you to add classes to an existing interface to implement persistence.

4.11.12 Passing Interfaces by Value

Consider the following definitions:

```
interface Time {
    idempotent TimeOfDay getTime();
    // ...
};

interface Record {
    void addTimeStamp(Time t); // Note: Time t, not Time* t
    // ...
};
```

Note that `addTimeStamp` accepts a parameter of type `Time`, not of type `Time*`. The question is, what does it mean to pass an interface *by value*? Obviously, at run time, we cannot pass an an actual interface to this operation because interfaces are abstract and cannot be instantiated. Neither can we pass a proxy to a `Time` object to `addTimeStamp` because a proxy cannot be passed where an interface is expected.

However, what we *can* pass to `addTimeStamp` is something that is not abstract and derives from the `Time` interface. For example, at run time, we could pass an instance of our `TimeOfDay` class from the previous section. Because the `TimeOfDay` class derives from the `Time` interface, the class type is compatible with

the formal parameter type `Time` and, at run time, what is sent over the wire to the server is the `TimeOfDay` class instance.

4.12 Forward Declarations

Both interfaces and classes can be forward declared. Forward declarations permit the creation of mutually dependent objects, for example:

```
module Family {
    interface Child;            // Forward declaration

    sequence<Child*> Children;  // OK

    interface Parent {
        Children getChildren(); // OK
    };

    interface Child {           // Definition
        Parent* getMother();
        Parent* getFather();
    };
};
```

Without the forward declaration of `Child`, the definition obviously could not compile because `Child` and `Parent` are mutually dependent interfaces. You can use forward-declared interfaces and classes to define types (such as the `Children` sequence in the previous example). Forward-declared interfaces and classes are also legal as the type of a structure, exception, or class member, as the value type of a dictionary, and as the parameter and return type of an operation. However, you cannot inherit from a forward-declared interface or class until after its definition has been seen by the compiler:

```
interface Base;                         // Forward declaration

interface Derived1 extends Base {};     // Error!

interface Base {};                      // Definition

interface Derived2 extends Base {};     // OK, definition was seen
```

Not inheriting from a forward-declared base interface or class until its definition is seen is necessary because, otherwise, the compiler could not enforce that derived interfaces must not redefine operations that appear in base interfaces.[10]

4.13 Type IDs

Each user-defined Slice type has an internal type identifier, known as its *type ID*. The type ID is simply the fully-qualified name of each type. For example, the type ID of the `Child` interface in the preceding example is `::Family::Children::Child`. All type IDs for user-defined types start with a leading `::`, so the type ID of the `Family` module is `::Family` (not `Family`). In general, a type ID is formed by starting with the global scope (`::`) and forming the fully-qualified name of a type by appending each module name in which the type is nested, and ending with the name of the type itself; the components of the type ID are separated by `::`.

The type ID of a proxy is formed by appending a `*` to the type ID of an interface or class. For example, the type ID of a `Child` proxy is `::Family::Children::Child*`.

The type ID of the Slice `Object` type is `::Ice::Object` and the type ID of an `Object` proxy is `::Ice::Object*`.

The type IDs for the remaining built-in types, such as `int`, `bool`, and so on, are the same as the corresponding keyword. For example, the type ID of `int` is `int`, and the type ID of `string` is `string`.

Type IDs are used internally by the Ice run time as a unique identifier for each type. For example, when an exception is raised, the marshaled form of the exception that is returned to the client is preceded by its Type ID on the wire. The client-side run time first reads the Type ID and, based on that, unmarshals the remainder of the data as appropriate for the type of the exception.

Type IDs are also used by the `ice_isA` operation (see page 150).

10.A multi-pass compiler could be used, but the added complexity is not worth it.

4.14 Operations on `Object`

The `Object` interface has a number of operations. We cannot define type `Object` in Slice because `Object` is a keyword; regardless, here is what (part of) the definition of `Object` would look like if it were legal:

```
sequence<string> StrSeq;

interface Object {                      // "Pseudo" Slice!
    idempotent void   ice_ping();
    idempotent bool   ice_isA(string typeID);
    idempotent string ice_id();
    idempotent StrSeq ice_ids();
    // ...
};
```

Note that, apart from the illegal use of the keyword `Object` as the interface name, the operation names all contain the `ice_` prefix. This prefix is reserved for use by Ice and cannot clash with a user-defined operation. This means that all Slice interfaces can inherit from `Object` without name clashes. There are three built-in operations that are commonly used:

- `ice_ping`

  All interfaces support the `ice_ping` operation. That operation is useful for debugging because it provides a basic reachability test for an object: if the object exists and a message can successfully be dispatched to the object, `ice_ping` simply returns without error. If the object cannot be reached or does not exist, `ice_ping` throws a run-time exception that provides the reason for the failure.

- `ice_isA`

  The `ice_isA` operation accepts a type identifier (such as the identifier returned by `ice_id`) and tests whether the target object supports the specified type, returning `true` if it does. You can use this operation to check whether a target object supports a particular type. For example, referring to Figure 4.7 once more, assume that you are holding a proxy to a target object of type `AlarmClock`. Table 4.2 illustrates the result of calling `ice_isA` on that proxy with

various arguments. (We assume that all type in Figure 4.7 are defined in a module `Times`):

Table 4.2. Calling `ice_isA` on a proxy denoting an object of type `AlarmClock`.

Argument	Result
`::Ice::Object`	`true`
`::Times::Clock`	`true`
`::Times::AlarmClock`	`true`
`::Times::Radio`	`false`
`::Times::RadioClock`	`false`

As expected, `ice_isA` returns true for `::Times::Clock` and `::Times::AlarmClock` and also returns true for `::Ice::Object` (because *all* interfaces support that type). Obviously, an `AlarmClock` supports neither the `Radio` nor the `RadioClock` interfaces, so `ice_isA` returns false for these types.

- `ice_id`

  The `ice_id` operation returns the type ID (see Section 4.13) of the most-derived type of an interface.

- `ice_ids`

  The `ice_ids` operation returns a sequence of type IDs that contains all of the type IDs supported by an interface. For example, for the RadioClock interface in Figure 4.7, `ice_ids` returns a sequence containing the type IDs `::Ice::Object`, `::Times::Clock`, `::Times::AlarmClock`, `::Times::Radio`, and `::Times::RadioClock`.

4.15 Local Types

In order to access certain features of the Ice run time, you must use APIs that are provided by libraries. However, instead of defining an API that is specific to each implementation language, Ice defines its APIs in Slice using the `local` keyword. The advantage of defining APIs in Slice is that a single definition suffices to

define the API for all possible implementation languages. The actual language-specific API is then generated by the Slice compiler for each implementation language. Types that are provided by Ice libraries are defined using the Slice `local` keyword. For example:

```
module Ice {
    local interface ObjectAdapter {
        // ...
    };
};
```

Any Slice definition (not just interfaces) can have a `local` modifier. If the `local` modifier is present, the Slice compiler does not generate marshaling code for the corresponding type. This means that a local type can *never* be accessed remotely because it cannot be transmitted between client and server. (The Slice compiler prevents use of `local` types in non-`local` contexts.)

In addition, local interfaces and local classes do *not* inherit from `Ice::Object`. Instead, local interfaces and classes have their own, completely separate inheritance hierarchy. At the root of this hierarchy is the type `Ice::LocalObject`, as shown in Figure 4.10.

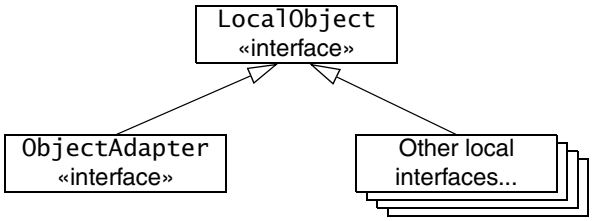


Figure 4.10. Inheritance from `LocalObject`

Because local interfaces form a completely separate inheritance hierarchy, you cannot pass a local interface where a non-local interface is expected and vice-versa.

You rarely need to define local types for your own applications—the `local` keyword exists mainly to allow definition of APIs for the Ice run time. (Because local objects cannot be invoked remotely, there is little point for an application to define local objects; it might as well define ordinary programming-language objects instead.) However, there is one exception to this rule: servant locators must be implemented as local objects (see Section 32.7).

4.16 Names and Scoping

Slice has a number of rules regarding identifiers. You will typically not have to concern yourself with these. However, occasionally, it is good to know how Slice uses naming scopes and resolves identifiers.

4.16.1 Naming Scopes

The following Slice constructs establish a naming scope:

- the global (file) scope
- modules
- interfaces
- classes
- structures
- exceptions
- parameter lists

Within a naming scope, identifiers must be unique, that is, you cannot use the same identifier for different purposes. For example:

```
interface Bad {
    void op(int p, string p);   // Error!
};
```

Because a parameter list forms a naming scope, it is illegal to use the same identifier `p` for different parameters. Similarly, data members, operation names, interface and class names, etc. must be unique within their enclosing scope.

4.16.2 Case Sensitivity

Identifiers that differ only in case are considered identical, so you must use identifiers that differ not only in capitalization within a naming scope. For example:

```
struct Bad {
    int    m;
    string M;   // Error!
};
```

The Slice compiler also enforces consistent capitalization for identifiers. Once you have defined an identifier, you must use the same capitalization for that identifier thereafter. For example, the following is in error:

```
sequence<string> StringSeq;

interface Bad {
    stringSeq op();     // Error!
};
```

Note that identifiers must not differ from a Slice keyword in case only. For example, the following is in error:

```
interface Module {      // Error, "module" is a keyword
    // ...
};
```

4.16.3 Qualified Names

The scope-qualification operator `::` allows you to refer to a type in a non-local scope. For example:

```
module Types {
    sequence<long> LongSeq;
};

module MyApp {
    sequence<Types::LongSeq> NumberTree;
};
```

Here, the qualified name `Types::LongSeq` refers to `LongSeq` defined in module `Types`. The global scope is denoted by a leading `::`, so we could also refer to `LongSeq` as `::Types::LongSeq`.

The scope-qualification operator also allows you to create mutually dependent interfaces that are defined in different modules. The obvious attempt to do this fails:

```
module Parents {
    interface Children::Child;  // Syntax error!
    interface Mother {
        Children::Child* getChild();
    };
    interface Father {
        Children::Child* getChild();
    };
```

```
};

module Children {
    interface Child {
        Parents::Mother* getMother();
        Parents::Father* getFather();
    };
};
```

This fails because it is syntactically illegal to forward-declare an interface in a different module. To make it work, we must use a reopened module:

```
module Children {
    interface Child;                    // Forward declaration
};

module Parents {
    interface Mother {
        Children::Child* getChild();    // OK
    };
    interface Father {
        Children::Child* getChild();    // OK
    };
};

module Children {                       // Reopen module
    interface Child {                   // Define Child
        Parents::Mother* getMother();
        Parents::Father* getFather();
    };
};
```

While this technique works, it is probably of dubious value: mutually dependent interfaces are, by definition, tightly coupled. On the other hand, modules are meant to be used to place related definitions into the same module, and unrelated definitions into different modules. Of course, this begs the question: if the interfaces are so closely related that they depend on each other, why are they defined in different modules? In the interest of clarity, you probably should avoid this construct, even though it is legal.

4.16.4 Names in Nested Scopes

Names defined in an enclosing scope can be redefined in an inner scope. For example, the following is legal:

```
module Outer {
    sequence<string> Seq;

    module Inner {
        sequence<short> Seq;
    };
};
```

Within module `Inner`, the name `Seq` refers to a sequence of `short` values and hides the definition of `Outer::Seq`. You can still refer to the other definition by using explicit scope qualification, for example:

```
module Outer {
    sequence<string> Seq;

    module Inner {
        sequence<short> Seq;

        struct Confusing {
            Seq          a;     // Sequence of short
            ::Outer::Seq b;     // Sequence of string
        };
    };
};
```

Needless to say, you should try to avoid such redefinitions—they make it harder for the reader to follow the meaning of a specification.

Same-named constructs cannot be nested inside each other. For example, a module named `M` cannot (recursively) contain any construct also named `M`. The same is true for interfaces, classes, structures, exceptions, and operations. For example, the following examples are all in error:

```
module M {
    interface M { /* ... */ };  // Error!

    interface I {
        void I();               // Error!
        void op(string op);     // Error!
    };

    struct S {
        long s;                 // Error, even if case differs!
    };

};
```

```
module Outer {
    module Inner {
        interface Outer {       // Error!
            // ...
        };
    };
};
```

The reason for this restriction is that nested types that have the same name are difficult to map into some languages. For example, C++ and Java reserve the name of a class as the name of the constructor, so an interface `I` could not contain an operation named `I` without artificial rules to avoid the name clash.

Similarly, some languages (such as C# prior to version 2.0) do not permit a qualified name to be anchored at the global scope. If a nested module or type is permitted to have the same name as the name of an enclosing module, it can become impossible to generate legal code in some cases.

In the interest of simplicity, Slice simply prohibits the name of a nested module or type to be the same as the name of one of its enclosing modules.

4.16.5 Introduced Identifiers

Within a naming scope, an identifier is introduced at the point of first use; thereafter, within that naming scope, the identifier cannot change meaning. For example:

```
module M {
    sequence<string> Seq;

    interface Bad {
        Seq op1();      // Seq and op1 introduced here
        int Seq();      // Error, Seq has changed meaning
    };
};
```

The declaration of `op1` uses `Seq` as its return type, thereby introducing `Seq` into the scope of interface `Bad`. Thereafter, `Seq` can only be used as a type name that denotes a sequence of strings, so the compiler flags the declaration of the second operation as an error.

Note that fully-qualified identifiers are *not* introduced into the current scope:

```
module M {
    sequence<string> Seq;

    interface Bad {
        ::M::Seq op1(); // Only op1 introduced here
        int Seq();      // OK
    };
};
```

In general, a fully-qualified name (one that is anchored at the global scope and, therefore, begins with a `::` scope resolution operator) does not introduce any name into the current scope. On the other hand, a qualified name that is not anchored at the global scope introduces only the first component of the name:

```
module M {
    sequence<string> Seq;

    interface Bad {
        M::Seq op1();   // M and op1 introduced here, but not Seq
        int Seq();      // OK
    };
};
```

4.16.6 Name Lookup Rules

When searching for the definition of a name that is not anchored at the global scope, the compiler first searches backward in the current scope of a definition of the name. If it can find the name in the current scope, it uses that definition. Otherwise, the compiler successively searches enclosing scopes for the name until it reaches the global scope. Here is an example to illustrate this:

```
module M1 {
    sequence<double> Seq;

    module M2 {
        sequence<string> Seq;   // OK, hides ::M1::Seq

        interface Base {
            Seq op1();          // Returns sequence of string
        };
    };

    module M3 {
        interface Derived extends M2::Base {
            Seq op2();          // Returns sequence of double
```

```
        };

        sequence<bool> Seq;     // OK, hides ::M1::Seq

        interface I {
            Seq op();           // Returns sequence of bool
        };
    };

    interface I {
        Seq op();               // Returns sequence of double
    };
};
```

Note that `M2::Derived::op2` returns a sequence of `double`, even though `M1::Base::op1` returns a sequence of `string`. That is, the meaning of a type in a base interface is irrelevant to determining its meaning in a derived interface—the compiler *always* searches for a definition only in the current scope and enclosing scopes, and *never* takes the meaning of a name from a base interface or class.

4.17 Metadata

Slice has the concept of a *metadata* directive. For example:

```
["java:type:java.util.LinkedList"] sequence<int> IntSeq;
```

A metadata directive can appear as a prefix to any Slice definition. Metadata directives appear in a pair of square brackets and contain one or more string literals separated by commas. For example, the following is a syntactically valid metadata directives containing two strings:

```
["a", "b"] interface Example {};
```

Metadata directives are not part of the Slice language per se: the presence of a metadata directive has no effect on the client–server contract, that is, metadata directives do not change the Slice type system in any way. Instead, metadata directives are targeted at specific back-ends, such as the code generator for a particular language mapping. In the preceding example, the `java:` prefix indicates that the directive is targeted at the Java code generator.

Metadata directives permit you to provide supplementary information that does not change the Slice types being defined, but somehow influences how the compiler will generate code for these definitions. For example, a metadata direc-

tive `java:type:java.util.LinkedList` instructs the Java code generator to map a sequence to a linked list instead of an array (which is the default).

Metadata directives are also used to create proxies and skeletons that support *Asynchronous Method Invocation* (*AMI*) and *Asynchronous Method Dispatch* (*AMD*).

Apart from metadata directives that are attached to a specific definition, there are also global metadata directives. For example:

```
[["java:package:com.acme"]]
```

Note that a global metadata directive is enclosed by double square brackets, whereas a local metadata directive (one that is attached to a specific definition) is enclosed by single square brackets. Global metadata directives are used to pass instructions that affect the entire compilation unit. For example, the preceding metadata directive instructs the Java code generator to generate the contents of the source file into the Java package `com.acme`. Global metadata directives must precede any definitions in a file (but can appear following any `#include` directives).

We discuss specific metadata directives in the relevant chapters to which they apply.

You can find a summary of all metadata directives in Appendix B.

4.18 Serializable Objects

Ice for Java and Ice for .NET allow you to send native Java and CLR objects as operation parameters. The Ice run time automatically serializes and deserializes the objects as part of an invocation. This mechanism allows you to transmit Java and CLR objects that do not have a corresponding Slice definition.

4.18.1 The `serializable` Metadata Directive

To enable serialization, the parameter type must be a byte sequence with appropriate metadata. For example:

```
["java:serializable:SomePackage.JavaClass"]
sequence<byte> JavaObj;

interface JavaExample {
    void sendJavaObj(JavaObj o);
};
```

```
["clr:serializable:SomeNamespace.CLRClass"]
sequence<byte> CLRObj;

interface CLRExample {
    void sendCLRObj(CLRObj o);
};
```

The `java:serializable` metadata indicates that the corresponding byte sequence holds a Java serializable type named `SomePackage.JavaClass`. Your program must provide an implementation of this class; the class must be derived from `java.io.Serializable`. (See Section 10.15 for details of the language mapping.)

Similarly, the `clr:serializable` metadata indicates that the corresponding byte sequences holds a CLR serializable type named `SomeNamespace.CLRClass`. Your program must provide an implementation of this class; the class must be marked with the `Serializable` attribute. (See Section 14.14 for the details of the language mapping.)

4.18.2 Architectural Implications

The `serializable` metadata directive permits you to transmit arbitrary Java and CLR objects across the network without the need to define corresponding Slice classes or structures. This is mainly a convenience feature: you could achieve the same thing by using ordinary Slice byte sequences and explicitly serializing your Java or CLR objects into byte sequences at the sending end, and deserializing them at the receiving end. The `serializable` metadata conveniently takes care of these chores for you and so is simpler to use.

Despite its convenience, you should use this feature with caution because it destroys language transparency. For example, a serialized Java object is useless to a C++ server. All the C++ server can do with such an object is to pass it on to some other process as a byte sequence. (Of course, if that receiving process is a Java process, it can deserialize the byte sequence.)

Further, similar to Slice classes with methods (see Section 4.11.8), a serialized object can be deserialized only if client and server agree on the definition of the serialized class. In Java, this is enforced by the `serialVersionUID` field of each instance; in the CLR, client and server must reference identical assembly versions. This creates much tighter coupling of client and server than exchanging Slice-defined types.

And, of course, if you build a system that relies on, for example, the exchange of serialized Java objects and you later find that you need to add C++ or C# components to the system, these components cannot do anything with the serialized Java objects other than pass them around as a blob of bytes.

So, if you do use these features, be clear that this implies tighter coupling between client and server, and that it creates additional library versioning and distribution issues because all parts of the system must agree on the implementation of the serialized objects.

4.19 Deprecating Slice Definitions

All Slice compilers support a metadata directive that allows you to deprecate a Slice definition. For example:

```
interface Example {
    ["deprecated:someOperation() has been deprecated, \
    use alternativeOperation() instead."]
    void someOperation();

    void alternativeOperation();
};
```

The `["deprecated"]` metadata directive causes the compiler to emit code that generates a warning if you compile application code that uses a deprecated feature. This is useful if you want to remove a feature from a Slice definition but do not want to cause a hard error.

The message that follows the colon is optional; if you omit the message and use `["deprecated"]`, the Slice compilers insert a default message into the generated code.

You can apply the `["deprecated"]` metadata directive to Slice constructs other than operations (for example, a structure or sequence definition).

4.20 Using the Slice Compilers

Ice provides a separate Slice compiler for each language mapping, as shown in Table 4.3.

Table 4.3. The Slice compilers.

Language	Compiler
C++	**slice2cpp**
Java	**slice2java**
C#	**slice2cs**
Python	**slice2py**
Ruby	**slice2rb**
Objective-C	**slice2objc**

The compilers share a similar command-line syntax:

```
<compiler-name> [options] file…
```

Regardless of which compiler you use, a number of command-line options are common to the compilers for any language mapping. (See the appropriate language mapping chapter for options that are specific to a particular language mapping.) The common command-line options are:

- **-h**, **--help**

  Displays a help message.

- **-v**, **--version**

  Displays the compiler version.

- **-D*NAME***

  Defines the preprocessor symbol ***NAME***.

- **-D*NAME=DEF***

  Defines the preprocessor symbol ***NAME*** with the value ***DEF***.

- **-U*NAME***

  Undefines the preprocessor symbol ***NAME***.

- **-I*DIR***

  Add the directory ***DIR*** to the search path for `#include` directives.

- **-E**

  Print the preprocessor output on `stdout`.

- **--output-dir *DIR***

  Place the generated files into directory ***DIR***.

- **-d**, **--debug**

  Print debug information showing the operation of the Slice parser.

- **--ice**

  Permit use of the normally reserved prefix `Ice` for identifiers. Use this option only when compiling the source code for the Ice run time.

- **--underscore**

  Permit use of underscores in Slice identifiers.

The Slice compilers permit you to compile more than a single source file, so you can compile several Slice definitions at once, for example:

```
slice2cpp -I. file1.ice file2.ice file3.ice
```

4.21 Slice Checksums

As distributed applications evolve, developers and system administrators must be careful to ensure that deployed components are using the same client–server contract. Unfortunately, mistakes do happen, and it is not always readily apparent when they do.

To minimize the chances of this situation, the Slice compilers support an option that generates checksums for Slice definitions, thereby enabling two peers to verify that they share an identical client–server contract. The checksum for a Slice definition includes details such as parameter and member names and the order in which operations are defined, but ignores information that is not relevant to the client–server contract, such as metadata, comments, and formatting.

This option causes the Slice compiler to construct a dictionary that maps Slice type identifiers to checksums. A server typically supplies an operation that returns

its checksum dictionary for the client to compare with its local version, at which point the client can take action if it discovers a mismatch.

The dictionary type is defined in the file `Ice/SliceChecksumDict.ice` as follows:

```
module Ice {
    dictionary<string, string> SliceChecksumDict;
};
```

This type can be incorporated into an application's Slice definitions like this:

```
#include <Ice/SliceChecksumDict.ice>

interface MyServer {
    idempotent Ice::SliceChecksumDict getSliceChecksums();
    // ...
};
```

The key of each element in the dictionary is a Slice type ID (see Section 4.13), and the value is the checksum of that type.

For more information on generating and using Slice checksums, see the appropriate language mapping chapter.

4.22 Generating Slice Documentation

If you look at the online Slice reference, you will find reference documentation for all the Slice definitions used by Ice and its services. In the binary distributions of Ice, you will also find HTML documentation that contains the same information. Both the PDF and the HTML documentation are generated from special comments in the Slice definitions by **`slice2html`**, a tool that scans Slice definitions for special comments and generates HTML pages for those comments.

As an example of documentation comments, here is the definition of `Ice::Current`:

```
/**
 *
 * Information about the current method invocation for servers.
 * Each operation on the server has a [Current] as its implicit
 * final parameter. [Current] is mostly used for Ice services.
 * Most applications ignore this parameter.
 *
 **/
```

```
local struct Current {
    /**
     * The object adapter.
     **/
    ObjectAdapter adapter;

    /**
     * Information about the connection over which the current
     * method invocation was received. If the invocation is direct
     * due to collocation optimization, this value is set to null.
     **/
    Connection con;

    /**
     * The Ice object identity.
     **/
    Identity id;

    /**
     * The facet.
     ***/
    string facet;

    /**
     * The operation name.
     **/
    string operation;

    /**
     * The mode of the operation.
     **/
    OperationMode mode;

    /**
     * The request context, as received from the client.
     **/
    Context ctx;

    /**
     * The request id unless oneway (0) or collocated (-1).
     **/
    int requestId;
};
```

If you look at the comments, you will see these reflected in the documentation for `Ice::Current` in the online Slice API Reference.

4.22.1 Documentation Comments

Any comment that starts with `/**` and ends with `**/` is a documentation comment. Such a comment can precede any Slice construct, such as a module, interface, structure, operation, and so on. Within a documentation comment, you can either start each line with a `*`, or you can leave the beginning of the line blank—**`slice2html`** can handle either convention:

```
/**
 *
 * This is a documentation comment for which every line
 * starts with a '*' character.
 **/

/**

 This is a documentation comment without a leading '*'
 for each line. Either style of comment is fine.

 **/
```

The first sentence of the documentation comment for a Slice construct should be a summary sentence. **`slice2html`** generates an index of all Slice constructs; the first sentence of the comments for each Slice construct is ued as a summary in that index.

Hyperlinks

Any Slice identifier enclosed in square brackets is presented as a hyperlink in code font. For example:

```
/**
 * An empty [name] denotes a null object.
 **/
```

This generates a hyperlink for the `name` markup that points at the definition of the corresponding Slice symbol. (The symbol can denote any Slice construct, such as a type, interface, parameter, or structure member.)

Explicit Cross-References

The directive `@see` is recognized by **`slice2html`**. Where it appears, the generated HTML contains a separate section titled "See Also", followed by a list of Slice identifiers. For example:

```
/**
 * The object adapter, which is responsible for receiving requests
 * from endpoints, and for mapping between servants, identities,
 * and proxies.
 *
 * @see Communicator
 * @see ServantLocator
 **/
```

The Slice identifiers are listed in the corresponding "See Also" section as hyperlinks in code font.

Markup for Operations

There are three directives specifically to document Slice operations: `@param`, `@return`, and `@throws`. For example:

```
/**
 * Look for an item with the specified
 * primary and secondary key.
 *
 * @param p The primary search key.
 *
 * @param s The secondary search key.
 *
 * @return The item that matches the specified keys.
 *
 * @throws NotFound Raised if no item matches the specified keys.
 **/

Item findItem(Key p, Key s) throws NotFound;
```

`slice2html` generates separate "Parameters", "Return Value", and "Exceptions" sections for these directives. Parameters are listed in the same order as they appear in the comments. (For clarity, that order should match the order of declaration of parameters for the corresponding operation.)

General HTML Markup

A documentation comment can contain any markup that is permitted by HTML in that place. For example, you can create separate paragraphs with <P> and </P> elements:

```
/**
 * This is a comment for some Slice construct.</p>
 *
 * <p>This comment appears in a separate paragraph.
 **/
```

Note that you must neither begin a documentation comment with a element nor end it with a element because, in the generated HTML, documentation comments are already surrounded by and elements.

There are various other ways to create markup—for example, you can use <table> or <ul> elements. Please see the HTML specification [25] for details.

4.22.2 Using `slice2html`

`slice2html` uses the following syntax:

`slice2html [options] slice_file…`

If you have cross-references that span Slice files, you must compile all of the Slice files with a single invocation of **`slice2html`**.

The command supports the following options:

- **`-h`**, **`--help`**

  Displays a help message.

- **`-v`**, **`--version`**

  Displays the compiler version.

- **`-D`*`NAME`***

  Defines the preprocessor symbol ***`NAME`***.

- **`-D`*`NAME=DEF`***

  Defines the preprocessor symbol ***`NAME`*** with the value ***`DEF`***.

- **`-U`*`NAME`***

  Undefines the preprocessor symbol ***`NAME`***.

- **`-I`*`DIR`***

  Add the directory ***`DIR`*** to the search path for `#include` directives.

- **-E**

  Print the preprocessor output on `stdout`.

- **--output-dir *DIR***

  Place the generated files into the directory ***DIR***. (The default setting is the current directory.)

- **--hdr *FILE***

  Prepend ***FILE*** to each generated HTML file (except for `_sindex.html`). This allows you to replace the HTML header and other preamble information with a custom version, so you can connect style sheets to the generated pages. The specified file must include the `<body>` tag (but need not end with a `<body>` tag).

  FILE is expected to contain the string `TITLE` on a line by itself, starting in column one. slice2html replaces the `TITLE` string with the fully-scoped name of the Slice symbol that is documented on the corresponding page.

- **--ftr *FILE***

  Append ***FILE*** to each generated HTML file (except for `_sindex.html`). This allows you to add, for example, a custom footer to each generated page.

  FILE is must end with a `</body>` tag.

- **--indexhdr *FILE***

  slice2html generates a file `_sindex.html` that contains a table of contents of all Slice symbols that hyperlink to the corresponding page. This option allows you to replace the standard header with a custom header, for example, to attach a JavaScript. The specified file must include the `<body>` tag (but need not end with a `<body>` tag).

  The default value is the setting of **--hdr** (if any).

- **--indexftr *FILE***

  Append ***FILE*** to the generated`_sindex.html` page. This allows you to add, for example, a custom footer to the table of contents, or to invoke a JavaScript.

  FILE is must end with a `</body>` tag.

  The default value is the setting of **--ftr** (if any).

- **--image-dir *DIR***

  With this option, **slice2html** looks in the specified directory for images to use for the generated navigation hyperlinks. (Without this option, text links

are used instead.) Please see the generated HTML for the names of the various image files. (They can easily be found by looking for `img` elements.)

- **`--logo-url` *`URL`***

  Use the specified URL as a hyperlink for the company logo that is added to each page (if **`--image-dir`** is specified). The company logo is expected to be in ***`<image-dir>`*`/logo.gif`**.

- **`--search` *`ACTION`***

  If this option is specified, the generated pages contain a search box that allows you to connect the generated pages to a search engine. On pressing the "Search" button, the specified ***`ACTION`*** is carrid out.

- **`--index` *`NUM`***

  `slice2html` generates sub-indexes for various Slice symbols. This option controls how many entries must be present before a sub-index is generated. For example, if ***`NUM`*** is set to 3, a sub-index will be generated only if there are three or more symbols that appear in that index. The default settings is 1, meaning that a sub-index is always generated. To disable sub-indexes entirely, set ***`NUM`*** to 0.

- **`--summary` *`NUM`***

  If this option is set, summary sentences that exceed ***`NUM`*** characters generate a warning.

- **`-d`**, **`--debug`**

  Print debug information showing the operation of the Slice parser.

- **`--ice`**

  Permit use of the normally reserved prefix `Ice` for identifiers. Use this option if your Slice definitions include Slice files for Ice or its services.

4.23 Summary

Slice is the fundamental mechanism for defining the client–server contract. By defining data types and interfaces in Slice, you create a language-independent API definition that are translated by a compiler into an API specific for a particular programming language.

Slice provides the usual built-in types and allows you to create user-defined types of arbitrary complexity, such as sequences, enumerations, structures,

dictionaries, and classes. Polymorphism is catered for via inheritance of interfaces, classes, and exceptions. In turn, exceptions provide you with facilities that permit sophisticated error reporting and handling. Modules permit you to group related parts of a specification and prevent pollution of the global namespace, and metadata can be used to augment Slice definitions with directives for specific compiler backends.

`slice2html` permits you to integrate Slice documentation with existing documentation tools.

Chapter 5
Slice for a Simple File System

5.1 Chapter Overview

The remainder of this book uses a file system application to illustrate various aspects of Ice. Throughout the book, we progressively improve and modify the application such that it evolves into an application that is realistic and illustrates the architectural and coding aspects of Ice. This allows us to explore the capabilities of the platform to a realistic degree of complexity without overwhelming you with an inordinate amount of detail early on. Section 5.2 outlines the file system functionality, Section 5.3 develops the data types and interfaces that are required for the file system, and Section 5.4 presents the complete Slice definition for the application.

5.2 The File System Application

Our file system application implements a simple hierarchical file system, similar to the file systems we find in Windows or Unix. To keep code examples to manageable size, we ignore many aspects of a real file system, such as ownership, permissions, symbolic links, and a number of other features. However, we build enough functionality to illustrate how you could implement a fully-featured file system, and we pay attention to things such as performance and scalability. In this

way, we can create an application that presents us with real-world complexity without getting buried in large amounts of code.

Our file system consists of directories and files. Directories are containers that can contain either directories or files, meaning that the file system is hierarchical. A dedicated directory is at the root of the file system. Each directory and file has a name. Files and directories with a common parent directory must have different names (but files and directories with different parent directories can have the same name). In other words, directories form a naming scope, and entries with a single directory must have unique names. Directories allow you to list their contents.

For now, we do not have a concept of pathnames, or the creation and destruction of files and directories. Instead, the server provides a fixed number of directories and files. (We will address the creation and destruction of files and directories in Chapter 34.)

Files can be read and written but, for now, reading and writing always replace the entire contents of a file; it is impossible to read or write only parts of a file.

5.3 Slice Definitions for the File System

Given the very simple requirements we just outlined, we can start designing interfaces for the system. Files and directories have something in common: they have a name and both files and directories can be contained in directories. This suggests a design that uses a base type that provides the common functionality, and derived types that provide the functionality specific to directories and files, as shown in Figure 5.1.

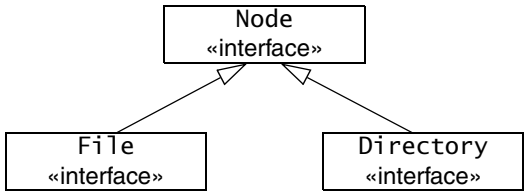


Figure 5.1. Inheritance Diagram of the File System.

The Slice definitions for this look as follows:

```
interface Node {
    // ...
};

interface File extends Node {
    // ...
};

interface Directory extends Node {
    // ...
};
```

Next, we need to think about what operations should be provided by each interface. Seeing that directories and files have names, we can add an operation to obtain the name of a directory or file to the `Node` base interface:

```
interface Node {
    idempotent string name();
};
```

The `File` interface provides operations to read and write a file. For simplicity, we limit ourselves to text files and we assume that `read` operations never fail and that only `write` operations can encounter error conditions. This leads to the following definitions:

```
exception GenericError {
    string reason;
};

sequence<string> Lines;

interface File extends Node {
    idempotent Lines read();
    idempotent void write (Lines text) throws GenericError;
};
```

Note that `read` and `write` are marked idempotent because either operation can safely be invoked with the same parameter value twice in a row: the net result of doing so is the same has having (successfully) called the operation only once.

The `write` operation can raise an exception of type `GenericError`. The exception contains a single `reason` data member, of type `string`. If a `write` operation fails for some reason (such as running out of file system space), the operation throws a `GenericError` exception, with an explanation of the cause of the failure provided in the `reason` data member.

Directories provide an operation to list their contents. Because directories can contain both directories and files, we take advantage of the polymorphism provided by the `Node` base interface:

```
sequence<Node*> NodeSeq;

interface Directory extends Node {
    idempotent NodeSeq list();
};
```

The `NodeSeq` sequence contains elements of type `Node*`. Because `Node` is a base interface of both `Directory` and `File`, the `NodeSeq` sequence can contain proxies of either type. (Obviously, the receiver of a `NodeSeq` must down-cast each element to either `File` or `Directory` in order to get at the operations provided by the derived interfaces; only the `name` operation in the `Node` base interface can be invoked directly, without doing a down-cast first. Note that, because the elements of `NodeSeq` are of type `Node*` (not `Node`), we are using pass-by-reference semantics: the values returned by the `list` operation are proxies that each point to a remote node on the server.

These definitions are sufficient to build a simple (but functional) file system. Obviously, there are still some unanswered questions, such as how a client obtains the proxy for the root directory. We will address these questions in the relevant implementation chapter.

5.4 The Complete Definition

We wrap our definitions in a module, resulting in the final definition as follows:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
```

```
        };

        sequence<Node*> NodeSeq;

        interface Directory extends Node {
            idempotent NodeSeq list();
        };
    };
```

Part III

C++ Mapping

Chapter 6
Client-Side Slice-to-C++ Mapping

6.1 Chapter Overview

In this chapter, we present the client-side Slice-to-C++ mapping (see Chapter 8 for the server-side mapping). One part of the client-side C++ mapping concerns itself with rules for representing each Slice data type as a corresponding C++ type; we cover these rules in Section 6.3 to Section 6.10. Another part of the mapping deals with how clients can invoke operations, pass and receive parameters, and handle exceptions. These topics are covered in Section 6.11 to Section 6.13. Slice classes have the characteristics of both data types and interfaces and are covered in Section 6.14. Section 6.15 presents asynchronous method invocations and, finally, Sections 6.16 and 6.17 show you how to use the Slice compiler and Slice checksums.

6.2 Introduction

The client-side Slice-to-C++ mapping defines how Slice data types are translated to C++ types, and how clients invoke operations, pass parameters, and handle errors. Much of the C++ mapping is intuitive. For example, Slice sequences map to STL vectors, so there is essentially nothing new you have to learn in order to use Slice sequences in C++.

The rules that make up the C++ mapping are simple and regular. In particular, the mapping is free from the potential pitfalls of memory management: all types are self-managed and automatically clean up when instances go out of scope. This means that you cannot accidentally introduce a memory leak by, for example, ignoring the return value of an operation invocation or forgetting to deallocate memory that was allocated by a called operation.

The C++ mapping is fully thread-safe. For example, the reference counting mechanism for classes (see Section 6.14.6) is interlocked against parallel access, so reference counts cannot be corrupted if a class instance is shared among a number of threads. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data—the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least Section 6.9 to Section 6.13 in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

A word of advice before you start: in order to use the C++ mapping, you should need no more than the Slice definition of your application and knowledge of the C++ mapping rules. In particular, looking through the generated header files in order to discern how to use the C++ mapping is likely to be confusing because the header files are not necessarily meant for human consumption and, occasionally, contain various cryptic constructs to deal with operating system and compiler idiosyncrasies. Of course, occasionally, you may want to refer to a header file to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

6.3 Mapping for Identifiers

Slice identifiers map to an identical C++ identifier. For example, the Slice identifier `Clock` becomes the C++ identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a C++ keyword, the corresponding C++ identi-

fier is prefixed with `_cpp_`. For example, the Slice identifier `while` is mapped as `_cpp_while`.[1]

A single Slice identifier often results in several C++ identifiers. For example, for a Slice interface named `Foo`, the generated C++ code uses the identifiers `Foo` and `FooPrx` (among others). If the interface has the name `while`, the generated identifiers are `_cpp_while` and `whilePrx` (*not* `_cpp_whilePrx`), that is, the prefix is applied only to those generated identifiers that actually require it.

6.4 Mapping for Modules

Slice modules map to C++ namespaces. The mapping preserves the nesting of the Slice definitions. For example:

```
module M1 {
    module M2 {
        // ...
    };
    // ...
};

// ...

module M1 {     // Reopen M1
    // ...
};
```

This definition maps to the corresponding C++ definition:

```
namespace M1 {
    namespace M2 {
        // ...
    }
    // ...
}

// ...
```

1. As suggested in Section 4.5.3 on page 92, you should try to avoid such identifiers as much as possible.

```
namespace M1 {  // Reopen M1
    // ...
}
```

If a Slice module is reopened, the corresponding C++ namespace is reopened as well.

6.5 The `Ice` Namespace

All of the APIs for the Ice run time are nested in the `Ice` namespace, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` namespace are generated from Slice definitions; other parts of the `Ice` namespace provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` namespace throughout the remainder of the book.

6.6 Mapping for Simple Built-In Types

The Slice built-in types are mapped to C++ types as shown in Table 6.1.

Table 6.1. Mapping of Slice built-in types to C++.

Slice	C++
`bool`	`bool`
`byte`	`Ice::Byte`
`short`	`Ice::Short`
`int`	`Ice::Int`
`long`	`Ice::Long`
`float`	`Ice::Float`
`double`	`Ice::Double`

Table 6.1. Mapping of Slice built-in types to C++.

Slice	C++
`string`	`std::string`

Slice `bool` and `string` map to C++ `bool` and `std::string`. The remaining built-in Slice types map to C++ type definitions instead of C++ native types. This allows the Ice run time to provide a definition as appropriate for each target architecture. (For example, `Ice::Int` might be defined as `long` on one architecture and as `int` on another.)

Note that `Ice::Byte` is a typedef for `unsigned char`. This guarantees that byte values are always in the range 0..255.

All the basic types are guaranteed to be distinct C++ types, that is, you can safely overload functions that differ in only the types in Table 6.1.

6.6.1 Alternate String Mapping

You can use a metadata directive, `["cpp:type:wstring"]`, to map strings to C++ `std::wstring`. This is useful for applications that use languages with alphabets that cannot be represented in 8-bit characters. The metadata directive can be applied to any Slice construct. For containers (such as modules, interfaces, or structures), the metadata directive applies to all strings within the container. A corresponding metadata directive, `["cpp:type:string"]` can be used to selectively override the mapping defined by the enclosing container. For example:

```
["cpp:type:wstring"]
struct S1 {
    string x;             // Maps to std::wstring
    ["cpp:type:wstring"]
    string y;             // Maps to std::wstring
    ["cpp:type:string"]
    string z;             // Maps to std::string
};

struct S2 {
    string x;             // Maps to std::string
    ["cpp:type:string"]
```

```
    string y;             // Maps to std::string
    ["cpp:type:wstring"]
    string z;             // Maps to std::wstring
};
```

With these metadata directives, the strings are mapped as indicated by the comments. By default, narrow strings are encoded as UTF-8, and wide strings use Unicode in an encoding that is appropriate for the platform on which the application executes. You can override the encoding for narrow and wide strings by registering a string converter with the Ice run time. (See Section 32.24 for details.)

6.7 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

6.7.1 Mapping for Enumerations

Enumerations map to the corresponding enumeration in C++. For example:

```
enum Fruit { Apple, Pear, Orange };
```

Not surprisingly, the generated C++ definition is identical:

```
enum Fruit { Apple, Pear, Orange };
```

6.7.2 Mapping for Structures

The mapping for structures maps Slice structures to C++ structures by default. In addition, you can use a metadata directive to map structures to classes (see page 189).

Default Mapping for Structures

Slice structures map to C++ structures with the same name. For each Slice data member, the C++ structure contains a public data member. For example, here is our `Employee` structure from Section 4.9.4 once more:

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The Slice-to-C++ compiler generates the following definition for this structure:

```
struct Employee {
    Ice::Long   number;
    std::string firstName;
    std::string lastName;
    bool operator==(const Employee&) const;
    bool operator!=(const Employee&) const;
    bool operator<(const Employee&) const;
    bool operator<=(const Employee&) const;
    bool operator>(const Employee&) const;
    bool operator>=(const Employee&) const;
};
```

For each data member in the Slice definition, the C++ structure contains a corresponding public data member of the same name. Constructors are intentionally omitted so that the C++ structure qualifies as a *plain old datatype* (POD).

Note that the structure also contains comparison operators. These operators have the following behavior:

- `operator==`

  Two structures are equal if (recursively), all its members are equal.

- `operator!=`

  Two structures are not equal if (recursively), one or more of its members are not equal.

- `operator<`
  `operator<=`
  `operator>`
  `operator>=`

  The comparison operators treat the members of a structure as sort order criteria: the first member is considered the first criterion, the second member the second criterion, and so on. Assuming that we have two `Employee` structures, `s1` and `s2`, this means that the generated code uses the following algorithm to compare `s1` and `s2`:

  ```
  bool Employee::operator<(const Employee& rhs) const
  {
  ```

```
        if (this == &rhs)   // Short-cut self-comparison
            return false;

        // Compare first members
        //
        if (number < rhs.number)
            return true;
        else if (rhs.number < number)
            return false;

        // First members are equal, compare second members
        //
        if (firstName < rhs.firstName)
            return true;
        else if (rhs.firstName < firstName)
            return false;

        // Second members are equal, compare third members
        //
        if (lastName < rhs.lastName)
            return true;
        else if (rhs.lastName < lastName)
            return false;

        // All members are equal, so return false
        return false;
    }
```

The comparison operators are provided to allow the use of structures as the key type of Slice dictionaries, which are mapped to `std::map` in C++ (see Section 6.7.5).

Note that copy construction and assignment always have deep-copy semantics. You can freely assign structures or structure members to each other without having to worry about memory management. The following code fragment illustrates both comparison and deep-copy semantics:

```
Employee e1, e2;
e1.firstName = "Bjarne";
e1.lastName = "Stroustrup";
e2 = e1;                        // Deep copy
assert(e1 == e2);
e2.firstName = "Andrew";        // Deep copy
e2.lastName = "Koenig";         // Deep copy
assert(e2 < e1);
```

Because strings are mapped to `std::string`, there are no memory management issues in this code and structure assignment and copying work as expected. (The default member-wise copy constructor and assignment operator generated by the C++ compiler do the right thing.)

Class Mapping for Structures

Occasionally, the mapping of Slice structures to C++ structures can be inefficient. For example, you may need to pass structures around in your application, but want to avoid having to make expensive copies of the structures. (This overhead becomes noticeable for structures with many complex data members, such as sequences or strings.) Of course, you could pass the structures by const reference, but that can create its own share of problems, such as tracking the life time of the structures to avoid ending up with dangling references.

For this reason, you can enable an alternate mapping that maps Slice structures to C++ classes. Classes (as opposed to structures) are reference-counted. Because the Ice C++ mapping provides smart pointers for classes (see Section 6.14.6), you can keep references to a class instance in many places in the code without having to worry about either expensive copying or life time issues.

The alternate mapping is enabled by a metadata directive, `["cpp:class"]`. Here is our Employee structure once again, but this time with the additional metadata directive:

```
["cpp:class"] struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

Here is the generated class:

```
class Employee : public IceUtil::Shared {
public:
    Employee() {}
    Employee(::Ice::Long,
             const ::std::string&,
             const ::std::string&);
    ::Ice::Long number;
    ::std::string firstName;
    ::std::string lastName;

    bool operator==(const Employee&) const;
    bool operator!=(const Employee&) const;
    bool operator<(const Employee&) const;
```

```
    bool operator<=(const Employee&) const;
    bool operator>(const Employee&) const;
    bool operator>=(const Employee&) const;
};
```

Note that the generated class, apart from a default constructor, has a constructor that accepts one argument for each member of the structure. This allows you to instantiate and initialize the class in a single statement (instead of having to first instantiate the class and then assign to its members).

As for the default structure mapping, the class contains one public data member for each data member of the corresponding Slice structure.

The comparison operators behave as for the default structure mapping.

For details on how to instantiate classes, and how to access them via smart pointers, please Section 6.14—the class mapping described there applies equally to Slice structures that are mapped to classes.

Constructors

Structures have an implicit default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

If you wish to ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition (see Section 4.9.2). The default constructor initializes each of these data members to its declared value.

The Slice compiler also generates a second constructor for structures that use the class mapping, and for structures having at least one member with a default value. This *one-shot* constructor has one parameter for each data member, allowing you to construct and initialize an instance in a single statement (instead of first having to construct the instance and then assigning to its members).

6.7.3 Mapping for Sequences

Here is the definition of our `FruitPlatter` sequence from Section 4.9.3 once more:

```
sequence<Fruit> FruitPlatter;
```

The Slice compiler generates the following C++ definition for the `FruitPlatter` sequence:

```
typedef std::vector<Fruit> FruitPlatter;
```

As you can see, the sequence simply maps to an STL vector. As a result, you can use the sequence like any other STL vector, for example:

```
// Make a small platter with one Apple and one Orange
//
FruitPlatter p;
p.push_back(Apple);
p.push_back(Orange);
```

As you would expect, you can use all the usual STL iterators and algorithms with this vector.

6.7.4 Custom Sequence Mapping

In addition to the default mapping of sequences to vectors, Ice supports three additional custom mappings for sequences.

STL Container Mapping for Sequences

You can override the default mapping of Slice sequences to C++ vectors with a metadata directive, for example:

```
[["cpp:include:list"]]

module Food {

    enum Fruit { Apple, Pear, Orange };

    ["cpp:type:std::list< ::Food::Fruit>"]
    sequence<Fruit> FruitPlatter;

};
```

With this metadata directive, the sequence now maps to a C++ `std::list`:

```
#include <list>

namespace Food {

    typedef std::list< Food::Fruit> FruitPlatter;

    // ...
}
```

The `cpp:type` metadata directive must be applied to a sequence definition; anything following the `cpp:type:` prefix is taken to be the name of the type. For example, we could use `["cpp:type:::std::list< ::Food::Fruit>"]`. In that case, the compiler would use a fully-qualified name to define the type:

```
typedef ::std::list< ::Food::Fruit> FruitPlatter;
```

Note that the code generator inserts whatever string you specify following the `cpp:type:` prefix literally into the generated code. This means that, to avoid C++ compilation failures due to unknown symbols, you should use a qualified name for the type.

Also note that, to avoid compilation errors in the generated code, you must instruct the compiler to generate an appropriate include directive with the `cpp:include` global metadata directive. This causes the compiler to add the line

```
#include <list>
```

to the generated header file.

Instead of `std::list`, you can specify a type of your own as the sequence type, for example:

```
[["cpp:include:FruitBowl.h"]]

module Food {

    enum Fruit { Apple, Pear, Orange };

    ["cpp:type:FruitBowl"]
    sequence<Fruit> FruitPlatter;

};
```

With these metadata directives, the compiler will use a C++ type `FruitBowl` as the sequence type, and add an include directive for the header file `FruitBowl.h` to the generated code.

You can use any class of your choice as a sequence type, but the class must meet certain requirements. (`vector`, `list`, and `deque` happen to meet these requirements.)

- The class must have a default constructor and a single-argument constructor that takes the size of the sequence as an argument of unsigned integral type.
- The class must have a copy constructor.
- The class must provide a member function `size` that returns the number elements in the sequence as an unsigned integral type.
- The class must provide a member function `swap` that swaps the contents of the sequence with another sequence of the same type.
- The class must define `iterator` and `const_iterator` types and must provide `begin` and `end` member functions with the usual semantics; the iterators must be comparable for equality and inequality.

Less formally, this means that if the class looks like a `vector`, `list`, or `deque` with respect to these points, you can use it as a custom sequence implementation.

In addition to modifying the type of a sequence itself, you can also modify the mapping for particular return values or parameters (see Section 6.12). For example:

```
[["cpp:include:list"]]
[["cpp:include:deque"]]

module Food {

    enum Fruit { Apple, Pear, Orange };

    sequence<Fruit> FruitPlatter;

    interface Market {
        ["cpp:type:list< ::Food::Fruit>"]
        FruitPlatter
        barter(
            ["cpp:type:deque< ::Food::Fruit>"] FruitPlatter offer
        );
    };

};
```

With this definition, the default mapping of `FruitPlatter` to a C++ `vector` still applies but the return value of `barter` is mapped as a `list`, and the `offer` parameter is mapped as a `deque`.

Array Mapping for Sequences

The array mapping for sequences applies to input parameters (see Section 6.12) and to out parameters of AMI (see Section 6.15) and AMD (see Section 8.8) operations . For example:

```
interface File {
    void write(["cpp:array"] Ice::ByteSeq contents);
};
```

The `cpp:array` metadata directive instructs the compiler to map the `contents` parameter to a pair of pointers. With this directive, the `write` method on the proxy has the following signature:

```
void write(const std::pair<const Ice::Byte*,
                           const Ice::Byte*>& contents);
```

To pass a byte sequence to the server, you pass a pair of pointers; the first pointer points at the beginning of the sequence, and the second point points one element past the end of the sequence.

Similarly, for the server side, the `write` method on the skeleton has the following signature:

```
virtual void write(const ::std::pair<const ::Ice::Byte*,
                                     const ::Ice::Byte*>&,
                   const ::Ice::Current& = ::Ice::Current()) = 0;
```

The passed pointers denote the beginning and end of the sequence as a range `[first, last)` (that is, they use the usual STL semantics for iterators).

The array mapping is useful to achieve zero-copy passing of sequences. The pointers point directly into the server-side transport buffer; this allows the server-side run time to avoid creating a `vector` to pass to the operation implementation, thereby avoiding both allocating memory for the sequence and copying its contents into that memory.

Note that you can use the array mapping for any sequence type. However, it provides a performance advantage only for byte sequences (on all platforms) and for sequences of integral or floating point types (x86 platforms only).

Also note that the called operation in the server must not store a pointer into the passed sequence because the transport buffer into which the pointer points is deallocated as soon as the operation completes.

Range Mapping for Sequences

The range mapping for sequences is similar to the array mapping and exists for the same purpose, namely, to enable zero-copy of sequence parameters:

```
interface File {
    void write(["cpp:range"] Ice::ByteSeq contents);
};
```

The `cpp:range` metadata directive instructs the compiler to map the `contents` parameter to a pair of `const_iterator`. With this directive, the `write` method on the proxy has the following signature:

```
void write(const std::pair<Ice::ByteSeq::const_iterator,
                           Ice::ByteSeq::const_iterator>& contents
);
```

Similarly, for the server side, the `write` method on the skeleton has the following signature:

```
virtual void write(const ::std::pair<
                        ::Ice::ByteSeq::const_iterator,
                        ::Ice::ByteSeq::const_iterator>&,
                   const ::Ice::Current& = ::Ice::Current()) = 0;
```

The passed iterators denote the beginning and end of the sequence as a range `[first, last)` (that is, they use the usual STL semantics for iterators).

The motivation for the range mapping is the same as for the array mapping: the passed iterators point directly into the server-side transport buffer and so avoid the need to create a temporary `vector` to pass to the operation.

As for the array mapping, the range mapping can be used with any sequence type, but offers a performance advantage only for byte sequences (on all platforms) and for sequences of integral type (x86 platforms only).

The operation must not store an iterator into the passed sequence because the transport buffer into which the iterator points is deallocated as soon as the operation completes.

You can optionally add a type name to the `cpp:range` metadata directive, for example:

```
interface File {
    void write(
        ["cpp:range:std::deque<Ice::Byte>"]
        Ice::ByteSeq contents);
};
```

This instructs the compiler to generate a pair of `const_iterator` for the specified type:

```
virtual void write(const ::std::pair<
                        std::deque<Ice::Byte>::const_iterator,
                        std::deque<Ice::Byte>::const_iterator>&,
                   const ::Ice::Current& = ::Ice::Current()) = 0;
```

This is useful if you want to combine the range mapping with a custom sequence type that behaves like an STL container.

6.7.5 Mapping for Dictionaries

Here is the definition of our `EmployeeMap` from Section 4.9.4 once more:

```
dictionary<long, Employee> EmployeeMap;
```

The following code is generated for this definition:

```
typedef std::map<Ice::Long, Employee> EmployeeMap;
```

Again, there are no surprises here: a Slice dictionary simply maps to an STL `map`. As a result, you can use the dictionary like any other STL `map`, for example:

```
EmployeeMap em;
Employee e;

e.number = 42;
e.firstName = "Stan";
e.lastName = "Lippman";
em[e.number] = e;

e.number = 77;
e.firstName = "Herb";
e.lastName = "Sutter";
em[e.number] = e;
```

Obviously, all the usual STL iterators and algorithms work with this map just as well as with any other STL container.

6.8 Mapping for Constants

Slice constant definitions map to corresponding C++ constant definitions. Here are the constant definitions we saw in Section 4.9.5 on page 103 once more:

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

```
const bool          AppendByDefault = true;
const Ice::Byte     LowerNibble =     15;
const std::string   Advice =          "Don't Panic!";
const Ice::Short    TheAnswer =       42;
const Ice::Double   PI =              3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit         FavoriteFruit =   Pear;
```

All constants are initialized directly in the header file, so they are compile-time constants and can be used in contexts where a compile-time constant expression is required, such as to dimension an array or as the `case` label of a `switch` statement.

6.9 Mapping for Exceptions

Here is a fragment of the Slice definition for our world time server from Section 4.10.5 on page 120 once more:

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

These exception definitions map as follows:

```
class GenericError: public Ice::UserException {
public:
    std::string reason;

    GenericError() {}
    explicit GenericError(const string&);
```

```
    virtual const std::string&  ice_name() const;
    virtual Ice::Exception*     ice_clone() const;
    virtual void                ice_throw() const;
    // Other member functions here...
};

class BadTimeVal: public GenericError {
public:
    BadTimeVal() {}
    explicit BadTimeVal(const string&);

    virtual const std::string&  ice_name() const;
    virtual Ice::Exception*     ice_clone() const;
    virtual void                ice_throw() const;
    // Other member functions here...
};

class BadZoneName: public GenericError {
public:
    BadZoneName() {}
    explicit BadZoneName(const string&);

    virtual const std::string&  ice_name() const;
    virtual Ice::Exception*     ice_clone() const;
    virtual void                ice_throw() const;
};
```

Each Slice exception is mapped to a C++ class with the same name. For each exception member, the corresponding class contains a public data member. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.)

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception has three additional member functions:

- `ice_name`

  As the name suggests, this member function returns the name of the exception. For example, if you call the `ice_name` member function of a `BadZoneName` exception, it (not surprisingly) returns the string `"BadZoneName"`. The `ice_name` member function is useful if you catch exceptions generically and want to produce a more meaningful diagnostic, for example:

  ```
  try {
      // ...
  } catch (const Ice::GenericError& e) {
  ```

```
    cerr << "Caught an exception: " << e.ice_name() << endl;
}
```

If an exception is raised, this code prints the name of the actual exception (`BadTimeVal` or `BadZoneName`) because the exception is being caught by reference (to avoid slicing).

- `ice_clone`

  This member function allows you to polymorphically clone an exception. For example:

  ```
  try {
      // ...
  } catch (const Ice::UserException& e) {
     Ice::UserException* copy = e.clone();
  }
  ```

  `ice_clone` is useful if you need to make a copy of an exception without knowing its precise run-time type. This allows you to remember the exception and throw it later by calling `ice_throw`.

- `ice_throw`

  `ice_throw` allows you to throw an exception without knowing its precise run-time type. It is implemented as:

  ```
  void
  GenericError::ice_throw() const
  {
      throw *this;
  }
  ```

  You can call `ice_throw` to throw an exception that you previously cloned with `ice_clone`.

Each exception has a default constructor. This constructor performs memberwise initialization; for simple built-in types, such as integers, the constructor performs no initialization, whereas complex types, such as strings, sequences, and dictionaries are initialized by their respective default constructors.

An exception also has a second constructor that accepts one argument for each exception member. This constructor allows you to instantiate and initialize an exception in a single statement, instead of having to first instantiate the exception and then assign to its members. For derived exceptions, the constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

Note that the generated exception classes contain other member functions that are not shown on page 197. However, those member functions are internal to the C++ mapping and are not meant to be called by application code.

All user exceptions ultimately inherit from `Ice::UserException`. In turn, `Ice::UserException` inherits from `Ice::Exception` (which is an alias for `IceUtil::Exception`):

```
namespace IceUtil {
    class Exception {
        virtual const std::string& ice_name() const;
        Exception*                 ice_clone() const;
        void                       ice_throw() const;
        virtual void               ice_print(std::ostream&) const;
        // ...
    };
    std::ostream& operator<<(std::ostream&, const Exception&);
    // ...
}

namespace Ice {
    typedef IceUtil::Exception Exception;

    class UserException: public Exception {
    public:
        virtual const std::string& ice_name() const = 0;
        // ...
    };
}
```

`Ice::Exception` forms the root of the exception inheritance tree. Apart from the usual `ice_name`, `ice_clone`, and `ice_throw` member functions, it contains the `ice_print` member functions. `ice_print` prints the name of the exception. For example, calling `ice_print` on a `BadTimeVal` exception prints:

```
BadTimeVal
```

To make printing more convenient, `operator<<` is overloaded for `Ice::Exception`, so you can also write:

```
try {
    // ...
} catch (const Ice::Exception& e) {
    cerr << e << endl;
}
```

This produces the same output because `operator<<` calls `ice_print` internally.

For Ice run time exceptions, `ice_print` also shows the file name and line number at which the exception was thrown.

Constructors

Exceptions have a default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

If you wish to ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition (see Section 4.10.2). The default constructor initializes each of these data members to its declared value.

Exceptions also have a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement (instead of first having to construct the instance and then assigning to its members). For derived exceptions, this constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

6.10 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice::LocalException` (which, in turn, derives from `Ice::Exception`). `Ice::LocalException` has the usual member functions (`ice_name`, `ice_clone`, `ice_throw`, and (inherited from `Ice::Exception`), `ice_print`, `ice_file`, and `ice_line`).

An inheritance diagram for user and run-time exceptions appears in Figure 4.4 on page 117. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice::Exception`

  This is the root of the complete inheritance tree. Catching `Ice::Exception` catches both user and run-time exceptions.

- `Ice::UserException`

  This is the root exception for all user exceptions. Catching `Ice::UserException` catches all user exceptions (but not run-time exceptions).

- `Ice::LocalException`

  This is the root exception for all run-time exceptions. Catching `Ice::LocalException` catches all run-time exceptions (but not user exceptions).

- `Ice::TimeoutException`

  This is the base exception for both operation-invocation and connection-establishment timeouts.

- `Ice::ConnectTimeoutException`

  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. An exception to this rule are `FacetNotExistException` (see Chapter 33) and `ObjectNotExistException` (see Chapter 34), which you may want to catch explicitly.

6.11 Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that represents the remote object. This makes the mapping easy and intuitive to use because, for all intents and purposes (apart from error semantics), making a remote procedure call is no different from making a local procedure call.

6.11.1 Proxy Classes and Proxy Handles

On the client side, interfaces map to classes with member functions that correspond to the operations on those interfaces. Consider the following simple interface:

```
module M {
    interface Simple {
        void op();
    };
};
```

The Slice compiler generates the following definitions for use by the client:

```
namespace IceProxy {
    namespace M {
        class Simple;
    }
}

namespace M {
    class Simple;
    typedef IceInternal::ProxyHandle< ::IceProxy::M::Simple>
                                                    SimplePrx;
    typedef IceInternal::Handle< ::M::Simple> SimplePtr;
}

namespace IceProxy {
    namespace M {
        class Simple : public virtual IceProxy::Ice::Object {
        public:
            typedef ::M::SimplePrx ProxyType;
            typedef ::M::SimplePtr PointerType;

            void op();
            void op(const Ice::Context&);
            // ...
        };
    };
}
```

As you can see, the compiler generates a *proxy class* `Simple` in the `IceProxy::M` namespace, as well as a *proxy handle* `M::SimplePrx`. In general, for a module `M`, the generated names are `::IceProxy::M::`*`<interface-name>`* and `::M::`*`<interface-name>`*`Prx`.

In the client's address space, an instance of `IceProxy::M::Simple` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy class instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `Simple` inherits from `IceProxy::Ice::Object`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`. For each operation in the interface, the proxy class has two overloaded member functions of the same name. For the preceding example, we find that the operation `op` has been mapped to two member functions `op`.

One of the overloaded member functions has a trailing parameter of type `Ice::Context`. This parameter is for use by the Ice run time to store information about how to deliver a request; normally, you do not need to supply a value here and can pretend that the trailing parameter does not exist. (We examine the `Ice::Context` parameter in detail in Chapter 32. The parameter is also used by IceStorm—see Chapter 44.)

Client-side application code never manipulates proxy class instances directly. In fact, you are not allowed to instantiate a proxy class directly. The following code will not compile because `Ice::Object` is an abstract base class with a protected constructor and destructor:

```
IceProxy::M::Simple s;  // Compile-time error!
```

Proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. When the client receives a proxy from the run time, it is given a *proxy handle* to the proxy, of type `<interface-name>Prx` (`SimplePrx` for the preceding example). The client accesses the proxy via its proxy handle; the handle takes care of forwarding operation invocations to its underlying proxy, as well as reference-counting the proxy. This means that no memory-management issues can arise: deallocation of a proxy is automatic and happens once the last handle to the proxy disappears (goes out of scope).

Because the application code always uses proxy handles and never touches the proxy class directly, we usually use the term *proxy* to denote both proxy handle and proxy class. This reflects the fact that, in actual use, the proxy handle looks and feels like the underlying proxy class instance. If the distinction is important, we use the terms *proxy class*, *proxy class instance*, and *proxy handle*.

Also note that the generated proxy class contains type definitions for `ProxyType` and `PointerType`. These are provided so you can refer to the proxy type

and smart pointer type (see Section 6.14.6) in template definitions without having to resort to preprocessor trickery, for example:

```
template<typename T>
class ProxyWrapper {
public:
    T::ProxyType proxy() const;
    // ...
};
```

6.11.2 Methods on Proxy Handles

As we saw for the preceding example, the handle is actually a template of type `IceInternal::ProxyHandle` that takes the proxy class as the template parameter. This template has the usual constructor, copy constructor, and assignment operator:

- Default constructor

  You can default-construct a proxy handle. The default constructor creates a proxy that points nowhere (that is, points at no object at all.) If you invoke an operation on such a null proxy, you get an `IceUtil::NullHandleException`:

  ```
  try {
      SimplePrx s;        // Default-constructed proxy
      s->op();            // Call via nil proxy
      assert(0);          // Can't get here
  } catch (const IceUtil::NullHandleException&) {
      cout << "As expected, got a NullHandleException" << endl;
  }
  ```

- Copy constructor

  The copy constructor ensures that you can construct a proxy handle from another proxy handle. Internally, this increments a reference count on the proxy; the destructor decrements the reference count again and, once the count drops to zero, deallocates the underlying proxy class instance. That way, memory leaks are avoided:

  ```
  {                                // Enter new scope
      SimplePrx s1 = ...;          // Get a proxy from somewhere
      SimplePrx s2(s1);            // Copy-construct s2
      assert(s1 == s2);            // Assertion passes
  }                                // Leave scope; s1, s2, and the
                                   // underlying proxy instance
  ```

```
                                        // are deallocated
```

Note the assertion in this example: proxy handles support comparison (see Section 6.11.4).

- Assignment operator

  You can freely assign proxy handles to each other. The handle implementation ensures that the appropriate memory-management activities take place. Self-assignment is safe and you do not have to guard against it:

```
SimplePrx s1 = ...;     // Get a proxy from somewhere
SimplePrx s2;           // s2 is nil
s2 = s1;                // both point at the same object
s1 = 0;                 // s1 is nil
s2 = 0;                 // s2 is nil
```

  Widening assignments work implicitly. For example, if we have two interfaces, `Base` and `Derived`, we can widen a `DerivedPrx` to a `BasePrx` implicitly:

```
BasePrx base;
DerivedPrx derived;
base = derived;         // Fine, no problem
derived = base;         // Compile-time error
```

  Implicit narrowing conversions result in a compile error, so the usual C++ semantics are preserved: you can always assign a derived type to a base type, but not vice versa.

- Checked cast

  Proxy handles provide a `checkedCast` method:

```
namespace IceInternal {
  template<typename T>
  class ProxyHandle : public IceUtil::HandleBase<T> {
  public:
    template<class Y>
    static ProxyHandle checkedCast(const ProxyHandle<Y>& r);

    template<class Y>
    static ProxyHandle checkedCast(const ProxyHandle<Y>& r,
                                   const ::Ice::Context& c);

    // ...
  };
```

```
}
```

A checked cast has the same function for proxies as a C++ `dynamic_cast` has for pointers: it allows you to assign a base proxy to a derived proxy. If the base proxy's actual run-time type is compatible with the derived proxy's static type, the assignment succeeds and, after the assignment, the derived proxy denotes the same object as the base proxy. Otherwise, if the base proxy's run-time type is incompatible with the derived proxy's static type, the derived proxy is set to null. Here is an example to illustrate this:

```
BasePrx base = ...;     // Initialize base proxy
DerivedPrx derived;
derived = DerivedPrx::checkedCast(base);
if (derived) {
        // Base has run-time type Derived,
        // use derived...
} else {
        // Base has some other, unrelated type
}
```

The expression `DerivedPrx::checkedCast(base)` tests whether `base` points at an object of type `Derived` (or an object with a type that is derived from `Derived`). If so, the cast succeeds and `derived` is set to point at the same object as `base`. Otherwise, the cast fails and `derived` is set to the null proxy.

Note that `checkedCast` is a static member function so, to do a down-cast, you always use the syntax `<`*`interface-name`*`>Prx::checkedCast`.

Also note that you can use proxies in boolean contexts. For example, `if (proxy)` returns true if the proxy is not null (see Section 6.11.4).

A `checkedCast` typically results in a remote message to the server.[2] The message effectively asks the server "is the object denoted by this reference of type `Derived`?" The reply from the server is communicated to the application code in form of a successful (non-null) or unsuccessful (null) result. Sending a remote message is necessary because, as a rule, there is no way for the client to find out what the actual run-time type of a proxy is without confirmation from the server. (For example, the server may replace the implementation of

2. In some cases, the Ice run time can optimize the cast and avoid sending a message. However, the optimization applies only in narrowly-defined circumstances, so you cannot rely on a `checkedCast` not sending a message.

the object for an existing proxy with a more derived one.) This means that you have to be prepared for a `checkedCast` to fail. For example, if the server is not running, you will receive a `ConnectFailedException`; if the server is running, but the object denoted by the proxy no longer exists, you will receive an `ObjectNotExistException`.

- Unchecked cast

  In some cases, it is known that an object supports a more derived interface than the static type of its proxy. For such cases, you can use an unchecked down-cast:

```
namespace IceInternal {
  template<typename T>
  class ProxyHandle : public IceUtil::HandleBase<T> {
  public:
    template<class Y>
    static ProxyHandle uncheckedCast(const ProxyHandle<Y>& r);
    // ...
  };
}
```

  An `uncheckedCast` provides a down-cast *without* consulting the server as to the actual run-time type of the object, for example:

```
BasePrx base = ...;     // Initialize to point at a Derived
DerivedPrx derived;
derived = DerivedPrx::uncheckedCast(base);
// Use derived...
```

  You should use an `uncheckedCast` only if you are certain that the proxy indeed supports the more derived type: an `uncheckedCast`, as the name implies, is not checked in any way; it does not contact the object in the server and, if it fails, it does not return null. (An unchecked cast is implemented internally like a C++ `static_cast`, no checks of any kind are made). If you use the proxy resulting from an incorrect `uncheckedCast` to invoke an operation, the behavior is undefined. Most likely, you will receive an `OperationNotExistException`, but, depending on the circumstances, the Ice run time may also report an exception indicating that unmarshaling has failed, or even silently return garbage results.

  Despite its dangers, `uncheckedCast` is still useful because it avoids the cost of sending a message to the server. And, particularly during initialization (see Chapter 7), it is common to receive a proxy of static type

`Ice::Object`, but with a known run-time type. In such cases, an `uncheckedCast` saves the overhead of sending a remote message.

- Stream insertion and stringification

  For convenience, proxy handles also support insertion of a proxy into a stream, for example:

  ```
  Ice::ObjectPrx p = ...;
  cout << p << endl;
  ```

  This code is equivalent to writing:

  ```
  Ice::ObjectPrx p = ...;
  cout << p->ice_toString() << endl;
  ```

  Either code prints the stringified proxy (see Appendix E). You could also achieve the same thing by writing:

  ```
  Ice::ObjectPrx p = ...;
  cout << communicator->proxyToString(p) << endl;
  ```

  The advantage of using the `ice_toString` member function instead of `proxyToString` is that you do not need to have the communicator available at the point of call.

6.11.3 Using Proxy Methods

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy (see Section 32.11). Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second timeout as shown below:

```
Ice::ObjectPrx proxy = communicator->stringToProxy(...);
proxy = proxy->ice_timeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method. The example below demonstrates these semantics:

```
Ice::ObjectPrx base = communicator->stringToProxy(...);
HelloPrx hello = HelloPrx::checkedCast(base);
hello = hello->ice_timeout(10000); # Type is preserved
hello->sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

6.11.4 Object Identity and Proxy Comparison

Apart from the methods discussed in Section 6.11.2, proxy handles also support comparison. Specifically, the following operators are supported:

- `operator==`
  `operator!=`

  These operators permit you to compare proxies for equality and inequality. To test whether a proxy is null, use a comparison with the literal `0`, for example:

  ```
  if (proxy == 0)
      // It's a nil proxy
  else
      // It's a non-nil proxy
  ```

- `operator<`
  `operator<=`
  `operator>`
  `operator>=`

  Proxies support comparison. This allows you to place proxies into STL containers such as maps or sorted lists.

- Boolean comparison

  Proxies have a conversion operator to `bool`. The operator returns true if a proxy is not null, and false otherwise. This allows you to write:

  ```
  BasePrx base = ...;
  if (base)
          // It's a non-nil proxy
  else
          // It's a nil proxy
  ```

Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `==` and `!=` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

```
Ice::ObjectPrx p1 = ...;        // Get a proxy...
Ice::ObjectPrx p2 = ...;        // Get another proxy...

if (p1 != p2) {
    // p1 and p2 denote different objects       // WRONG!
} else {
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `==`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `==`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use helper functions in the `Ice` namespace:

```
namespace Ice {

    bool proxyIdentityLess(const ObjectPrx&,
                           const ObjectPrx&);
    bool proxyIdentityEqual(const ObjectPrx&,
                            const ObjectPrx&);
    bool proxyIdentityAndFacetLess(const ObjectPrx&,
                                   const ObjectPrx&);
    bool proxyIdentityAndFacetEqual(const ObjectPrx&,
                                    const ObjectPrx&);

}
```

The `proxyIdentityEqual` function returns true if the object identities embedded in two proxies are the same and ignores other information in the proxies, such as facet and transport information. To include the facet name (see Chapter 33) in the comparison, use `proxyIdentityAndFacetEqual` instead.

The `proxyIdentityLess` function establishes a total ordering on proxies. It is provided mainly so you can use object identity comparison with STL sorted containers. (The function uses `name` as the major ordering criterion, and `category` as the minor ordering criterion.) The `proxyIdentityAndFacetLess`

function behaves similarly to `proxyIdentityLess`, except that it also compares the facet names of the proxies when their identities are equal.

`proxyIdentityEqual` and `proxyIdentityAndFacetLess` allow you to correctly compare proxies for object identity. The example below demonstrates how to use `proxyIdentityEqual`:

```
Ice::ObjectPrx p1 = ...;        // Get a proxy...
Ice::ObjectPrx p2 = ...;        // Get another proxy...

if (!Ice::proxyIdentityEqual(p1, p2) {
    // p1 and p2 denote different objects       // Correct
} else {
    // p1 and p2 denote the same object         // Correct
}
```

6.12 Mapping for Operations

As we saw in Section 6.11, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy handle. For example, here is part of the definitions for our file system from Section 5.4:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The proxy class for the `Node` interface, tidied up to remove irrelevant detail, is as follows:

```
namespace IceProxy {
    namespace Filesystem {
        class Node : virtual public IceProxy::Ice::Object {
        public:
            std::string name();
            // ...
        };
        typedef IceInternal::ProxyHandle<Node> NodePrx;
```

```
        // ...
    }
    // ...
}
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

```
NodePrx node = ...;             // Initialize proxy
string name = node->name();     // Get name via RPC
```

The proxy handle overloads `operator->` to forward method calls to the underlying proxy class instance which, in turn, sends the operation invocation to the server, waits until the operation is complete, and then unmarshals the return value and returns it to the caller.

Because the return value is of type `string`, it is safe to ignore the return value. For example, the following code contains no memory leak:

```
NodePrx node = ...;             // Initialize proxy
node->name();                   // Useless, but no leak
```

This is true for all mapped Slice types: you can safely ignore the return value of an operation, no matter what its type—return values are always returned by value. If you ignore the return value, no memory leak occurs because the destructor of the returned value takes care of deallocating memory as needed.

6.12.1 Normal and `idempotent` Operations

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy methods is concerned, `idempotent` has no effect. For example, consider the following interface:

```
interface Example {
                string op1();
    idempotent  string op2();
    idempotent  void op3(string s);
};
```

The proxy class for this interface looks like this:

```
namespace IceProxy {
    class Example : virtual public IceProxy::Ice::Object {
    public:
        std::string op1();
        std::string op2();              // idempotent
```

```
        void op3(const std::string&);   // idempotent
        // ...
    };
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the mapping to be unaffected by the `idempotent` keyword.

6.12.2 Passing Parameters

In-Parameters

The parameter passing rules for the C++ mapping are very simple: parameters are passed either by value (for small values) or by `const` reference (for values that are larger than a machine word). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation (with some caveats—see page 217 and page 1093).

Here is an interface with operations that pass parameters of various types from client to server:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following code for this definition:

```
struct NumberAndString {
    Ice::Int x;
    std::string str;
    // ...
};

typedef std::vector<std::string> StringSeq;
```

```
typedef std::map<Ice::Long, StringSeq> StringTable;

namespace IceProxy {
    class ClientToServer : virtual public IceProxy::Ice::Object {
    public:
        void op1(Ice::Int, Ice::Float, bool, const std::string&);
        void op2(const NumberAndString&,
                 const StringSeq&,
                 const StringTable&);
        void op3(const ClientToServerPrx&);
        // ...
    };
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

```
ClientToServerPrx p = ...;              // Get proxy...

p->op1(42, 3.14, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14;
bool b = true;
string s = "Hello world!";
p->op1(i, f, b, s);                     // Pass simple variables

NumberAndString ns = { 42, "The Answer" };
StringSeq ss;
ss.push_back("Hello world!");
StringTable st;
st[0] = ss;
p->op2(ns, ss, st);                     // Pass complex variables

p->op3(p);                              // Pass proxy
```

You can pass either literals or variables to the various operations. Because everything is passed by value or `const` reference, there are no memory-management issues to consider.

Out-Parameters

The C++ mapping passes out-parameters by reference. Here is the Slice definition from page 214 once more, modified to pass all parameters in the `out` direction:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Slice compiler generates the following code for this definition:

```
namespace IceProxy {
    class ServerToClient : virtual public IceProxy::Ice::Object {
    public:
       void op1(Ice::Int&, Ice::Float&, bool&, std::string&);
       void op2(NumberAndString&, StringSeq&, StringTable&);
       void op3(ServerToClientPrx&);
       // ...
    };
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

```
ServerToClientPrx p = ...;      // Get proxy...

int i;
float f;
bool b;
string s;

p->op1(i, f, b, s);
// i, f, b, and s contain updated values now

NumberAndString ns;
StringSeq ss;
StringTable st;

p->op2(ns, ss, st);
```

```
// ns, ss, and st contain updated values now

p->op3(p);
// p has changed now!
```

Again, there are no surprises in this code: the caller simply passes variables to an operation; once the operation completes, the values of those variables will be set by the server.

It is worth having another look at the final call:

```
p->op3(p);      // Weird, but well-defined
```

Here, `p` is the proxy that is used to dispatch the call. That same variable `p` is also passed as an out-parameter to the call, meaning that the server will set its value. In general, passing the same parameter as both an input and output parameter is safe: the Ice run time will correctly handle all locking and memory-management activities.

Another, somewhat pathological example is the following:

```
sequence<int> Row;
sequence<Row> Matrix;

interface MatrixArithmetic {
    void multiply(Matrix m1,
                  Matrix m2,
                  out Matrix result);
};
```

Given a proxy to a `MatrixArithmetic` interface, the client code could do the following:

```
MatrixArithmeticPrx ma = ...;      // Get proxy...
Matrix m1 = ...;                   // Initialize one matrix
Matrix m2 = ...;                   // Initialize second matrix
ma->squareAndCubeRoot(m1, m2, m1); // !!!
```

This code is technically legal, in the sense that no memory corruption or locking issues will arise, but it has surprising behavior: because the same variable `m1` is passed as an input parameter as well as an output parameter, the final value of `m1` is indeterminate—in particular, if client and server are collocated in the same address space, the implementation of the operation will overwrite parts of the input matrix `m1` in the process of computing the result because the result is written to the same physical memory location as one of the inputs. In general, you should take care when passing the same variable as both an input and output parameter and only do so if the called operation guarantees to be well-behaved in this case.

Chained Invocations

Consider the following simple interface containing two operations, one to set a value and one to get it:

```
interface Name {
    string getName();
    void setName(string name);
};
```

Suppose we have two proxies to interfaces of type `Name`, `p1` and `p2`, and chain invocations as follows:

```
p2->setName(p1->getName());
```

This works exactly as intended: the value returned by `p1` is transferred to `p2`. There are no memory-management or exception safety issues with this code.

6.13 Exception Handling

Any operation invocation may throw a run-time exception (see Section 6.10 on page 201) and, if the operation has an exception specification, may also throw user exceptions (see Section 6.9 on page 197). Suppose we have the following simple interface:

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as C++ exceptions, so you can simply enclose one or more operation invocations in a `try–catch` block:

```
ChildPrx child = ...;           // Get proxy...
try {
    child->askToCleanUp();      // Give it a try...
} catch (const Tantrum& t) {
    cout << "The child says: " << t.reason << endl;
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be dealt with by exception handlers higher in the hierarchy. For example:

```
void run()
{
    ChildPrx child = ...;       // Get proxy...
    try {
        child->askToCleanUp();  // Give it a try...
    } catch (const Tantrum& t) {
        cout << "The child says: " << t.reason << endl;

        child->scold();         // Recover from error...
    }
    child->praise();            // Give positive feedback...
}

int
main(int argc, char* argv[])
{
    int status = 1;
    try {
        // ...
        run();
        // ...
        status = 0;
    } catch (const Ice::Exception& e) {
        cerr << "Unexpected run-time error: " << e << endl;
    }
    // ...
    return status;
}
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Chapter 3.)

For efficiency reasons, you should always catch exceptions by `const` reference. This permits the compiler to avoid calling the exception's copy constructor (and, of course, prevents the exception from being sliced to a base type).

Exceptions and Out-Parameters

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may have still have its original value or may have been changed by the operation's implementation in the target object.

In other words, for out-parameters, Ice provides the weak exception guarantee [21] but does not provide the strong exception guarantee.[3]

Exceptions and Return Values

For return values, C++ provides the guarantee that a variable receiving the return value of an operation will not be overwritten if an exception is thrown. (Of course, this guarantee holds only if you do not use the same variable as both an out-parameter and to receive the return value of an invocation (see page 217).)

6.14 Mapping for Classes

Slice classes are mapped to C++ classes with the same name. The generated class contains a public data member for each Slice data member, and a virtual member function for each operation. Consider the following class definition:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler generates the following code for this definition:[4]

```
class TimeOfDay;

typedef IceInternal::ProxyHandle<IceProxy::TimeOfDay> TimeOfDayPrx
;
typedef IceInternal::Handle<TimeOfDay> TimeOfDayPtr;

class TimeOfDay : virtual public Ice::Object {
public:
    Ice::Short hour;
    Ice::Short minute;
    Ice::Short second;
```

3. This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

4. The `ProxyType` and `PointerType` definitions are for template programming (see page 204).

```
    virtual std::string format() = 0;

    TimeOfDay() {};
    TimeOfDay(Ice::Short, Ice::Short, Ice::Short);

    virtual bool ice_isA(const std::string&);
    virtual const std::string& ice_id();
    static const std::string& ice_staticId();

    typedef TimeOfDayPrx ProxyType;
    typedef TimeOfDayPtr PointerType;

    // ...
};
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice::Object`. This means that all classes implicitly inherit from `Ice::Object`, which is the ultimate ancestor of all classes. Note that `Ice::Object` is *not* the same as `IceProxy::Ice::Object`. In other words, you *cannot* pass a class where a proxy is expected and vice versa. (However, you *can* pass a proxy for the class—see Section 6.14.6.)

2. The generated class contains a public member for each Slice data member.

3. The generated class has a constructor that takes one argument for each data member, as well as a default constructor.

4. The generated class contains a pure virtual member function for each Slice operation.

5. The generated class contains additional member functions: `ice_isA`, `ice_id`, `ice_staticId`, and `ice_factory`.

6. The compiler generates a type definition `TimeOfDayPtr`. This type implements a smart pointer that wraps dynamically-allocated instances of the class. In general, the name of this type is `<class-name>Ptr`. Do not confuse this with `<class-name>Prx`—that type exists as well, but is the proxy handle for the class, not a smart pointer.

There is quite a bit to discuss here, so we will look at each item in turn.

6.14.1 Inheritance from `Ice::Object`

Like interfaces, classes implicitly inherit from a common base class, `Ice::Object`. However, as shown in Figure 6.1, classes inherit from `Ice::Object` instead of `Ice::ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

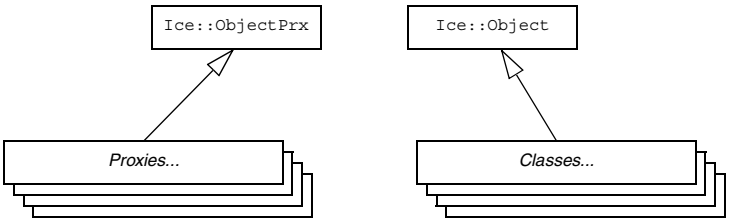


Figure 6.1. Inheritance from `Ice::ObjectPrx` and `Ice::Object`.

`Ice::Object` contains a number of member functions:

```
namespace Ice {
    class Object : public virtual IceInternal::GCShared {
    public:
        virtual bool ice_isA(const std::string&,
                             const Current& = Current()) const;
        virtual void ice_ping(const Current&  = Current()) const;
        virtual std::vector<std::string> ice_ids(
                             const Current& = Current()) const;
        virtual const std::string& ice_id(
                             const Current& = Current()) const;
        static const std::string& ice_staticId();
        virtual Ice::Int ice_getHash() const;
        virtual ObjectPtr ice_clone() const;

        virtual void ice_preMarshal();
        virtual void ice_postUnmarshal();

        virtual DispatchStatus ice_dispatch(
                  Ice::Request&,
                  const DispatchInterceptorAsyncCallbackPtr& = 0);

        virtual bool operator==(const Object&) const;
        virtual bool operator!=(const Object&) const;
```

```
        virtual bool operator<(const Object&) const;
        virtual bool operator<=(const Object&) const;
        virtual bool operator>(const Object&) const;
        virtual bool operator>=(const Object&) const;
    };
}
```

The member functions of `Ice::Object` behave as follows:

- `ice_isA`

  This function returns `true` if the object supports the given type ID, and `false` otherwise.

- `ice_ping`

  As for interfaces, `ice_ping` provides a basic reachability test for the class.

- `ice_ids`

  This function returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.

- `ice_id`

  This function returns the actual run-time type ID for a class. If you call `ice_id` through a smart pointer to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.

- `ice_staticId`

  This function returns the static type ID of a class.

- `ice_getHash`

  This method returns a hash value for the class, allowing you to easily place classes into hash tables.

- `ice_clone`

  This function makes a polymorphic shallow copy of a class (see page 236).

- `ice_preMarshal`

  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.

- `ice_postUnmarshal`

  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.

- `ice_dispatch`

  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors (see Section 32.23).

- `operator==`
  `operator!=`
  `operator<`
  `operator<=`
  `operator>`
  `operator>=`

  The comparison operators permit you to use classes as elements of STL sorted containers. Note that sort order, unlike for structures (see Section 6.12), is based on the memory address of the class, not on the contents of its data members of the class.

6.14.2 Data Members of Classes

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

```
class TimeOfDay {
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler produces the following generated code for this definition:

```
class TimeOfDay : virtual public Ice::Object {
public:

    virtual std::string format() = 0;

    // ...
```

```
protected:

    Ice::Short hour;
    Ice::Short minute;
    Ice::Short second;
};
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

```
["protected"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

6.14.3 Class Constructors

Classes have a default constructor that default-constructs each data member. Members having a complex type, such as strings, sequences, and dictionaries, are initialized by their own default constructor. However, the default constructor performs no initialization for members having one of the simple built-in types boolean, integer, floating point, or enumeration. For such a member, it is not safe to assume that the member has a reasonable default value. This is especially true for enumerated types as the member's default value may be outside the legal range for the enumeration, in which case an exception will occur during marshaling unless the member is explicitly set to a legal value.

If you wish to ensure that data members of primitive types are initialized to reasonable values, you can declare default values in your Slice definition (see Section 4.11.1). The default constructor initializes each of these data members to its declared value.

Classes also have a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement (instead of first having to construct the instance and then assigning to its members).

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order. For example:

```
class Base {
    int i;
};

class Derived extends Base {
    string s;
};
```

This generates:

```
class Base : virtual public ::Ice::Object
{
public:
    ::Ice::Int i;

    Base() {};
    explicit Base(::Ice::Int);

    // ...
};

class Derived : public Base
{
public:
    ::std::string s;

    Derived() {};
    Derived(::Ice::Int, const ::std::string&);

    // ...
};
```

Note that single-parameter constructors are defined as `explicit`, to prevent implicit argument conversions.

By default, derived classes derive non-virtually from their base class. If you need virtual inheritance, you can enable it using the `["cpp:virtual"]` metadata directive (see Appendix B).

6.14.4 Operations of Classes

Operations of classes are mapped to pure virtual member functions in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:[5]

```
class TimeOfDayI : virtual public TimeOfDay {
public:
    virtual std::string format() {
        std::ostringstream s;
        s << setw(2) << setfill('0') << hour << ":";
        s << setw(2) << setfill('0') << minute << ":";
        s << setw(2) << setfill('0') << second;
        return s.c_str();
    }

protected:
    virtual ~TimeOfDayI() {}  // Optional
};
```

6.14.5 Class Factories

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };
```

5. We discuss the motivation for the protected destructor on page 238.

```
    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };
};
```

The object factory's `create` operation is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` operation is called by the Ice run time when its communicator is destroyed. A possible implementation of our object factory is:

```
class ObjectFactory : public Ice::ObjectFactory {
public:
    virtual Ice::ObjectPtr create(const std::string& type) {
        assert(type == M::TimeOfDay::ice_staticId());
        return new TimeOfDayI;
    }
    virtual void destroy() {}
};
```

The `create` method is passed the type ID (see Section 4.13) of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::M::TimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `TimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoObjectFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

```
Ice::CommunicatorPtr ic = ...;
ic->addObjectFactory(new ObjectFactory,
                     M::TimeOfDay::ice_staticId());
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::M::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator—if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called. However, calls to `create` can be made concurrently.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

6.14.6 Smart Pointers for Classes

A recurring theme for C++ programmers is the need to deal with memory allocations and deallocations in their programs. The difficulty of doing so is well known: in the face of exceptions, multiple return paths from functions, and callee-allocated memory that must be deallocated by the caller, it can be extremely difficult to ensure that a program does not leak resources. This is particularly important in multi-threaded programs: if you do not rigorously track ownership of dynamic memory, a thread may delete memory that is still used by another thread, usually with disastrous consequences.

To alleviate this problem, Ice provides smart pointers for classes. These smart pointers use reference counting to keep track of each class instance and, when the last reference to a class instance disappears, automatically delete the instance.[6] Smart pointers are generated by the Slice compiler for each class type. For a Slice class *<class-name>*, the compiler generates a C++ smart pointer called *<class-name>*`Ptr`. Rather than showing all the details of the generated class, here is the basic usage pattern: whenever you allocate a class instance on the heap, you simply assign the pointer returned from `new` to a smart pointer for the class. Thereafter, memory management is automatic and the class instance is deleted once the last smart pointer for it goes out of scope:

```
{                                       // Open scope
    TimeOfDayPtr tod = new TimeOfDayI;  // Allocate instance
    // Initialize...
    tod->hour = 18;
    tod->minute = 11;
    tod->second = 15;
    // ...
}                                       // No memory leak here!
```

As you can see, you use `operator->` to access the members of the class via its smart pointer. When the `tod` smart pointer goes out of scope, its destructor runs and, in turn, the destructor takes care of calling `delete` on the underlying class instance, so no memory is leaked.

The smart pointers perform reference counting of their underlying class instance:

- The constructor of a class sets its reference count to zero.
- Initializing a smart pointer with a dynamically-allocated class instance causes the smart pointer to increment the reference count for the class by one.
- Copy constructing a smart pointer increments the reference count for the class by one.
- Assigning one smart pointer to another increments the target's reference count and decrements the source's reference count. (Self-assignment is safe.)
- The destructor of a smart pointer decrements the reference count by one and calls `delete` on its class instance if the reference count drops to zero.

6. Smart pointer classes are an example of the *RAII* (*Resource Acquisition Is Initialization*) idiom [20].

Figure 6.2 shows the situation after default-constructing a smart pointer as follows:

```
TimeOfDayPtr tod;
```

This creates a smart pointer with an internal null pointer.

Figure 6.2. Newly initialized smart pointer.

Constructing a class instance creates that instance with a reference count of zero; the assignment to the class pointer causes the smart pointer to increment the class's reference count:

```
tod = new TimeOfDayI;   // Refcount == 1
```

The resulting situation is shown in Figure 6.3.

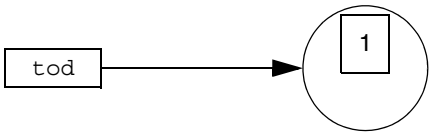


Figure 6.3. Initialized smart pointer.

Assigning or copy-constructing a smart pointer assigns and copy-constructs the smart pointer (not the underlying instance) and increments the reference count of the instance:

```
TimeOfDayPtr tod2(tod); // Copy-construct tod2
TimeOfDayPtr tod3;
tod3 = tod;             // Assign to tod3
```

The situation after executing these statements is shown in Figure 6.4:

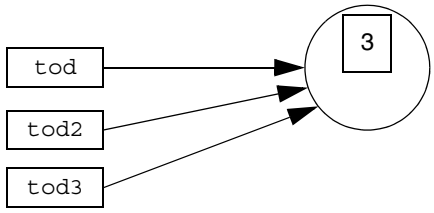


Figure 6.4. Three smart pointers pointing at the same class instance.

Continuing the example, we can construct a second class instance and assign it to one of the original smart pointers, `tod2`:

```
tod2 = new TimeOfDayI;
```

This decrements the reference count of the class originally denoted by `tod2` and increments the reference count of the class that is assigned to `tod2`. The resulting situation is shown in Figure 6.5.

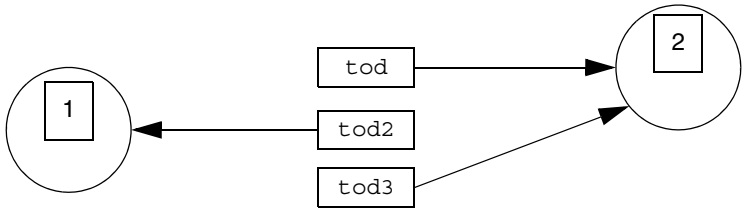


Figure 6.5. Three smart pointers and two instances.

You can clear a smart pointer by assigning zero to it:

```
tod = 0;         // Clear handle
```

As you would expect, this decrements the reference count of the instance, as shown in Figure 6.6.

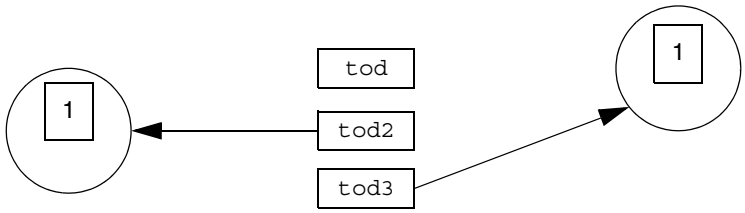


Figure 6.6. Decremented reference count after clearing a smart pointer.

If a smart pointer goes out of scope, is cleared, or has a new instance assigned to it, the smart pointer decrements the reference count of its instance; if the reference count drops to zero, the smart pointer calls `delete` to deallocate the instance. The following code snippet deallocates the instance on the right by assigning `tod2` to `tod3`:

```
tod3 = tod2;
```

This results in the situation shown in Figure 6.7.

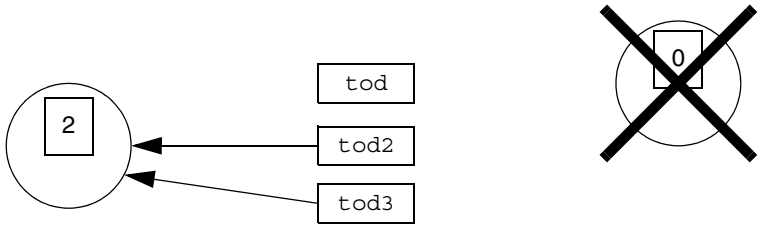


Figure 6.7. Deallocation of an instance with a reference count of zero.

Copying and Assignment of Classes

Classes have a default memberwise copy constructor and assignment operator, so you can copy and assign class instances:

```
TimeOfDayPtr tod = new TimeOfDayI(2, 3, 4); // Create instance
TimeOfDayPtr tod2 = new TimeOfDayI(*tod);   // Copy instance

TimeOfDayPtr tod3 = new TimeOfDayI;
*tod3 = *tod;                               // Assign instance
```

Copying and assignment in this manner performs a memberwise shallow copy or assignment, that is, the source members are copied into the target members; if a class contains class members (which are mapped as smart pointers), what is copied or assigned is the smart pointer, not the target of the smart pointer.

To illustrate this, consider the following Slice definitions:

```
class Node {
    int i;
    Node next;
};
```

Assume that we initialize two instances of type `Node` as follows:

```
NodePtr p1 = new Node(99, new Node(48, 0));
NodePtr p2 = new Node(23, 0);

// ...

*p2 = *p1; // Assignment
```

After executing the first two statements, we have the situation shown in Figure 6.8.

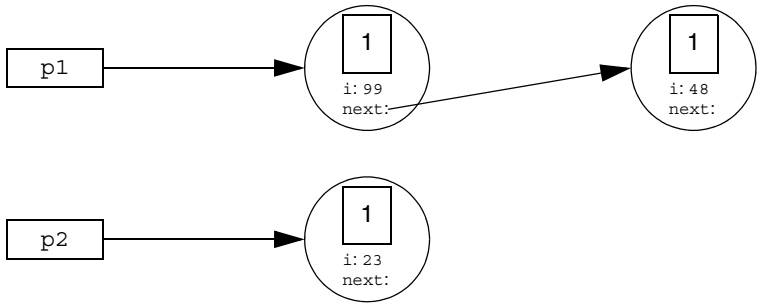


Figure 6.8. Class instances prior to assignment.

After executing the assignment statement, we end up with the result shown in Figure 6.9.

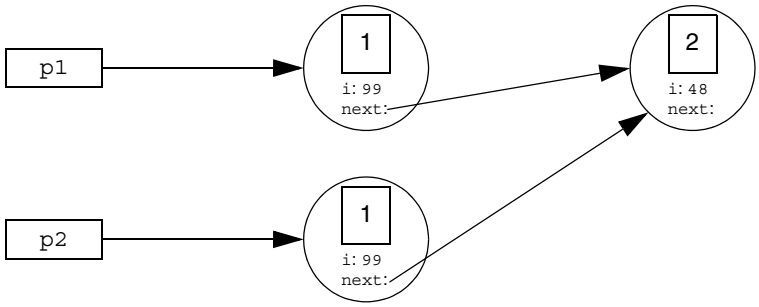


Figure 6.9. Class instances after assignment.

Note that copying and assignment also works for the implementation of abstract classes, such as our `TimeOfDayI` class, for example:

```
class TimeOfDayI;

typedef IceUtil::Handle<TimeOfDayI> TimeOfDayIPtr;

class TimeOfDayI : virtual public TimeOfDay {
    // As before...
};
```

The default copy constructor and assignment operator will perform a memberwise copy or assignment of your implementation class:

```
TimeOfDayIPtr tod1 = new TimeOfDayI;
TimeOfDayIPtr tod2 = new TimeOfDayI(*tod1);     // Make copy
```

Of course, if your implementation class contains raw pointers (for which a memberwise copy would almost certainly be inappropriate), you must implement your own copy constructor and assignment operator that take the appropriate action (and probably call the base copy constructor or assignment operator to take care of the base part).

Note that the preceding code uses `TimeOfDayIPtr` as a typedef for `IceUtil::Handle<TimeOfDayI>`. This class is a template that contains the smart pointer implementation. If you want smart pointers for the implementation of an abstract class, you must define a smart pointer type as illustrated by this type definition.

Copying and assignment of classes also works correctly for derived classes: you can assign a derived class to a base class, but not vice-versa; during such an assignment, the derived part of the source class is sliced, as per the usual C++ assignment semantics.

Polymorphic Copying of Classes

As shown in Section 6.14.1 on page 222, the C++ mapping generates an `ice_clone` member function for every class:

```
class TimeOfDay : virtual public Ice::Object {
public:
    // ...

    virtual Ice::ObjectPtr ice_clone() const;
};
```

This member function makes a polymorphic shallow copy of a class: members that are not class members are deep copied; all members that are class members are shallow copied. To illustrate, consider the following class definition:

```
class Node {
    Node n1;
    Node n2;
};
```

Assume that we have an instance of this class, with the `n1` and `n2` members initialized to point at separate instances, as shown in Figure 6.10.

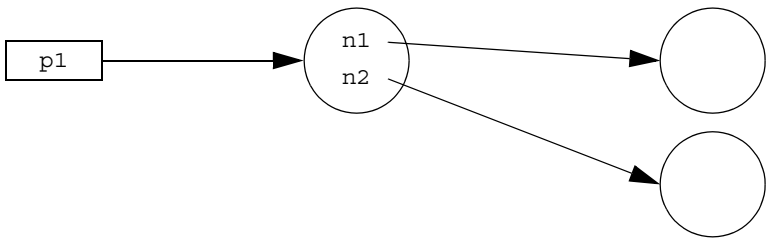


Figure 6.10. A class instance pointing at two other instances.

If we call `ice_clone` on the instance on the left, we end up with the situation shown in Figure 6.11.

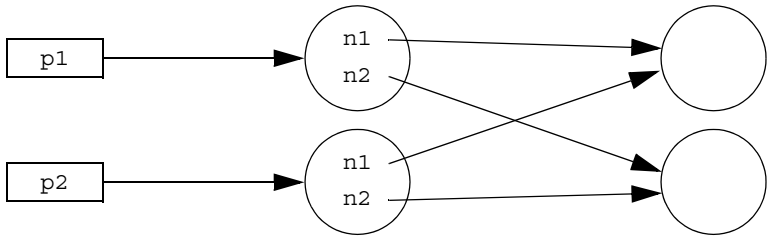


Figure 6.11. Resulting graph after calling `ice_clone` on the left-most instance of Figure 6.10.

As you can see, class members are shallow copied, that is, `ice_clone` makes a copy of the class instance on which it is invoked, but does not copy any class instances that are pointed at by the copied instance.

Note that `ice_clone` returns a value of type `Ice::ObjectPtr`, to avoid problems with compilers that do not support covariant return types. The generated `Ptr` classes contain a `dynamicCast` member that allows you to safely downcast the return value of `ice_clone`. For example, the code to achieve the situation shown in Figure 6.11 looks as follows:

```
NodePtr p1 = new Node(new Node, new Node);
NodePtr p2 = NodePtr::dynamicCast(p1->ice_clone());
```

`ice_clone` is generated by the Slice compiler for concrete classes (that is, classes that do not have operations). However, because classes with operations are abstract, for abstract classes, the generated `ice_clone` cannot know how to instantiate an instance of the derived concrete class (because the name of the derived class is not known). This means that, for abstract classes, the generated `ice_clone` throws a `CloneNotImplementedException`.

If you want to clone the implementation of an abstract class, you must override the virtual `ice_clone` member in your concrete implementation class. For example:

```
class TimeOfDayI : public TimeOfDay {
public:
    virtual Ice::ObjectPtr ice_clone() const
    {
        return new TimeOfDayI(*this);
    }
};
```

Null Smart Pointers

A null smart pointer contains a null C++ pointer to its underlying instance. This means that if you attempt to dereference a null smart pointer, you get an `IceUtil::NullHandleException`:

```
TimeOfDayPtr tod;               // Construct null handle

try {
    tod->minute = 0;            // Dereference null handle
    assert(false);              // Cannot get here
} catch (const IceUtil::NullHandleException&) {
    ; // OK, expected
} catch (...) {
    assert(false);              // Must get NullHandleException
}
```

Preventing Stack-Allocation of Class Instances

The Ice C++ mapping expects class instances to be allocated on the heap. Allocating class instances on the stack or in static variables is pragmatically useless because all the Ice APIs expect parameters that are smart pointers, not class instances. This means that, to do anything with a stack-allocated class instance, you must initialize a smart pointer for the instance. However, doing so does not work because it inevitably leads to a crash:

```
{                               // Enter scope
    TimeOfDayI t;               // Stack-allocated class instance
    TimeOfDayPtr todp;          // Handle for a TimeOfDay instance

    todp = &t;                  // Legal, but dangerous
    // ...
}                               // Leave scope, looming crash!
```

This goes badly wrong because, when `todp` goes out of scope, it decrements the reference count of the class to zero, which then calls `delete` on itself. However, the instance is stack-allocated and cannot be deleted, and we end up with undefined behavior (typically, a core dump).

The following attempt to fix this is also doomed to failure:

```
{                               // Enter scope
    TimeOfDayI t;               // Stack-allocated class instance
    TimeOfDayPtr todp;          // Handle for a TimeOfDay instance
```

```
    todp = &t;                  // Legal, but dangerous
    // ...
    todp = 0;                   // Crash imminent!
}
```

This code attempts to circumvent the problem by clearing the smart pointer explicitly. However, doing so also causes the smart pointer to drop the reference count on the class to zero, so this code ends up with the same call to `delete` on the stack-allocated instance as the previous example.

The upshot of all this is: *never* allocate a class instance on the stack or in a static variable. The C++ mapping assumes that all class instances are allocated on the heap and no amount of coding trickery will change this.[7]

You can prevent allocation of class instances on the stack or in static variables by adding a protected destructor to your implementation of the class: if a class has a protected destructor, allocation must be made with `new`, and static or stack allocation causes a compile time error. In addition, explicit calls to `delete` on a heap-allocated instance also are flagged at compile time. You may want to habitually add a protected destructor to your implementation of abstract Slice classes to protect yourself from accidental heap allocation, as shown on page 226. (For Slice classes that do not have operations, the Slice compiler automatically adds a protected destructor.)

Smart Pointers and Constructors

Slice classes inherit their reference-counting behavior from the `IceUtil::Shared` class (see Appendix F), which ensures that reference counts are managed in a thread-safe manner. When a stack-allocated smart pointer goes out of scope, the smart pointer invokes the `__decRef` function on the reference-counted object. Ignoring thread-safety issues, `__decRef` is implemented like this:

```
void
IceUtil::Shared::__decRef()
{
    if (--_ref == 0 && !_noDelete)
        delete this;
}
```

7. You could abuse the `__setNoDelete` member to disable deallocation, but we strongly discourage you from doing this.

In other words, when the smart pointer calls `__decRef` on the pointed-at instance and the reference count reaches zero (which happens when the last smart pointer for a class instance goes out of scope), the instance self-destructs by calling `delete this`.

However, as you can see, the instance self-destructs only if `_noDelete` is false (which it is by default, because the constructor initializes it to false). You can call `__setNoDelete(true)` to prevent this self-destruction and, later, call `__setNoDelete(false)` to enable it again. This is necessary if, for example, a class in its constructor needs to pass `this` to another function:

```
void someFunction(const TimeOfDayPtr& t)
{
    // ...
}

TimeOfDayI::TimeOfDayI()
{
    someFunction(this); // Trouble looming here!
}
```

At first glance, this code looks innocuous enough. While `TimeOfDayI` is being constructed, it passes `this` to `someFunction`, which expects a smart pointer. The compiler constructs a temporary smart pointer at the point of call (because the smart pointer template has a single-argument constructor that accepts a pointer to a heap-allocated instance, so the constructor acts a conversion function). However, this code fails badly. The `TimeOfDayI` instance is constructed with a statement such as:

```
TimeOfDayPtr tod = new TimeOfDayI;
```

The constructor of `TimeOfDayI` is called by `operator new` and, when the constructor starts executing, the reference count of the instance is zero (because that is what the reference count is initialized to by the `Shared` base class of `TimeOfDayI`). When the constructor calls `someFunction`, the compiler creates a temporary smart pointer, which increments the reference count of the instance and, once `someFunction` completes, the compiler dutifully destroys that temporary smart pointer again. But, of course, that drops the reference count back to zero and causes the `TimeOfDayI` instance to self-destruct by calling `delete this`. The net effect is that the call to `new TimeOfDayI` returns a pointer to an already deleted object, which is likely to cause the program to crash.

To get around the problem, you can call `__setNoDelete`:

```
TimeOfDayI::TimeOfDayI()
{
    __setNoDelete(true);
    someFunction(this);
    __setNoDelete(false);
}
```

The code disables self-destruction while `someFunction` uses its temporary smart pointer by calling `__setNoDelete(true)`. By doing this, the reference count of the instance is incremented before `someFunction` is called and decremented back to zero when `someFunction` completes without causing the object to self-destruct. The constructor then re-enables self-destruction by calling `__setNoDelete(false)` before returning, so the statement

```
TimeOfDayPtr tod = new TimeOfDayI;
```

does the usual thing, namely to increment the reference count of the object to 1, despite the fact that a temporary smart pointer existed while the constructor ran.

In general, whenever a class constructor passes `this` to a function or another class that accepts a smart pointer, you must temporarily disable self-destruction.

Smart Pointers and Exception Safety

Smart pointers are exception safe: if an exception causes the thread of execution to leave a scope containing a stack-allocated smart pointer, the C++ run time ensures that the smart pointer's destructor is called, so no resource leaks can occur:

```
{ // Enter scope...

    TimeOfDayPtr tod = new TimeOfDayI; // Allocate instance

    someFuncThatMightThrow();          // Might throw...

    // ...

} // No leak here, even if an exception is thrown
```

If an exception is thrown, the destructor of `tod` runs and ensures that it deallocates the underlying class instance.

There is one potential pitfall you must be aware of though: if a constructor of a base class throws an exception, and another class instance holds a smart pointer to the instance being constructed, you can end up with a double deallocation. You can use the `__setNoDelete` mechanism to temporarily disable self-destruction in this case, as described in the previous section.

Smart Pointers and Cycles

One thing you need to be aware of is the inability of reference counting to deal with cyclic dependencies. For example, consider the following simple self-referential class:

```
class Node {
    int val;
    Node next;
};
```

Intuitively, this class implements a linked list of nodes. As long as there are no cycles in the list of nodes, everything is fine, and our smart pointers will correctly deallocate the class instances. However, if we introduce a cycle, we have a problem:

```
{                           // Open scope...

    NodePtr n1 = new Node; // N1 refcount == 1
    NodePtr n2 = new Node; // N2 refcount == 1
    n1->next = n2;         // N2 refcount == 2
    n2->next = n1;         // N1 refcount == 2

} // Destructors run:      // N2 refcount == 1,
                           // N1 refcount == 1, memory leak!
```

The nodes pointed to by `n1` and `n2` do not have names but, for the sake of illustration, let us assume that `n1`'s node is called N1, and `n2`'s node is called N2. When we allocate the N1 instance and assign it to `n1`, the smart pointer `n1` increments N1's reference count to 1. Similarly, N2's reference count is 1 after allocating the node and assigning it to `n2`. The next two statements set up a cyclic dependency between `n1` and `n2` by making their `next` pointers point at each other. This sets the reference count of both N1 and N2 to 2. When the enclosing scope closes, the destructor of `n2` is called first and decrements N2's reference count to 1, followed by the destructor of `n1`, which decrements N1's reference count to 1. The net effect is that neither reference count ever drops to zero, so both N1 and N2 are leaked.

Garbage Collection of Class Instances

The previous example illustrates a problem that is generic to using reference counts for deallocation: if a cyclic dependency exists anywhere in a graph (possibly via many intermediate nodes), all nodes in the cycle are leaked.

To avoid memory leaks due to such cycles, Ice for C++ contains a garbage collector. The collector identifies class instances that are part of one or more cycles but are no longer reachable from the program and deletes such instances:

- By default, garbage is collected whenever you destroy a communicator. This means that no memory is leaked when your program exits. (Of course, this assumes that you correctly destroy your communicators as described in Section 8.3.)
- You can also explicitly call the garbage collector by calling `Ice::collectGarbage`. For example, the leak caused by the preceding example can be avoided as follows:

```
{                           // Open scope...

    NodePtr n1 = new Node; // N1 refcount == 1
    NodePtr n2 = new Node; // N2 refcount == 1
    n1->next = n2;         // N1 refcount == 2
    n2->next = n1;         // N2 refcount == 2

} // Destructors run:      // N2 refcount == 1,
                           // N1 refcount == 1

Ice::collectGarbage();     // Deletes N1 and N2
```

  The call to `Ice::collectGarbage` deletes the no longer reachable instances N1 and N2 (as well as any other non-reachable instances that may have accumulated earlier).
- Deleting leaked memory with explicit calls to the garbage collector can be inconvenient because it requires polluting the code with calls to the collector. You can ask the Ice run time to run a garbage collection thread that periodically cleans up leaked memory by setting the property `Ice.GC.Interval` to a non-zero value.[8] For example, setting `Ice.GC.Interval` to 5 causes the collector thread to run the garbage collector once every five seconds. You can trace the execution of the collector by setting `Ice.Trace.GC` to a non-zero value (Appendix D).

Note that the garbage collector is useful *only* if your program actually creates cyclic class graphs. There is no point in running the garbage collector in programs that do not create such cycles. (For this reason, the collector thread is disabled by

8. See Chapter 30 for how to set properties.

default and runs only if you explicitly set `Ice.GC.Interval` to a non-zero value.)

Smart Pointer Comparison

As for proxy handles (see Section 6.11.4 on page 210), class handles support the comparison operators `==`, `!=`, and `<`. This allows you to use class handles in STL sorted containers. Be aware that, for smart pointers, object identity is *not* used for the comparison, because class instances do not have identity. Instead, these operators simply compare the memory address of the classes they point to. This means that `operator==` returns true only if two smart pointers point at the same physical class instance:

```
// Create a class instance and initialize
//
TimeOfDayIPtr p1 = new TimeOfDayI;
p1->hour = 23;
p1->minute = 10;
p1->second = 18;

// Create another class instance with
// the same member values
//
TimeOfDayIPtr p2 = new TimeOfDayI;
p2->hour = 23;
p2->minute = 10;
p2->second = 18;

assert(p1 != p2);       // The two do not compare equal

TimeOfDayIPtr p3 = p1;  // Point at first class again

assert(p1 == p3);       // Now they compare equal
```

6.15 Asynchronous Method Invocation (AMI)

NOTE: As of version 3.4, Ice provides a new API for asynchronous method invocation. This section describes this API. You can find documentation for the previous API in Appendix K. Note that the old API is deprecated and will be removed in a future release.

Asynchronous Method Invocation (*AMI*) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

6.15.1 Basic Asynchronous API

Consider the following simple Slice definition:

```
module Demo {
    interface Employees {
        string getName(int number);
    };
};
```

Proxy Methods

Besides the synchronous proxy methods, **`slice2cpp`** generates the following asynchronous proxy methods:[9]

```
Ice::AsyncResultPtr begin_getName(Ice::Int number);
Ice::AsyncResultPtr begin_getName(Ice::Int number,
                                  const Ice::Context& __ctx)

std::string end_getName(const Ice::AsyncResultPtr&);
```

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. (The `begin_` method is overloaded so you can pass a per-invocation context—see Section 32.12.)

- The `begin_getName` method sends (or queues) an invocation of `getName`. This method does not block the calling thread.

9. There are four additional overloads of `begin_getName` that we discuss in Sections 6.15.4 and 6.15.5.

- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

```
EmployeesPrx e = ...;
Ice::AsyncResultPtr r = e->begin_getName(99);

// Continue to do other things here...

string name = e->end_getName(r);
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `AsyncResultPtr`. The `AsyncResult` associated with this smart pointer contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResultPtr` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. Similarly, the `end_` method has one out-parameter for each out-parameter of the corresponding Slice operation (plus the `AsyncResultPtr` parameter). For example, consider the following operation:

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

```
Ice::AsyncResultPtr begin_op(Ice::Int inp1,
                             const ::std::string& inp2)

Ice::Double end_op(bool& outp1, Ice::Long& outp2,
                   const Ice::AsyncResultPtr&);
```

Exception Handling

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` method is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `CommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `IceUtil::IllegalArgumentException`. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_` method, or if the `AsyncResult` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

6.15.2 The `AsyncResult` Class

The `AsyncResult` that is returned by the `begin_` method encapsulates the state of the asynchronous invocation:

```
class AsyncResult
    : virtual public IceUtil::Shared,
      private IceUtil::noncopyable {
public:
    virtual bool operator==(const AsyncResult&) const;
    virtual bool operator<(const AsyncResult&) const;

    virtual Int getHash() const;

    virtual CommunicatorPtr getCommunicator() const;
    virtual ConnectionPtr getConnection() const;
    virtual ObjectPrx getProxy() const;
    const string& getOperation() const;
    LocalObjectPtr getCookie() const;

    bool isCompleted() const;
    void waitForCompleted();

    bool isSent() const;
    void waitForSent();

    bool sentSynchronously() const;

};
```

The methods have the following semantics:

- `bool operator==(const AsyncResult&) const`
  `bool operator<(const AsyncResult&) const`
  `Int getHash() const`

  These methods allow you to create ordered or hashed collections of pending asynchronous invocations. This is useful, for example, if you can have a number of outstanding requests, and need to pass state between the `begin_` and the `end_` methods. In this case, you can use the returned `AsyncResults` as the key into a map that stores the state for each call.

- `CommunicatorPtr getCommunicator() const`

  This method returns the communicator that sent the invocation.

- `virtual ConnectionPtr getConnection() const`

  This method returns the connection that was used for the invocation.

- `virtual ObjectPrx getProxy() const`

  This method returns the proxy that was used to call the `begin_` method.

- `const string& getOperation() const`

  This method returns the name of the operation.

- `LocalObjectPtr getCookie() const`

  This method returns the cookie that was passed to the `begin_` method (see Section 6.15.4). If you did not pass a cookie to the `begin_` method, the return value is null.

- `bool isCompleted() const`

  This method returns true if, at the time it is called, the result of an invocation is available, indicating that a call to the `end_` method will not block the caller. Otherwise, if the result is not yet available, the method returns false.

- `void waitForCompleted()`

  This method blocks the caller until the result of an invocation becomes available.

- `bool isSent() const`

  When you call the `begin_` method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to

the local transport (whether it was initially queued or not). Otherwise, if the request is still queued, `isSent` returns false.

- `void waitForSent()`

  This method blocks the calling thread until a request has been written to the client-side transport.

- `bool sentSynchronously() const`

  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

6.15.3 Polling for Completion

The `AsyncResult` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
};
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;

Ice::Int offset = 0;
while (!file.eof()) {
    ByteSeq bs;
    bs = file.read(chunkSize); // Read a chunk
    ft->send(offset, bs);      // Send the chunk
    offset += bs.size();
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing—the client does nothing

while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
Ice::Int offset = 0;

list<Ice::AsyncResultPtr> results;
const int numRequests = 5;

while (!file.eof()) {
    ByteSeq bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    Ice::AsyncResultPtr r = ft->begin_send(offset, bs);
    offset += bs.size();

    // Wait until this request has been passed to the transport.
    r->waitForSent();
    results.push_back(r);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while (results.size() > numRequests) {
        Ice::AsyncResultPtr r = results.front();
        results.pop_front();
        r->waitForCompleted();
    }
}

// Wait for any remaining requests to complete.
while (!results.empty()) {
    Ice::AsyncResultPtr r = results.front();
    results.pop_front();
    r->waitForCompleted();
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to “keep the pipe

to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

6.15.4 Generic Completion Callbacks

The `begin_` method is overloaded to allow you to provide completion callbacks. Here are the corresponding methods for the `getName` operation:

```
Ice::AsyncResultPtr begin_getName(
                        Ice::Int number,
                        const Ice::CallbackPtr& __del,
                        const Ice::LocalObjectPtr& __cookie = 0);

Ice::AsyncResultPtr begin_getName(
                        Ice::Int number,
                        const Ice::Context& __ctx,
                        const Ice::CallbackPtr& __del,
                        const Ice::LocalObjectPtr& __cookie = 0);
```

The second version of `begin_getName` lets you override the default context. (We discuss the purpose of the `cookie` parameter in the next section.) Following the in-parameters, the `begin_` method accepts a parameter of type `Ice::CallbackPtr`. This is a smart pointer to a callback class that is provided by the Ice run time. This class stores an instance of a callback class that you implement. The Ice run time invokes a method on your callback instance when an asynchronous operation completes. Your callback class must provide a method that returns `void` and accepts a single parameter of type `const AsyncResultPtr&`, for example:

```
class MyCallback : public IceUtil::Shared {
public:
    void finished(const Ice::AsyncResultPtr& r) {
        EmployeesPrx e =
            EmployeesPrx::uncheckedCast(r->getProxy());
        try {
            string name = e->end_getName(r);
```

```
            cout << "Name is: " << name << endl;
        } catch (const Ice::Exception& ex) {
            cerr << "Exception is: " << ex << endl;
        }
    }
};
typedef IceUtil::Handle<MyCallback> MyCallbackPtr;
```

Note that your callback class must derive from `IceUtil::Shared`. The callback method can have any name you prefer but its signature must match the preceding example.

The implementation of your callback method must call the `end_` method. The proxy for the call is available via the `getProxy` method on the `AsyncResult` that is passed by the Ice run time. The return type of `getProxy` is `Ice::ObjectPrx`, so you must down-cast the proxy to its correct type. (You should always use an `uncheckedCast` to do this, otherwise you send an additional message to the server to verify the proxy type.)

Your callback method should catch and handle any exceptions that may be thrown by the `end_` method. If you allow an exception to escape from the callback method, the Ice run time produces a log entry by default and ignores the exception. (You can disable the log message by setting the property `Ice.Warn.AMICallback` to zero.)

To inform the Ice run time that you want to receive a callback for the completion of the asynchronous call, you pass the callback instance to the `begin_` method:

```
EmployeesPrx e = ...;

MyCallbackPtr cb = new MyCallback;
Ice::CallbackPtr d = Ice::newCallback(cb, &MyCallback::finished);

e->begin_getName(99, d);
```

Note the call to `Ice::newCallback` in this example. This helper function expects a smart pointer to your callback instance and a member function pointer that specifies your callback method.

Using Cookies

It is common for the `end_` method to require access to some state that is established by the code that calls the `begin_` method. As an example, consider an application that asynchronously starts a number of operations and, as each operation completes, needs to update different user interface elements with the results.

In this case, the `begin_` method knows which user interface element should receive the update, and the `end_` method needs access to that element.

The API allows you to pass such state by providing a cookie. A cookie is an instance of a class that you write; the class can contain whatever data you want to pass, as well as any methods you may want to add to manipulate that data.

The only requirement on the cookie class is that it must derive from `Ice::LocalObject`. Here is an example implementation that stores a `WidgetHandle`. (We assume that this class provides whatever methods are needed by the `end_` method to update the display.)

```
class Cookie : public Ice::LocalObject
{
public:
    Cookie(WidgetHandle h) : _h(h) {}
    WidgetHandle getWidget() { return _h; }

private:
    WidgetHandle _h;
};
typedef IceUtil::Handle<Cookie> CookiePtr;
```

When you call the `begin_` method, you pass the appropriate cookie instance to inform the `end_` method how to update the display:

```
// Make cookie for call to getName(99).
CookiePtr cookie1 = new Cookie(widgetHandle1);

// Make cookie for call to getName(42);
CookiePtr cookie2 = new Cookie(widgetHandle2);

// Invoke the getName operation with different cookies.
e->begin_getName(99, getNameCB, cookie1);
e->begin_getName(24, getNameCB, cookie2);
```

The `end_` method can retrieve the cookie from the `AsyncResult` by calling `getCookie`. For this example, we assume that widgets have a `writeString` method that updates the relevant UI element:

```
void
MyCallback::getName(const Ice::AsyncResultPtr& r)
{
    EmployeesPrx e = EmployeesPrx::uncheckedCast(r->getProxy());
    CookiePtr cookie = CookiePtr::dynamicCast(r->getCookie());
    try {
        string name = e->end_getName(r);
        cookie->getWidget()->writeString(name);
```

```
    } catch (const Ice::Exception& ex) {
        handleException(ex);
    }
}
```

The cookie provides a simple and effective way for you to pass state between the point where an operation is invoked and the point where its results are processed. Moreover, if you have a number of operations that share common state, you can pass the same cookie instance to multiple invocations.

6.15.5 Type-Safe Completion Callbacks

The generic callback API we saw in Section 6.15.4 is not entirely type-safe:

- You must down-cast the return value of `getProxy` to the correct proxy type before you can call the `end_` method.
- You must call the correct `end_` method to match the operation called by the `begin_` method.
- If you use a cookie, you must down-cast the cookie to the correct type before you can access the data inside the cookie.
- You must remember to catch exceptions when you call the `end_` method; if you forget to do this, you will not know that the operation failed.

`slice2cpp` generates an additional type-safe API that takes care of these chores for you. The type-safe API is provided as a template that works much like the `Ice::Callback` class of the generic API, but requires strongly-typed method signatures.

To use type-safe callbacks, you must implement a callback class that provides two callback methods:

- a success callback that is called if the operation succeeds
- a failure callback that is called if the operation raises an exception

As for the generic API, your callback class must derive from `IceUtil::Shared`. Here is a callback class for an invocation of the `getName` operation:

```
class MyCallback : public IceUtil::Shared
{
public:
    void getNameCB(const string& name) {
        cout << "Name is: " << name << endl;
    }
```

```
    void failureCB(const Ice::Exception& ex) {
        cerr << "Exception is: << ex << endl;
    }
};
```

The callback methods can have any name you prefer and must have `void` return type. The failure callback always has a single parameter of type `const Ice::Exception&`. The success callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the success callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

To receive these callbacks, you instantiate your callback instance and specify the methods you have defined before passing a smart pointer to a callback wrapper instance to the `begin_` method:

```
MyCallbackPtr cb = new MyCallback;

Callback_Employees_getNamePtr getNameCB =
    newCallback_Employees_getName(
        cb, &MyCallback::getNameCB, &MyCallback::failureCB);

Callback_Employees_getNumberPtr getNumberCB =
    newCallback_Employees_getNumber(
        cb, &MyCallback::getNumberCB, &MyCallback::failureCB);

e->begin_getName(99, getNameCB);
e->begin_getNumber("Fred", getNumberCB);
```

Note how this code creates instances of two smart pointer types generated by **`slice2cpp`** named `Callback_Employees_getNamePtr` and `Callback_Employees_getNumberPtr`. Each smart pointer points to a template instance that encapsulates your callback instance and two member function pointers for the callback methods. The name of this smart pointer type is `<module>::Callback_<interface>_<operation>Ptr`.

Also note that the code uses helper functions to initialize the smart pointers. The first argument to the helper function is your callback instance, and the two following arguments are the success and failure member function pointers, respectively. The name of this helper function is `<module>::newCallback_<interface>_<operation>`.

It is legal to pass a null pointer as the success or failure callback. For the success callback, this is legal only for operations that have `void` return type and no

out-parameters. This is useful if you do not care when the operation completes but want to know if the call failed. If you pass a null exception callback, the Ice run time will ignore any exception that is raised by the invocation.

The type of the success and exception member function pointers is provided as `Response` and `Exception` typedefs by the callback template. For example, you can ignore exceptions for an invocation of `getName` as follows:

```
Callback_Employees_op::Exception nullException = 0;

MyCallbackPtr cb = new MyCallback;

Callback_Employees_getNamePtr getNameCB =
    newCallback_Employees_getName(
        cb, &MyCallback::getNameCB, nullException);

e->begin_getName(99, getNameCB); // Ignores exceptions
```

Using Cookies

The `begin_` method optionally accepts a cookie as a trailing parameter. As for the generic API, you can use the cookie to share state between the `begin_` and `end_` methods. However, with the type-safe API, there is no need to down-cast the cookie. Instead, the cookie parameter that is passed to the `end_` method is strongly typed. Assuming that you have defined a `Cookie` class and `CookiePtr` smart pointer, you can pass a cookie to the `begin_` method as follows:

```
MyCallbackPtr cb = new MyCallback;

Callback_Employees_getNamePtr getNameCB =
    newCallback_Employees_getName(
        cb, &MyCallback::getNameCB, &MyCallback::failureCB);

CookiePtr cookie = new Cookie(widgetHandle);
e->begin_getName(99, getNameCB, cookie);
```

The callback methods of your callback class simply add the cookie parameter:

```
class MyCallback : public IceUtil::Shared
{
public:
    void getNameCB(const string& name, const CookiePtr& cookie) {
        cookie->getWidget()->writeString(name);
    }

    void failureCB(const Ice::Exception& ex,
```

```
                    const CookiePtr& cookie) {
            cookie->getWidget()->writeError(ex.what());
        }
};
```

6.15.6 Oneway Invocations

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws an `IceUtil::IllegalArgumentException`.

For the generic API, the callback method looks exactly as for a twoway invocation. However, for oneway invocations, the Ice run time does not call the callback method unless the invocation raised an exception during the `begin_` method ("on the way out").

For the type-safe API, the `newCallback` helper for `void` operations is overloaded so you can omit the success callback. For example, here is how you could call `ice_ping` asynchronously:

```
MyCallbackPtr cb = new MyCallback;

Ice::Callback_Object_ice_pingPtr callback =
    Ice::newCallback_Object_ice_ping(cb, &MyCallback::failureCB);

p->begin_opVoid(callback);
```

6.15.7 Flow Control

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `AsyncResult::sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `AsyncResult::sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the

client stops invoking more operations until some of the queued operations have drained out of the local transport.

For the generic API, you can create an additional callback method:

```
class MyCallback : public IceUtil::Shared {
public:
    void finished(const Ice::AsyncResultPtr&);
    void sent(const Ice::AsyncResultPtr&);
};
typedef IceUtil::Handle<MyCallback> MyCallbackPtr;
```

As with any other callback method, you are free to choose any name you like. For this example, the name of the callback method is `sent`. You inform the Ice run time that you want to be informed when a call has been passed to the local transport by specifying the `sent` method as an additional parameter when you create the `Ice::Callback`:

```
EmployeesPrx e = ...;

MyCallbackPtr cb = new MyCallback;
Ice::CallbackPtr d = Ice::newCallback(cb,
                                      &MyCallback::finished,
                                      &MyCallback::sent);

e->begin_getName(99, d);
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the `sent` method from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the `sent` method from a different thread once it has written the request to the local transport. In addition, you can find out from the `AsyncResult` that is returned by the `begin_` method whether the request was sent synchronously or was queued, by calling `sentSynchronously`.

For the generic API, the `sent` method has the following signature:

```
void sent(const Ice::AsyncResult&);
```

For the type-safe API, there are two versions, one without and one with a cookie:

```
void sent(bool sentSynchronously);
void sent(bool sentSynchronously, const <CookiePtr>& cookie);
```

For the version with a cookie, *`<CookiePtr>`* is replaced with the actual type of the cookie smart pointer you passed to the `begin_` method.

The `sent` methods allow you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

6.15.8 Batch Requests

Applications that send batched requests (see Section 32.16) can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `AsyncResult::getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

6.15.9 Concurrency

The Ice run time always invokes your callback methods from a separate thread. This means that you can safely use a non-recursive mutex without risking deadlock. There is one exception to this rule: the run time calls the `sent` callback from the thread calling the `begin_` method if the request could be sent synchronously. In the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` member or parameter, so you can take appropriate action to avoid a self-deadlock.

6.15.10 Limitations

AMI invocations cannot be sent using collocated optimization. If you attempt to invoke an AMI operation using a proxy that is configured to use collocation optimization, the Ice run time raises `CollocationOptimizationException` if the servant happens to be collocated; the request is sent normally if the servant is not collocated. Section 32.21 provides more information about this optimization and describes how to disable it when necessary.

6.16 `slice2cpp` Command-Line Options

The Slice-to-C++ compiler, **`slice2cpp`**, offers the following command-line options in addition to the standard options described in Section 4.20:

- **`--header-ext`** ***`EXT`***

  Changes the file extension for the generated header files from the default `h` to the extension specified by ***`EXT`***.

  You can also change the header file extension with a global metadata directive:

  ```
  [["cpp:header-ext:hpp"]]

  // ...
  ```

  Only one such directive can appear in each source file. If you specify a header extension on both the command line and with a metadata directive, the metadata directive takes precedence. This ensures that included Slice files that were compiled separately get the correct header extension (provided that the included Slice files contain a corresponding metadata directive). For example:

  ```
  // File example.ice
  #include <Ice/BuiltinSequences.ice>

  // ...
  ```

  Compiling this file with

  ```
  $ slice2cpp --header-ext=hpp -I/opt/Ice/include \
  example.ice
  ```

  generates `example.hpp`, but the `#include` directive in that file is for `Ice/BuiltinSequences.h` (not `Ice/BuiltinSequences.hpp`) because `BuiltinSequences.ice` contains a metadata directive `[["cpp:header-ext:h"]]`.

  You normally will not need to use this metadata directive. The directive is necessary only if:

  - You `#include` a Slice file in one of your own Slice files.
  - The included Slice file is part of a library you link against.
  - The library ships with the included Slice file's header.

- The library header uses a different header extension than your own code.

For example, if the library uses `.hpp` as the header extension, but your own code uses `.h`, the library's Slice file should contain a `[["cpp:header-ext:hpp"]]` directive. (If the directive is missing, you can add it to the library's Slice file.)

- **`--source-ext` *`EXT`***

Changes the file extension for the generated source files from the default `cpp` to the extension specified by ***`EXT`***.

- **`--add-header` *`HDR`*`[,`*`GUARD`*`]`**

This option adds an include directive for the specified header at the beginning of the generated source file (preceding any other include directives). If ***`GUARD`*** is specified, the include directive is protected by the specified guard. For example, **`--add-header precompiled.h,__PRECOMPILED_H__`** results in the following directives at the beginning of the generated source file:

```
#ifndef __PRECOMPILED_H__
#define __PRECOMPILED_H__
#include <precompiled.h>
#endif
```

The option can be repeated to create include directives for several files.

As suggested by the preceding example, this option is useful mainly to integrate the generated code with a compiler's precompiled header mechanism.

- **`--include-dir` *`DIR`***

Modifies `#include` directives in source files to prepend the path name of each header file with the directory ***`DIR`***. See Section 6.16.1 for more information.

- **`--impl`**

Generate sample implementation files. This option will not overwrite an existing file.

- **`--depend`**

Prints makefile dependency information to standard output. No code is generated when this option is specified. The output generally needs to be filtered before it can be included in a makefile; the Ice build system uses the script `config/makedepend.py` for this purpose.

- **`--dll-export` *`SYMBOL`***

  Use ***SYMBOL*** to control DLL exports or imports. This option allows you to selectively export or import global symbols in the generated code. As an example, compiling a Slice definition with

  ```
  $ slice2cpp --dll-export ENABLE_DLL x.ice
  ```

  results in the following additional code being generated into `x.h`:

  ```
  #ifndef ENABLE_DLL
  #   ifdef ENABLE_DLL_EXPORTS
  #       define ENABLE_DLL ICE_DECLSPEC_EXPORT
  #   else
  #       define ENABLE_DLL ICE_DECLSPEC_IMPORT
  #   endif
  #endif
  ```

  `ICE_DECLSPEC_EXPORT` and `ICE_DECLSPEC_IMPORT` are platform-specific macros. For example, for Windows, they are defined as `__declspec(dllexport)` and `__declspec(dllimport)`, respectively; for Solaris using **CC** version 5.5 or later, `ICE_DECLSPEC_EXPORT` is defined as `__global`, and `ICE_DECLSPEC_IMPORT` is empty.[10]

  The symbol name you specify on the command line (`ENABLE_DLL` in this example) is used by the generated code to export or import any symbols that must be visible to code outside the generated compilation unit. The net effect is that, if you want to create a DLL that includes `x.cpp`, but also want to use the generated types in compilation units outside the DLL, you can arrange for the relevant symbols to be exported by compiling `x.cpp` with **`-DENABLE_DLL_EXPORTS`**.

- **`--checksum`**

  Generate checksums for Slice definitions.

- **`--stream`**

  Generate streaming helper functions for Slice types (see Section 35.2).

10. Similar definitions exist for other platforms. For platforms that do not have any concept of explicit export or import of shared library symbols, both macros are empty.

6.16.1 Include Directives

The #include directives generated by the Slice-to-C++ compiler can be a source of confusion if the semantics governing their generation are not well-understood. The generation of #include directives is influenced by the command-line options **-I** and **--include-dir**; these options are discussed in more detail below. The **--output-dir** option directs the translator to place all generated files in a particular directory, but has no impact on the contents of the generated code.

Given that the #include directives in header files and source files are generated using different semantics, we describe them in separate sections.

Header Files

In most cases, the compiler generates the appropriate #include directives by default. As an example, suppose file A.ice includes B.ice using the following statement:

```
// A.ice
#include <B.ice>
```

Assuming both files are in the current working directory, we run the compiler as shown below:

```
$ slice2cpp -I. A.ice
```

The generated file A.h contains this #include directive:

```
// A.h
#include <B.h>
```

If the proper include paths are specified to the C++ compiler, everything should compile correctly.

Similarly, consider the common case where A.ice includes B.ice from a subdirectory:

```
// A.ice
#include <inc/B.ice>
```

Assuming both files are in the inc subdirectory, we run the compiler as shown below:

```
$ slice2cpp -I. inc/A.ice
```

The default output of the compiler produces this #include directive in A.h:

```
// A.h
#include <inc/B.h>
```

Again, it is the user's responsibility to ensure that the C++ compiler is configured to find `inc/B.h` during compilation.

Now let us consider a more complex example, in which we do not want the `#include` directive in the header file to match that of the Slice file. This can be necessary when the organizational structure of the Slice files does not match the application's C++ code. In such a case, the user may need to relocate the generated files from the directory in which they were created, and the `#include` directives must be aligned with the new structure.

For example, let us assume that `B.ice` is located in the subdirectory `slice/inc`:

```
// A.ice
#include <slice/inc/B.ice>
```

However, we do not want the `slice` subdirectory to appear in the `#include` directive generated in the header file, therefore we specify the additional compiler option **`-Islice`**:

```
$ slice2cpp -I. -Islice slice/inc/A.ice
```

The generated code demonstrates the impact of this extra option:

```
// A.h
#include <inc/B.h>
```

As you can see, the `#include` directives generated in header files are affected by the include paths that you specify when running the compiler. Specifically, the include paths are used to abbreviate the path name in generated `#include` directives.

When translating an `#include` directive from a Slice file to a header file, the compiler compares each of the include paths against the path of the included file. If an include path matches the leading portion of the included file, the compiler removes that leading portion when generating the `#include` directive in the header file. If more than one include path matches, the compiler selects the one that results in the shortest path for the included file.

For example, suppose we had used the following options when compiling `A.ice`:

```
$ slice2cpp -I. -Islice -Islice/inc slice/inc/A.ice
```

In this case, the compiler compares all of the include paths against the included file `slice/inc/B.ice` and generates the following directive:

```
// A.h
#include <B.h>
```

The option **-Islice/inc** produces the shortest result, therefore the default path for the included file (slice/inc/B.h) is replaced with B.h.

In general, the **-I** option plays two roles: it enables the preprocessor to locate included Slice files, and it provides you with a certain amount of control over the generated #include directives. In the last example above, the preprocessor locates slice/inc/B.ice using the include path specified by the **-I.** option. The remaining **-I** options do not help the preprocessor locate included files; they are simply hints to the compiler.

Finally, we recommend using caution when specifying include paths. If the preprocessor is able to locate an included file via multiple include paths, it always uses the first include path that successfully locates the file. If you intend to modify the generated #include directives by specifying extra **-I** options, you must ensure that your include path hints match the include path selected by the preprocessor to locate the included file. As a general rule, you should avoid specifying include paths that enable the preprocessor to locate a file in multiple ways.

Source Files

By default, the compiler generates #include directives in source files using only the base name of the included file. This behavior is usually appropriate when the source file and header file reside in the same directory.

For example, suppose A.ice includes B.ice from a subdirectory, as shown in the following snippet of A.ice:

```
// A.ice
#include <inc/B.ice>
```

We generate the source file using this command:

```
$ slice2cpp -I. inc/A.ice
```

Upon examination, we see that the source file contains the following #include directive:

```
// A.cpp
#include <B.h>
```

However, suppose that we wish to enforce a particular standard for generated #include directives so that they are compatible with our C++ compiler's existing include path settings. In this case, we use the **--include-dir** option to modify the generated code. For example, consider the compiler command shown below:

```
$ slice2cpp --include-dir src -I. inc/A.ice
```

The source file now contains the following `#include` directive:

```
// A.cpp
#include <src/B.h>
```

Any leading path in the included file is discarded as usual, and the value of the **`--include-dir`** option is prepended.

6.17 Using Slice Checksums

As described in Section 4.21, the Slice compilers can optionally generate checksums of Slice definitions. For **`slice2cpp`**, the **`--checksum`** option causes the compiler to generate code in each C++ source file that accumulates checksums in a global map. A copy of this map can be obtained by calling a function defined in the header file `Ice/SliceChecksums.h`:

```
namespace Ice {
    Ice::SliceChecksumDict sliceChecksums();
}
```

In order to verify a server's checksums, a client could simply compare the dictionaries using the equality operator. However, this is not feasible if it is possible that the server might be linked with more Slice definitions than the client. A more general solution is to iterate over the local checksums as demonstrated below:

```
Ice::SliceChecksumDict serverChecksums = ...
Ice::SliceChecksumDict localChecksums = Ice::sliceChecksums();

for (Ice::SliceChecksumDict::const_iterator
     p = localChecksums.begin();
     p != localChecksums.end(); ++p) {

    Ice::SliceChecksumDict::const_iterator q
        = serverChecksums.find(p->first);
    if (q == serverChecksums.end()) {
        // No match found for type id!
    } else if (p->second != q->second) {
        // Checksum mismatch!
    }
}
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

Chapter 7
Developing a File System Client in C++

7.1 Chapter Overview

In this chapter, we present the source code for a C++ client that accesses the file system we developed in Chapter 5 (see Chapter 9 for the corresponding server).

7.2 The C++ Client

We now have seen enough of the client-side C++ mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
```

```
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

```
#include <Ice/Ice.h>
#include <Filesystem.h>
#include <iostream>
#include <iterator>

using namespace std;
using namespace Filesystem;

static void
listRecursive(const DirectoryPrx& dir, int depth = 0)
{
    // ...
}

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        // Create a communicator
        //
        ic = Ice::initialize(argc, argv);

        // Create a proxy for the root directory
        //
        Ice::ObjectPrx base
            = ic->stringToProxy("RootDir:default -p 10000");
```

```
        if (!base)
            throw "Could not create proxy";

        // Down-cast the proxy to a Directory proxy
        //
        DirectoryPrx rootDir = DirectoryPrx::checkedCast(base);
        if (!rootDir)
            throw "Invalid proxy";

        // Recursively list the contents of the root directory
        //
        cout << "Contents of root directory:" << endl;
        listRecursive(rootDir);
    } catch (const Ice::Exception& ex) {
        cerr << ex << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }

    // Clean up
    //
    if (ic)
        ic->destroy();

    return status;
}
```

1. The code includes a few header files:

   1. `Ice/Ice.h`

      This file is always included in both client and server source files. It provides definitions that are necessary for accessing the Ice run time.

   2. `Filesystem.h`

      This is the header that is generated by the Slice compiler from the Slice definitions in `Filesystem.ice`.

   3. `iostream`

      The client uses the iostream library to produce its output.

   4. `iterator`

      The implementation of `listRecursive` uses an STL iterator.

2. The code adds `using` declarations for the `std` and `Filesystem` namespaces.

3. The structure of the code in `main` follows what we saw in Chapter 3. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.

4. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`:

```
// Recursively print the contents of directory "dir" in
// tree fashion. For files, show the contents of each file.
// The "depth" parameter is the current nesting level
// (for indentation).

static void
listRecursive(const DirectoryPrx& dir, int depth = 0)
{
    string indent(++depth, '\t');

    NodeSeq contents = dir->list();

    for (NodeSeq::const_iterator i = contents.begin();
         i != contents.end();
         ++i) {
        DirectoryPrx dir = DirectoryPrx::checkedCast(*i);
        FilePrx file = FilePrx::uncheckedCast(*i);
        cout << indent << (*i)->name()
             << (dir ? " (directory):" : " (file):") << endl;
        if (dir) {
            listRecursive(dir, depth);
        } else {
            Lines text = file->read();
            for (Lines::const_iterator j = text.begin();
                 j != text.end();
                 ++j) {
                cout << indent << "\t" << *j << endl;
            }
        }
    }
}
```

The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.

   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.

2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.

3. The code checks the type of the node:

   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:

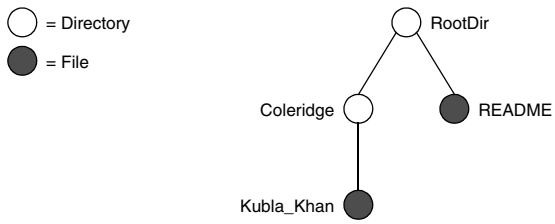


Figure 7.1. A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in Chapter 38 and Chapter 34.

7.3 Summary

This chapter presented a very simple client to access a server that implements the file system we developed in Chapter 5. As you can see, the C++ code hardly differs from the code you would write for an ordinary C++ program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local C++ object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. As we will see in Chapter 9, this is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

Chapter 8
Server-Side Slice-to-C++ Mapping

8.1 Chapter Overview

In this chapter, we present the server-side Slice-to-C++ mapping (see Chapter 6 for the client-side mapping). Section 8.3 discusses how to initialize and finalize the server-side run time, sections 8.4 to 8.6 show how to implement interfaces and operations, and Section 8.7 discusses how to register objects with the server-side Ice run time. Finally, Section 8.8 shows how to implement operations asynchronously.

8.2 Introduction

The mapping for Slice data types to C++ is identical on the client side and server side. This means that everything in Chapter 6 also applies to the server side. However, for the server side, there are a few additional things you need to know, specifically:

- how to initialize and finalize the server-side run time
- how to implement servants
- how to pass parameters and throw exceptions
- how to create servants and register them with the Ice run time.

We discuss these topics in the remainder of this chapter.

8.3 The Server-Side `main` Function

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. As for the client side, you must initialize the Ice run time by calling `Ice::initialize` before you can do anything else in your server. `Ice::initialize` returns a smart pointer to an instance of an `Ice::Communicator`:

```
int
main(int argc, char* argv[])
{
    Ice::CommunicatorPtr ic
        = Ice::initialize(argc, argv);
    // ...
}
```

`Ice::initialize` accepts a C++ reference to `argc` and `argv`. The function scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `Ice::initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.[1]

Before leaving your `main` function, you *must* call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, `destroy` waits for any operation implementations that are still executing in the server to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your `main` function to terminate without calling `destroy` first; doing so has undefined behavior.

The general shape of our server-side `main` function is therefore as follows:

```
#include <Ice/Ice.h>

int
main(int argc, char* argv[])
```

1. `Ice::initialize` has additional overloads to permit other information to be passed to the Ice run time (see Section 32.3).

```
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);

        // Server code here...

    } catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
    } catch (const std::string& msg) {
        cerr << msg << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const std::string& msg) {
            cerr << msg << endl;
            status = 1;
        }
    }
    return status;
}
```

Note that the code places the call to `Ice::initialize` in to a `try` block and takes care to return the correct exit status to the operating system. Also note that an attempt to destroy the communicator is made only if the initialization succeeded.

The `catch` handlers for `const std::string &` and `const char *` are in place as a convenience feature: if we encounter a fatal error condition anywhere in the server code, we can simply throw a string or a string literal containing an error message; this causes the stack to be unwound back to `main`, at which point the error message is printed and, after destroying the communicator, `main` terminates with non-zero exit status.

8.3.1 The `Ice::Application` Class

The preceding structure for the `main` function is so common that Ice offers a class, `Ice::Application`, that encapsulates all the correct initialization and finalization activities. The definition of the class is as follows (with some detail omitted for now):

```
namespace Ice {
    enum SignalPolicy { HandleSignals, NoSignalHandling };

    class Application /* ... */ {
    public:
        Application(SignalPolicy = HandleSignals);
        virtual ~Application();

        int main(int argc, char*[] argv);
        int main(int argc, char*[] argv, const char* config);
        int main(int argc, char*[] argv,
                 const Ice::InitializationData& id);
        int main(int argc, char* const [] argv);
        int main(int argc, char* const [] argv, const char* config
);
        int main(int argc, char* const [] argv,
                 const Ice::InitializationData& id);
        int main(const Ice::StringSeq&);
        int main(const Ice::StringSeq&, const char* config);
        int main(const Ice::StringSeq&,
                 const Ice::InitializationData& id);

#ifdef _WIN32
        int main(int argc, wchar_t*[] argv);
        int main(int argc, wchar_t*[] argv, const char* config);
        int main(int argc, wchar_t*[] argv,
                 const Ice::InitializationData& id);
#endif

        virtual int run(int, char*[]) = 0;

        static  const char* appName();
        static  CommunicatorPtr communicator();
        // ...
    };
}
```

The intent of this class is that you specialize `Ice::Application` and implement the pure virtual `run` method in your derived class. Whatever code you

would normally place in main goes into the run method instead. Using Ice::Application, our program looks as follows:

```
#include <Ice/Ice.h>

class MyApplication : virtual public Ice::Application {
public:
    virtual int run(int, char*[]) {

        // Server code here...

        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

Note that Application::main is overloaded: you can pass a string sequence instead of an argc/argv pair. This is useful if you need to parse application-specific property settings on the command line (see Section 30.9.3). You also can call main with an optional file name or an InitializationData structure (see Section 32.3 and Section 30.9).

If you pass a configuration file name to main, the settings in this file are overridden by settings in a file identified by the ICE_CONFIG environment variable (if defined). Property settings supplied on the command line take precedence over all other settings.

The Application::main function does the following:

1. It installs an exception handler for Ice::Exception. If your code fails to handle an Ice exception, Application::main prints the exception details on stderr before returning with a non-zero return value.
2. It installs exception handlers for const std::string & and const char *. This allows you to terminate your server in response to a fatal error condition by throwing a std::string or a string literal. Application::main prints the string on stderr before returning a non-zero return value.
3. It initializes (by calling Ice::initialize) and finalizes (by calling Communicator::destroy) a communicator. You can get access to the

communicator for your server by calling the static `communicator()` member.

4. It scans the argument vector for options that are relevant to the Ice run time and removes any such options. The argument vector that is passed to your `run` method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.
5. It provides the name of your application via the static `appName` member function. The return value from this call is `argv[0]`, so you can get at `argv[0]` from anywhere in your code by calling `Ice::Application::appName` (which is usually required for error messages).
6. It creates an `IceUtil::CtrlCHandler` that properly destroys the communicator.
7. It installs a per-process logger (see Section 32.19.5) if the application has not already configured one. The per-process logger uses the value of the `Ice.ProgramName` property (see Section 30.8) as a prefix for its messages and sends its output to the standard error channel. An application can specify an alternate logger as described in Section 32.19.

Using `Ice::Application` ensures that your program properly finalizes the Ice run time, whether your server terminates normally or in response to an exception or signal. We recommend that all your programs use this class; doing so makes your life easier. In addition, `Ice::Application` also provides features for signal handling and configuration that you do not have to implement yourself when you use this class.

Using `Ice::Application` on the Client Side

You can use `Ice::Application` for your clients as well: simply implement a class that derives from `Ice::Application` and place the client code into its `run` method. The advantage of this approach is the same as for the server side: `Ice::Application` ensures that the communicator is destroyed correctly even in the presence of exceptions.

Catching Signals

The simple server we developed in Chapter 3 had no way to shut down cleanly: we simply interrupted the server from the command line to force it to exit. Terminating a server in this fashion is unacceptable for many real-life server applications: typically, the server has to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is particu-

larly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

To make it easier to deal with signals, `Ice::Application` encapsulates the platform-independent signal handling capabilities provided by the class `IceUtil::CtrlCHandler` (see Section 31.10). This allows you to cleanly shut down on receipt of a signal and to use the same source code regardless of the underlying operating system and threading package:

```
namespace Ice {
    class Application : /* ... */ {
    public:
        // ...
        static void destroyOnInterrupt();
        static void shutdownOnInterrupt();
        static void ignoreInterrupt();
        static void callbackOnInterrupt();
        static void holdInterrupt();
        static void releaseInterrupt();
        static bool interrupted();

        virtual void interruptCallback(int);
    };
}
```

You can use `Ice::Application` under both Windows and Unix: for Unix, the member functions control the behavior of your application for **`SIGINT`**, **`SIGHUP`**, and **`SIGTERM`**; for Windows, the member functions control the behavior of your application for **`CTRL_C_EVENT`**, **`CTRL_BREAK_EVENT`**, **`CTRL_CLOSE_EVENT`**, **`CTRL_LOGOFF_EVENT`**, and **`CTRL_SHUTDOWN_EVENT`**.

The functions behave as follows:

- `destroyOnInterrupt`

  This function creates an `IceUtil::CtrlCHandler` that destroys the communicator when one of the monitored signals is raised. This is the default behavior.

- `shutdownOnInterrupt`

  This function creates an `IceUtil::CtrlCHandler` that shuts down the communicator when one of the monitored signals is raised.

- `ignoreInterrupt`

  This function causes signals to be ignored.

- `callbackOnInterrupt`

  This function configures `Ice::Application` to invoke `interruptCallback` when a signal occurs, thereby giving the subclass responsibility for handling the signal. Note that if the signal handler needs to terminate the program, you must call `_exit` (instead of `exit`). This prevents global destructors from running which, depending on the activities of other threads in the program, could cause deadlock or assertion failures.

- `holdInterrupt`

  This function causes signals to be held.

- `releaseInterrupt`

  This function restores signal delivery to the previous disposition. Any signal that arrives after `holdInterrupt` was called is delivered when you call `releaseInterrupt`.

- `interrupted`

  This function returns `true` if a signal caused the communicator to shut down, `false` otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by a signal. This is useful, for example, for logging purposes.

- `interruptCallback`

  A subclass overrides this function to respond to signals. The Ice run time may call this function concurrently with any other thread. If the function raises an exception, the Ice run time prints a warning on `cerr` and ignores the exception.

By default, `Ice::Application` behaves as if `destroyOnInterrupt` was invoked, therefore our server `main` function requires no change to ensure that the program terminates cleanly on receipt of a signal. (You can disable the signal-handling functionality of `Ice::Application` by passing the enumerator `NoSignalHandling` to the constructor. In that case, signals retain their default behavior, that is, terminate the process.) However, we add a diagnostic to report the occurrence of a signal, so our `main` function now looks like:

```
#include <Ice/Ice.h>

class MyApplication : virtual public Ice::Application {
public:
    virtual int run(int, char*[]) {

        // Server code here...
```

```
        if (interrupted())
            cerr << appName() << ": terminating" << endl;

        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

Note that, if your server is interrupted by a signal, the Ice run time waits for all currently executing operations to finish. This means that an operation that updates persistent state cannot be interrupted in the middle of what it was doing and cause partial update problems.

Under Unix, if you handle signals with your own handler (by deriving a subclass from `Ice::Application` and calling `callbackOnInterrupt`), the handler is invoked synchronously from a separate thread. This means that the handler can safely call into the Ice run time or make system calls that are not async-signal-safe without fear of deadlock or data corruption. Note that `Ice::Application` blocks delivery of `SIGINT`, `SIGHUP`, and `SIGTERM`. If your application calls `exec`, this means that the child process will also ignore these signals; if you need the default behavior of these signals in the exec'd process, you must explicitly reset them to `SIG_DFL` before calling `exec`.

`Ice::Application` and Properties

Apart from the functionality shown in this section, `Ice::Application` also takes care of initializing the Ice run time with property values. Properties allow you to configure the run time in various ways. For example, you can use properties to control things such as the thread pool size or port number for a server. We discuss Ice properties in more detail in Chapter 30.

Limitations of `Ice::Application`

`Ice::Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use `Ice::Application`. Instead, you must structure your code as we saw in Chapter 3 (taking care to always destroy the communicators).

8.3.2 The `Ice::Service` Class

The `Ice::Application` class described in Section 8.3.1 is very convenient for general use by Ice client and server applications. In some cases, however, an application may need to run at the system level as a Unix daemon or Windows service. For these situations, Ice includes `Ice::Service`, a singleton class that is comparable to `Ice::Application` but also encapsulates the low-level, platform-specific initialization and shutdown procedures common to system services. The `Ice::Service` class is defined as follows:

```
namespace Ice {
    class Service {
    public:
        Service();

        virtual bool shutdown();
        virtual void interrupt();

        int main(int& argc, char* argv[],
                 const Ice::InitializationData& =
                     Ice::InitializationData());
        int main(Ice::StringSeq& args,
                 const Ice::InitializationData& =
                     Ice::InitializationData());

        Ice::CommunicatorPtr communicator() const;

        static Service* instance();

        bool service() const;
        std::string name() const;
        bool checkSystem() const;

        int run(int& argc, char* argv[],
                const Ice::InitializationData&);

#ifdef _WIN32
        int main(int& argc, wchar_t* argv[],
                  const InitializationData& =
                      InitializationData());

        void configureService(const std::string& name);
#else
        void configureDaemon(bool changeDir,
                             bool closeFiles,
```

```
                                const std::string& pidFile);
#endif

        virtual void handleInterrupt(int);

    protected:
        virtual bool start(int argc, char* argv[],
                           int& status) = 0;
        virtual void waitForShutdown();
        virtual bool stop();
        virtual Ice::CommunicatorPtr initializeCommunicator(
            int& argc, char* argv[],
            const Ice::InitializationData&);

        virtual void syserror(const std::string& msg);
        virtual void error(const std::string& msg);
        virtual void warning(const std::string& msg);
        virtual void trace(const std::string& msg);
        virtual void print(const std::string& msg);

        void enableInterrupt();
        void disableInterrupt();

        // ...
    };
}
```

At a minimum, an Ice application that uses the `Ice::Service` class must define a subclass and override the `start` member function, which is where the service must perform its startup activities, such as processing command-line arguments, creating an object adapter, and registering servants. The application's `main` function must instantiate the subclass and typically invokes its `main` member function, passing the program's argument vector as parameters. The example below illustrates a minimal `Ice::Service` subclass:

```
#include <Ice/Service.h>

class MyService : public Ice::Service {
protected:
    virtual bool start(int, char*[]);
private:
    Ice::ObjectAdapterPtr _adapter;
};

bool
```

```
MyService::start(int argc, char* argv[], int& status)
{
    _adapter = communicator()->createObjectAdapter("MyAdapter");
    _adapter->addWithUUID(new MyServantI);
    _adapter->activate();
    status = EXIT_SUCCESS;
    return true;
}

int
main(int argc, char* argv[])
{
    MyService svc;
    return svc.main(argc, argv);
}
```

The `Service::main` member function performs the following sequence of tasks:

1. Scans the argument vector for reserved options that indicate whether the program should run as a system service and removes these options from the argument vector (`argc` is adjusted accordingly). Additional reserved options are supported for administrative tasks.
2. Configures the program for running as a system service (if necessary) by invoking `configureService` or `configureDaemon`, as appropriate for the platform.
3. Invokes the `run` member function and returns its result.

Note that, as for `Application::main`, `Service::main` is overloaded to accept a string sequence instead of an `argc`/`argv` pair. This is useful if you need to parse application-specific property settings on the command line (see Section 30.9.3).

The `Service::run` member function executes the service in the steps shown below:

1. Installs an `IceUtil::CtrlCHandler` (see Section 31.10) for proper signal handling.
2. Invokes the `initializeCommunicator` member function to obtain a communicator. The communicator instance can be accessed using the `communicator` member function.
3. Invokes the `start` member function. If `start` returns `false` to indicate failure, `run` destroys the communicator and returns immediately using the exit status provided in `status`.

4. Invokes the `waitForShutdown` member function, which should block until `shutdown` is invoked.
5. Invokes the `stop` member function. If `stop` returns `true`, `run` considers the application to have terminated successfully.
6. Destroys the communicator.
7. Gracefully terminates the system service (if necessary).

If an unhandled exception is caught by `Service::run`, a descriptive message is logged, the communicator is destroyed and the service is terminated.

`Ice::Service` Member Functions

The virtual member functions in `Ice::Service` represent the points at which a subclass can intercept the service activities. All of the virtual member functions (except `start`) have default implementations.

- `void handleInterrupt(int sig)`

  Invoked by the `CtrlCHandler` when a signal occurs. The default implementation ignores the signal if it represents a logoff event and the `Ice.Nohup` property is set to a value larger than zero, otherwise it invokes the `interrupt` member function.

- `Ice::CommunicatorPtr initializeCommunicator(int & argc, char * argv[], const Ice::InitializationData & data)`

  Initializes a communicator. The default implementation invokes `Ice::initialize` and passes the given arguments.

- `void interrupt()`

  Invoked by the signal handler to indicate a signal was received. The default implementation invokes the `shutdown` member function.

- `bool shutdown()`

  Causes the service to begin the shutdown process. The default implementation invokes `shutdown` on the communicator. The subclass must return `true` if shutdown was started successfully, and `false` otherwise.

- `bool start(int argc, char * argv[], int & status)`

  Allows the subclass to perform its startup activities, such as scanning the provided argument vector for recognized command-line options, creating an object adapter, and registering servants. The subclass must return `true` if

startup was successful, and `false` otherwise. The subclass can set an exit status via the `status` parameter. This status is returned by `main`.

- `bool stop()`

  Allows the subclass to clean up prior to termination. The default implementation does nothing but return `true`. The subclass must return `true` if the service has stopped successfully, and `false` otherwise.

- `void syserror(const std::string & msg)`
- `void error(const std::string & msg)`
- `void warning(const std::string & msg)`
- `void trace(const std::string & msg)`
- `void print(const std::string & msg)`

  Convenience functions for logging messages to the communicator's logger. The `syserror` member function includes a description of the system's current error code.

- `void waitForShutdown()`

  Waits indefinitely for the service to shut down. The default implementation invokes `waitForShutdown` on the communicator.

The non-virtual member functions shown in the class definition are described below:

- `bool checkSystem() const`

  Returns true if the operating system supports Windows services or Unix daemons. This function returns false on Windows 95/98/ME.

- `Ice::CommunicatorPtr communicator() const`

  Returns the communicator used by the service, as created by `initializeCommunicator`.

- `void configureDaemon(bool chdir, bool close,`
  `const std::string & pidFile)`

  Configures the program to run as a Unix daemon. The `chdir` parameter determines whether the daemon changes its working directory to the root directory. The `close` parameter determines whether the daemon closes unnecessary file descriptors (i.e., stdin, stdout, etc.). If a non-empty string is provided in the `pidFile` parameter, the daemon writes its process ID to the given file.

- `void configureService(const std::string & name)`

  Configures the program to run as a Windows service with the given name.

- `void disableInterrupt()`

  Disables the signal handling behavior in `Ice::Service`. When disabled, signals are ignored.

- `void enableInterrupt()`

  Enables the signal handling behavior in `Ice::Service`. When enabled, the occurrence of a signal causes the `handleInterrupt` member function to be invoked.

- `static Service * instance()`

  Returns the singleton `Ice::Service` instance.

- ```
  int main(int & argc, char * argv[],
           const Ice::InitializationData & data =
               Ice::InitializationData())
  int main(Ice::StringSeq& args,
           const Ice::InitializationData& =
               Ice::InitializationData());
  int main(int & argc, wchar_t * argv[],
           const Ice::InitializationData & data =
               Ice::InitializationData())
  ```

  The primary entry point of the `Ice::Service` class. The tasks performed by this function are described earlier in this section. The function returns `EXIT_SUCCESS` for success, `EXIT_FAILURE` for failure. For Windows, this function is overloaded to allow you to pass a `wchar_t` argument vector.

- `std::string name() const`

  Returns the name of the service. If the program is running as a Windows service, the return value is the Windows service name, otherwise it returns the value of `argv[0]`.

- ```
  int run(int & argc, char * argv[],
          const Ice::InitializationData & data)
  ```

  Alternative entry point for applications that prefer a different style of service configuration. The program must invoke `configureService` (Windows) or `configureDaemon` (Unix) in order to run as a service. The tasks performed by this function are described earlier in this section. The function normally returns `EXIT_SUCCESS` or `EXIT_FAILURE`, but the `start` method can also supply a different value via its `status` argument.

- `bool service() const`

  Returns true if the program is running as a Windows service or Unix daemon, or false otherwise.

Unix Daemons

On Unix platforms, `Ice::Service` recognizes the following command-line options:

- **`--daemon`**

  Indicates that the program should run as a daemon. This involves the creation of a background child process in which `Service::main` performs its tasks. The parent process does not terminate until the child process has successfully invoked the `start` member function[2]. Unless instructed otherwise, `Ice::Service` changes the current working directory of the child process to the root directory, and closes all unnecessary file descriptors. Note that the file descriptors are not closed until after the communicator is initialized, meaning standard input, standard output, and standard error are available for use during this time. For example, the IceSSL plug-in may need to prompt for a passphrase on standard input, or Ice may print the child's process id on standard output if the property `Ice.PrintProcessId` is set.

- **`--pidfile` *`FILE`***

  This option writes the process ID of the service into the specified file. (This option requires **`--daemon`**.)

- **`--noclose`**

  Prevents `Ice::Service` from closing unnecessary file descriptors. This can be useful during debugging and diagnosis because it provides access to the output from the daemon's standard output and standard error.

- **`--nochdir`**

  Prevents `Ice::Service` from changing the current working directory.

The **`--noclose`** and **`--nochdir`** options can only be specified in conjunction with **`--daemon`**. These options are removed from the argument vector that is passed to the `start` member function.

2. This behavior avoids the uncertainty often associated with starting a daemon from a shell script, because it ensures that the command invocation does not complete until the daemon is ready to receive requests.

Windows Services

On Windows, `Ice::Service` recognizes the following command-line options:

- **`--service NAME`**

  Run as a Windows service named **`NAME`**, which must already be installed. This option is removed from the argument vector that is passed to the `start` member function.

Installing and configuring a Windows service is outside the scope of the `Ice::Service` class. Ice includes a utility for installing its services (see Appendix H) which you can use as a model for your own applications.

The `Ice::Service` class supports the Windows service control codes `SERVICE_CONTROL_INTERROGATE` and `SERVICE_CONTROL_STOP`. Upon receipt of `SERVICE_CONTROL_STOP`, `Ice::Service` invokes the `shutdown` member function.

Logging Considerations

A service that uses a custom logger has several ways of configuring it:

- as a process-wide logger (see Section 32.19.5),
- in the `InitializationData` argument that is passed to `main`,
- by overriding the `initializeCommunicator` member function.

On Windows, `Ice::Service` installs its own logger that uses the Windows `Application` event log if no custom logger is defined. The source name for the event log is the service's name unless a different value is specified using the property `Ice.EventLog.Source` (see Appendix D).

On Unix, the default Ice logger (which logs to the standard error output) is used when no other logger is configured. For daemons, this is not appropriate because the output will be lost. To change this, you can either implement a custom logger or set the `Ice.UseSyslog` property, which selects a logger implementation that logs to the **`syslog`** facility. Alternatively, you can set the `Ice.LogFile` property to write log messages to a file.

Note that `Ice::Service` may encounter errors before the communicator is initialized. In this situation, `Ice::Service` uses its default logger unless a process-wide logger is configured. Therefore, even if a failing service is configured to use a different logger implementation, you may find useful diagnostic information in the `Application` event log (on Windows) or sent to standard error (on Unix).

8.4 Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing virtual functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

8.4.1 Skeleton Classes

On the client side, interfaces map to proxy classes (see Section 6.11). On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has a pure virtual member function for each operation on the corresponding interface. For example, consider the Slice definition for the `Node` interface we defined in Chapter 5 once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The Slice compiler generates the following definition for this interface:

```
namespace Filesystem {

    class Node : virtual public Ice::Object {
    public:
        virtual std::string name(const Ice::Current& =
                                     Ice::Current()) = 0;
        // ...
    };
    // ...
}
```

For the moment, we will ignore a number of other member functions of this class. The important points to note are:

- As for the client side, Slice modules are mapped to C++ namespaces with the same name, so the skeleton class definition is nested in the namespace `Filesystem`.
- The name of the skeleton class is the same as the name of the Slice interface (`Node`).

- The skeleton class contains a pure virtual member function for each operation in the Slice interface.
- The skeleton class is an abstract base class because its member functions are pure virtual.
- The skeleton class inherits from `Ice::Object` (which forms the root of the Ice object hierarchy).

8.4.2 Servant Classes

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

```
#include <Filesystem.h> // Slice-generated header

class NodeI : public virtual Filesystem::Node {
public:
    NodeI(const std::string&);
    virtual std::string name(const Ice::Current&);
private:
    std::string _name;
};
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can chose any name you prefer for your servant classes.)

Note that `NodeI` inherits from `Filesystem::Node`, that is, it derives from its skeleton class. It is a good idea to always use virtual inheritance when defining servant classes. Strictly speaking, virtual inheritance is necessary only for servants that implement interfaces that use multiple inheritance; however, the `virtual` keyword does no harm and, if you add multiple inheritance to an interface hierarchy half-way through development, you do not have to go back and add a `virtual` keyword to all your servant classes.

As far as Ice is concerned, the `NodeI` class must implement only a single member function: the pure virtual `name` function that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. Obviously, the constructor initializes the `_name` member and the `name` function returns its value:

```
NodeI::NodeI(const std::string& name) : _name(name)
{
}

std::string
NodeI::name(const Ice::Current&) const
{
    return _name;
}
```

Normal and `idempotent` Operations

The `name` member function of the `NodeI` skeleton on page 291 is not a `const` member function. However, given that the operation does not modify the state of its object, it really should be a `const` member function. We can achieve this by adding the `["cpp:const"]` metadata directive. For example:

```
interface Example {
                void normalOp();

    idempotent  void idempotentOp();

    ["cpp:const"]
    idempotent  void readonlyOp();
};
```

The skeleton class for this interface looks like this:

```
class Example : virtual public Ice::Object {
public:
    virtual void normalOp(const Ice::Current&
                                = Ice::Current()) = 0;
    virtual void idempotentOp(const Ice::Current&
                                = Ice::Current()) = 0;
    virtual void readonlyOp(const Ice::Current&
                                = Ice::Current()) const = 0;
    // ...
};
```

Note that `readonlyOp` is mapped as a `const` member function due to the `["cpp:const"]` metadata directive; normal and `idempotent` operations (without the metadata directive) are mapped as ordinary, non-`const` member functions.

8.5 Parameter Passing

For each parameter of a Slice operation, the C++ mapping generates a corresponding parameter for the virtual member function in the skeleton. In addition, every operation has an additional, trailing parameter of type `Ice::Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` member function of the `Node` skeleton class has a single parameter of type `Ice::Current`. We explain the purpose of this parameter in Section 32.6 and will ignore it for now.

Parameter passing on the server side follows the rules for the client side:

- in-parameters are passed by value or `const` reference.
- out-parameters are passed by reference.
- return values are passed by value

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

```
module M {
    interface Example {
        string op(string sin, out string sout);
    };
};
```

The generated skeleton class for this interface looks as follows:

```
namespace M {
    class Example : virtual public ::Ice::Object {
    public:
        virtual std::string
                    op(const std::string&, std::string&,
                       const Ice::Current& = Ice::Current()) = 0;
        // ...
    };
}
```

As you can see, there are no surprises here. For example, we could implement `op` as follows:

```
std::string
ExampleI::op(const std::string& sin,
                   std::string& sout,
             const Ice::Current&)
{
```

```
    cout << sin << endl;        // In parameters are initialized
    sout = "Hello World!";      // Assign out parameter
    return "Done";              // Return a string
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal C++ rules and do not require special-purpose API calls or memory management.

8.6 Raising Exceptions

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

```
void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
    // Try to write the file contents here...
    // Assume we are out of space...
    if (error) {
        Filesystem::GenericError e;
        e.reason = "file too large";
        throw e;
    }
};
```

No memory management issues arise in the presence of exceptions.

Note that the Slice compiler never generates exception specifications for operations, regardless of whether the corresponding Slice operation definition has an exception specification or not. This is deliberate: C++ exception specifications do not add any value and are therefore not used by the Ice C++ mapping. (See [22] for an excellent treatment of the problems associated with C++ exception specifications.)

If you throw an arbitrary C++ exception (such as an `int` or other unexpected type), the Ice run time catches the exception and then returns an `UnknownException` to the client. Similarly, if you throw an “impossible” user exception (a user

exception that is not listed in the exception specification of the operation), the client receives an `UnknownUserException`.

If you throw a run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`.[3] For that reason, you should never throw system exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

8.7 Object Incarnation

Having created a servant class such as the rudimentary `NodeI` class in Section 8.4.2, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must follow the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

8.7.1 Instantiating a Servant

Instantiating a servant means to allocate an instance on the heap:

```
NodePtr servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance on the heap and assigns its address to a smart pointer of type `NodePtr` (see also page 238). This works because `NodeI` is derived from `Node`, so a smart pointer of type `NodePtr` can also look after an instance of type `NodeI`. However, if we want to invoke a member function of the derived `NodeI` class at this point, we have a problem: we cannot access member functions of the derived `NodeI` class through a `NodePtr` smart pointer, only member functions of `Node` base class. (The C++ type rules prevent us from

3. There are three system exceptions that are not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`. We discuss these exceptions in more detail in Section 4.10.4 and Chapter 33.

accessing a member of a derived class through a pointer to a base class.) To get around this, we can modify the code as follows:

```
typedef IceUtil::Handle<NodeI> NodeIPtr;
NodeIPtr servant = new NodeI("Fred");
```

This code makes use of the smart pointer template we presented in Section 6.14.6 by defining `NodeIPtr` as a smart pointer to `NodeI` instances. Whether you use a smart pointer of type `NodePtr` or `NodeIPtr` depends solely on whether you want to invoke a member function of the `NodeI` derived class; if you only want to invoke member functions that are defined in the `Node` skeleton base class, it is sufficient to use a `NodePtr` and you need not define the `NodeIPtr` type.

Whether you use `NodePtr` or `NodeIPtr`, the advantages of using a smart pointer class should be obvious from the discussion in Section 6.14.6: they make it impossible to accidentally leak memory.

8.7.2 Creating an Identity

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.[4] An Ice object identity is a structure with the following Slice definition:

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
    // ...
};
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (See Section 32.7 for a discussion of the `category` field.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

```
Ice::Identity id;
id.name = "Fred"; // Not unique, but good enough for now
```

4. The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity. See Chapter 34 for further discussion.

8.7.3 Activating a Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the add operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

```
_adapter->add(servant, id);
```

Note the two arguments to `add`: the smart pointer to the servant and the object identity. Calling `add` on the object adapter adds the servant pointer and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant pointer from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state (see Section 32.4.5), client requests are dispatched to the servant as soon as you call `add`.

Servant Life Time and Reference Counts

Putting the preceding points together, we can write a simple function that instantiates and activates one of our `NodeI` servants. For this example, we use a simple helper function called `activateServant` that creates and activates a servant with a given identity:

```
void
activateServant(const string& name)
{
    NodePtr servant = new NodeI(name);          // Refcount == 1
    Ice::Identity id;
    id.name = name;
    _adapter->add(servant, id);                 // Refcount == 2
}                                               // Refcount == 1
```

Note that we create the servant on the heap and that, once `activateServant` returns, we lose the last remaining handle to the servant (because the `servant`

variable goes out of scope). The question is, what happens to the heap-allocated servant instance? The answer lies in the smart pointer semantics:

- When the new servant is instantiated, its reference count is initialized to 0.
- Assigning the servant's address to the `servant` smart pointer increments the servant's reference count to 1.
- Calling `add` passes the `servant` smart pointer to the object adapter which keeps a copy of the handle internally. This increments the reference count of the servant to 2.
- When `activateServant` returns, the destructor of the `servant` variable decrements the reference count of the servant to 1.

The net effect is that the servant is retained on the heap with a reference count of 1 for as long as the servant is in the servant map of its object adapter. (If we deactivate the servant, that is, remove it from the servant map, the reference count drops to zero and the memory occupied by the servant is reclaimed; we discuss these life cycle issues in Chapter 34.)

8.7.4 UUIDs as Identities

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) [14] as identities. The `IceUtil` namespace contains a helper function to create such identities:

```
#include <IceUtil/UUID.h>
#include <iostream>

using namespace std;

int
main()
{
    cout << IceUtil::generateUUID() << endl;
}
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` creates a string that differs from all previous ones.[5] You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant

map in a single step. Using this operation, we can rewrite the code on page 297 like this:

```
void
activateServant(const string& name)
{
    NodePtr servant = new NodeI(name);
    _adapter->addWithUUID(servant);
}
```

8.7.5 Creating Proxies

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Chapter 3. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

```
typedef IceUtil::Handle<NodeI> NodeIPtr;
NodeIPtr servant = new NodeI(name);
NodePrx proxy = NodePrx::uncheckedCast(
                                _adapter->addWithUUID(servant));

// Pass proxy to client...
```

5. Well, almost: eventually, the UUID algorithm wraps around and produces strings that repeat themselves, but this will not happen until approximately the year 3400.

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice::ObjectPrx`.

Direct Proxy Creation

The object adapter offers an operation to create a proxy for a given identity:

```
module Ice {
    local interface ObjectAdapter {
        Object* createProxy(Identity id);
        // ...
    };
};
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

```
Ice::Identity id;
id.name = IceUtil::generateUUID();
ObjectPrx o = _adapter->createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Chapter 34.)

8.8 Asynchronous Method Dispatch (AMD)

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool (see Section 32.10). If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (*AMD*), the server-side equivalent of AMI (see Section 6.15), addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available,

the server sends a response explicitly using a callback object provided by the Ice run time.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data (i.e., the callback object and operation arguments) for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results while still in the dispatch thread, and then continues using the dispatch thread for additional work.

8.8.1 Enabling AMD with Metadata

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive *replaces* synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

```
["amd"] interface I {
  bool isValid();
  float computeRate();
};

interface J {
  ["amd"] void startProcess();
  int endProcess();
};
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface or class level) minimizes the amount of generated code and, more importantly, minimizes

complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

8.8.2 AMD Mapping

The C++ mapping emits the following code for each AMD operation:

1. A callback class used by the implementation to notify the Ice run time about the completion of an operation. The name of this class is formed using the pattern `AMD_class_op`. For example, an operation named `foo` defined in interface `I` results in a class named `AMD_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Several methods are provided:

   ```
   void ice_response(<params>);
   ```

   The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-`void` return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's `out` parameters follow the return value, in the order of declaration.

   ```
   void ice_exception(const std::exception &);
   ```

   This version of `ice_exception` allows the server to raise any standard exception, Ice run-time exception, or Ice user exception.

   ```
   void ice_exception()
   ```

   This version of `ice_exception` allows the server to report an `UnknownException`.

   Neither `ice_response` nor `ice_exception` throw any exceptions to the caller.

2. The dispatch method, whose name has the suffix `_async`. This method has a `void` return type. The first parameter is a smart pointer to an instance of the callback class described above. The remaining parameters comprise the in-parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
  ["amd"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

```
class AMD_I_foo : public ... {
public:
    void ice_response(Ice::Int, Ice::Long);
    void ice_exception(const std::exception&);
    void ice_exception();
};
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

```
void foo_async(const AMD_I_fooPtr&, Ice::Short);
```

8.8.3 Exceptions

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread that receives the invocation), and the response thread (the thread that sends the response).[6] Although we recommend that the callback object be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run-time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the callback object. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

Whether raised in a dispatch thread or reported via the callback object, user exceptions are validated as described in Section 4.10.2, and local exceptions may undergo the translation described in Section 4.10.4.

6. These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

8.8.4 Example

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

```
module Demo {
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {};

    interface Model {
        ["amd"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    };
};
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo::Model` and supplies a definition for the `interpolate_async` method:

```
class ModelI : virtual public Demo::Model,
               virtual public IceUtil::Mutex {
public:
    virtual void interpolate_async(
        const Demo::AMD_Model_interpolatePtr&,
        const Demo::Grid&,
        Ice::Float,
        const Ice::Current&);

private:
  std::list<JobPtr> _jobs;
};
```

The implementation of `interpolate_async` uses synchronization to safely record the callback object and arguments in a `Job` that is added to a queue:

```
void ModelI::interpolate_async(
    const Demo::AMD_Model_interpolatePtr& cb,
    const Demo::Grid& data,
    Ice::Float factor,
    const Ice::Current& current)
{
```

```
    IceUtil::Mutex::Lock sync(*this);
    JobPtr job = new Job(cb, data, factor);
    _jobs.push_back(job);
}
```

After queuing the information, the operation returns control to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute` to perform the interpolation. `Job` is defined as follows:

```
class Job : public IceUtil::Shared {
public:
    Job(
        const Demo::AMD_Model_interpolatePtr&,
        const Demo::Grid&,
        Ice::Float);
    void execute();

private:
    bool interpolateGrid();

    Demo::AMD_Model_interpolatePtr _cb;
    Demo::Grid _grid;
    Ice::Float _factor;
};
typedef IceUtil::Handle<Job> JobPtr;
```

The implementation of `execute` uses `interpolateGrid` (not shown) to perform the computational work:

```
Job::Job(
    const Demo::AMD_Model_interpolatePtr& cb,
    const Demo::Grid& grid,
    Ice::Float factor) :
    _cb(cb), _grid(grid), _factor(factor)
{
}

void Job::execute()
{
    if (!interpolateGrid()) {
        _cb->ice_exception(Demo::RangeError());
        return;
    }
    _cb->ice_response(_grid);
}
```

If `interpolateGrid` returns `false`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

8.9 Summary

This chapter presented the server-side C++ mapping. Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanism to the client side: a small API to initialize and finalize the run time, plus a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Even though the examples in this chapter are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers (which we discuss in Chapter 32), you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any C++ mapping rules beyond those we described here.

Chapter 9
Developing a File System Server in C++

9.1 Chapter Overview

In this chapter, we present the source code for a C++ server that implements the file system we developed in Chapter 5 (see Chapter 7 for the corresponding client). The code we present here is fully functional, apart from the required interlocking for threads. (We examine threading issues in detail in Chapter 31.)

9.2 Implementing a File System Server

We have now seen enough of the server-side C++ mapping to implement a server for the file system we developed in Chapter 5. (You may find it useful to review the Slice definition for our file system in Section 5 before studying the source code.)

Our server is composed of two source files:

- `Server.cpp`

  This file contains the server main program.

- `FilesystemI.cpp`

  This file contains the implementation for the file system servants.

9.2.1 The Server `main` Program

Our server main program, in the file `Server.cpp`, uses the `Ice::Application` class we discussed in Section 8.3.1. The `run` method installs a signal handler, creates an object adapter, instantiates a few servants for the directories and files in the file system, and then activates the adapter. This leads to a `main` program as follows:

```
#include <Ice/Ice.h>
#include <FilesystemI.h>

using namespace std;
using namespace Filesystem;

class FilesystemApp : virtual public Ice::Application {
public:
    virtual int run(int, char*[]) {
        // Terminate cleanly on receipt of a signal
        //
        shutdownOnInterrupt();

        // Create an object adapter.
        //
        Ice::ObjectAdapterPtr adapter =
            communicator()->createObjectAdapterWithEndpoints(
                        "SimpleFilesystem", "default -p 10000");

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root =
            new DirectoryI(communicator(), "/", 0);
        root->activate(adapter);

        // Create a file called "README" in the root directory
        //
        FileIPtr file = new FileI(communicator(), "README", root);
        Lines text;
        text.push_back("This file system contains "
                       "a collection of poetry.");
        file->write(text);
        file->activate(adapter);

        // Create a directory called "Coleridge"
        // in the root directory
        //
        DirectoryIPtr coleridge =
```

```
            new DirectoryI(communicator(), "Coleridge", root);
        coleridge->activate(adapter);

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = new FileI(communicator(), "Kubla_Khan", coleridge);
        text.erase(text.begin(), text.end());
        text.push_back("In Xanadu did Kubla Khan");
        text.push_back("A stately pleasure-dome decree:");
        text.push_back("Where Alph, the sacred river, ran");
        text.push_back("Through caverns measureless to man");
        text.push_back("Down to a sunless sea.");
        file->write(text);
        file->activate(adapter);

        // All objects are created, allow client requests now
        //
        adapter->activate();

        // Wait until we are done
        //
        communicator()->waitForShutdown();
        if (interrupted()) {
            cerr << appName()
                 << ": received signal, shutting down" << endl;
        }

        return 0;
    };
};

int
main(int argc, char* argv[])
{
    FilesystemApp app;
    return app.main(argc, argv);
}
```

There is quite a bit of code here, so let us examine each section in detail:

```
#include <FilesystemI.h>
#include <Ice/Application.h>

using namespace std;
using namespace Filesystem;
```

The code includes the header file `FilesystemI.h` (see page 319). That file includes `Ice/Ice.h` as well as the header file that is generated by the Slice compiler, `Filesystem.h`. Because we are using `Ice::Application`, we need to include `Ice/Application.h` as well.

Two using declarations, for the namespaces `std` and `Filesystem`, permit us to be a little less verbose in the source code.

The next part of the source code is the definition of `FilesystemApp`, which derives from `Ice::Application` and contains the main application logic in its `run` method:

```
class FilesystemApp : virtual public Ice::Application {
public:
    virtual int run(int, char*[]) {
        // Terminate cleanly on receipt of a signal
        //
        shutdownOnInterrupt();

        // Create an object adapter.
        //
        Ice::ObjectAdapterPtr adapter =
            communicator()->createObjectAdapterWithEndpoints(
                        "SimpleFilesystem", "default -p 10000");

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root =
            new DirectoryI(communicator(), "/", 0);
        root->activate(adapter);

        // Create a file called "README" in the root directory
        //
        FileIPtr file = new FileI(communicator(), "README", root);
        Lines text;
        text.push_back("This file system contains "
                       "a collection of poetry.");
        file->write(text);
        file->activate(adapter);

        // Create a directory called "Coleridge"
        // in the root directory
        //
        DirectoryIPtr coleridge =
            new DirectoryI(communicator(), "Coleridge", root);
        coleridge->activate(adapter);
```

```
        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = new FileI(communicator(), "Kubla_Khan", coleridge);
        text.erase(text.begin(), text.end());
        text.push_back("In Xanadu did Kubla Khan");
        text.push_back("A stately pleasure-dome decree:");
        text.push_back("Where Alph, the sacred river, ran");
        text.push_back("Through caverns measureless to man");
        text.push_back("Down to a sunless sea.");
        file->write(text);
        file->activate(adapter);

        // All objects are created, allow client requests now
        //
        adapter->activate();

        // Wait until we are done
        //
        communicator()->waitForShutdown();
        if (interrupted()) {
            cerr << appName()
                 << ": received signal, shutting down" << endl;
        }

        return 0;
    };
};
```

Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown in Figure 9.1.

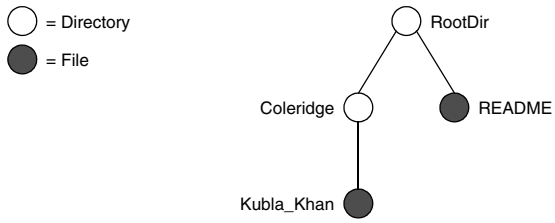


Figure 9.1. A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts three parameters: the communicator, the name of the directory or file to be created, and a handle to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent handle.) Thus, the statement

```
DirectoryIPtr root = new DirectoryI(communicator(), "/", 0);
```

creates the root directory, with the name `"/"` and no parent directory. Note that we use the smart pointer class we discussed in Section 6.14.6 to hold the return value from `new`; that way, we avoid any memory management issues. The types `DirectoryIPtr` and `FileIPtr` are defined as follows in a header file `FilesystemI.h` (see page 319):

```
typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;
typedef IceUtil::Handle<FileI> FileIPtr;
```

Here is the code that establishes the structure in Figure 9.1:

```
// Create the root directory (with name "/" and no parent)
//
DirectoryIPtr root =
    new DirectoryI(communicator(), "/", 0);
root->activate(adapter);

// Create a file called "README" in the root directory
//
FileIPtr file = new FileI(communicator(), "README", root);
Lines text;
text.push_back("This file system contains "
               "a collection of poetry.");
```

```
file->write(text);
file->activate(adapter);

// Create a directory called "Coleridge"
// in the root directory
//
DirectoryIPtr coleridge =
    new DirectoryI(communicator(), "Coleridge", root);
coleridge->activate(adapter);

// Create a file called "Kubla_Khan"
// in the Coleridge directory
//
file = new FileI(communicator(), "Kubla_Khan", coleridge);
text.erase(text.begin(), text.end());
text.push_back("In Xanadu did Kubla Khan");
text.push_back("A stately pleasure-dome decree:");
text.push_back("Where Alph, the sacred river, ran");
text.push_back("Through caverns measureless to man");
text.push_back("Down to a sunless sea.");
file->write(text);
file->activate(adapter);
```

We first create the root directory and a file `README` within the root directory. (Note that we pass the handle to the root directory as the parent pointer when we create the new node of type `FileI`.)

After creating each servant, the code calls `activate` on the servant. (We will see the definition of this member function shortly.) The `activate` member function adds the servant to the ASM.

The next step is to fill the file with text:

```
FileIPtr file = new FileI(communicator(), "README", root);
Lines text;
text.push_back("This file system contains "
               "a collection of poetry.");
file->write(text);
file->activate(adapter);
```

Recall from Section 6.7.3 that Slice sequences map to STL vectors. The Slice type `Lines` is a sequence of strings, so the C++ type `Lines` is a vector of strings; we add a line of text to our `README` file by calling `push_back` on that vector.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

```
file->write(text);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a smart class pointer (of type `FilePtr`) and *not* via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place—such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary C++ function call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in Figure 9.1.

9.2.2 The Servant Class Definitions

We must provide servants for the concrete interfaces in our Slice specification, that is, we must provide servants for the `File` and `Directory` interfaces in the C++ classes `FileI` and `DirectoryI`. This means that our servant classes might look as follows:

```
namespace Filesystem {
  class FileI : virtual public File {
    // ...
  };

  class DirectoryI : virtual public Directory {
     // ...
  };
}
```

This leads to the C++ class structure as shown in Figure 9.2.

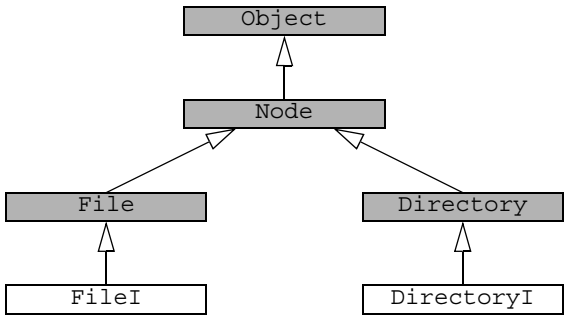


Figure 9.2. File system servants using interface inheritance.

The shaded classes in Figure 9.2 are skeleton classes and the unshaded classes are our servant implementations. If we implement our servants like this, `FileI` must implement the pure virtual operations it inherits from the `File` skeleton (`read` and `write`), as well as the operation it inherits from the `Node` skeleton (`name`). Similarly, `DirectoryI` must implement the pure virtual function it inherits from the `Directory` skeleton (`list`), as well as the operation it inherits from the `Node` skeleton (`name`). Implementing the servants in this way uses interface inheritance from `Node` because no implementation code is inherited from that class.

Alternatively, we can implement our servants using the following definitions:

```
namespace Filesystem {
  class NodeI : virtual public Node {
    // ...
  };

  class FileI : virtual public File,
                virtual public NodeI {
    // ...
  };

  class DirectoryI : virtual public Directory,
                     virtual public NodeI {
    // ...
  };
}
```

This leads to the C++ class structure shown in Figure 9.3.

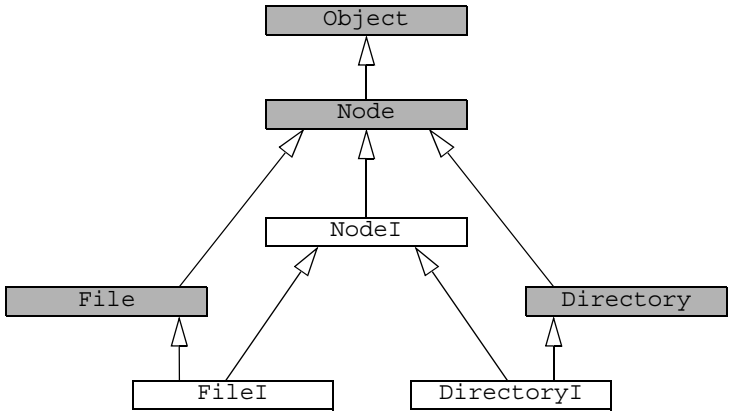


Figure 9.3. File system servants using implementation inheritance.

In this implementation, `NodeI` is a concrete base class that implements the `name` operation it inherits from the `Node` skeleton. `FileI` and `DirectoryI` use multiple inheritance from `NodeI` and their respective skeletons, that is, `FileI` and `DirectoryI` use implementation inheritance from their `NodeI` base class.

Either implementation approach is equally valid. Which one to choose simply depends on whether we want to re-use common code provided by `NodeI`. For the implementation that follows, we have chosen the second approach, using implementation inheritance.

Given the structure in Figure 9.3 and the operations we have defined in the Slice definition for our file system, we can add these operations to the class definition for our servants:

```
namespace Filesystem {
  class NodeI : virtual public Node {
  public:
    virtual std::string name(const Ice::Current&);
  };

  class FileI : virtual public File,
                virtual public NodeI {
  public:
    virtual Lines read(const Ice::Current&);
    virtual void write(const Lines&,
                       const Ice::Current&);
```

```
  };

  class DirectoryI : virtual public Directory,
                     virtual public NodeI {
  public:
    virtual NodeSeq list(const Ice::Current&);
  };
}
```

This simply adds signatures for the operation implementations to each class. Note that the signatures must exactly match the operation signatures in the generated skeleton classes—if they do not match exactly, you end up overloading the pure virtual function in the base class instead of overriding it, meaning that the servant class cannot be instantiated because it will still be abstract. To avoid signature mismatches, you can copy the signatures from the generated header file (`Filesystem.h`), or you can use the **`--impl`** option with **`slice2cpp`** to generate header and implementation files that you can add your application code to (see Section 6.16).

Now that we have the basic structure in place, we need to think about other methods and data members we need to support our servant implementation. Typically, each servant class hides the copy constructor and assignment operator, and has a constructor to provide initial state for its data members. Given that all nodes in our file system have both a name and a parent directory, this suggests that the `NodeI` class should implement the functionality relating to tracking the name of each node, as well as the parent–child relationships:

```
namespace Filesystem {
  class DirectoryI;
  typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

  class NodeI : virtual public Node {
  public:
    virtual std::string name(const Ice::Current&);
    NodeI(const Ice::CommunicatorPtr&,
          const std::string&,
          const DirectoryIPtr&);
    void activate(const Ice::ObjectAdapterPtr&);
  private:
    std::string _name;
    Ice::Identity _id;
    DirectoryIPtr _parent;
```

```
    NodeI(const NodeI&);                // Copy forbidden
    void operator=(const NodeI&);       // Assignment forbidden
  };
}
```

The `NodeI` class has a private data member to store its name (of type `std::string`) and its parent directory (of type `DirectoryIPtr`). The constructor accepts parameters that set the value of these data members. For the root directory, by convention, we pass a null handle to the constructor to indicate that the root directory has no parent. The constructor also requires the communicator to be passed to it. This is necessary because the constructor creates the identity for the servant, which requires access to the communicator. The `activate` member function adds the servant to the ASM (which requires access to the object adapter) and connects the child to its parent.

The `FileI` servant class must store the contents of its file, so it requires a data member for this. We can conveniently use the generated `Lines` type (which is a `std::vector<std::string>`) to hold the file contents, one string for each line. Because `FileI` inherits from `NodeI`, it also requires a constructor that accepts the communicator, file name, and parent directory, leading to the following class definition:

```
namespace Filesystem {
  class FileI : virtual public File,
                virtual public NodeI {
  public:
    virtual Lines read(const Ice::Current&);
    virtual void write(const Lines&,
                       const Ice::Current&);
    FileI(const Ice::CommunicatorPtr&,
          const std::string&,
          const DirectoryIPtr&);
  private:
    Lines _lines;
  };
}
```

For directories, each directory must store its list of child notes. We can conveniently use the generated `NodeSeq` type (which is a `vector<NodePrx>`) to do this. Because `DirectoryI` inherits from `NodeI`, we need to add a constructor to initialize the directory name and its parent directory. As we will see shortly, we also need a private helper function, `addChild`, to make it easier to connect a newly created directory to its parent. This leads to the following class definition:

```
namespace Filesystem {
  class DirectoryI : virtual public Directory,
                     virtual public NodeI {
  public:
    virtual NodeSeq list(const Ice::Current&) const;
    DirectoryI(const Ice::CommunicatorPtr&,
               const std::string&,
               const DirectoryIPtr&);
    void addChild(NodePrx child);
  private:
    NodeSeq _contents;
  };
}
```

Putting all this together, we end up with a servant header file, `FilesystemI.h`, as follows:

```
#include <Ice/Ice.h>
#include <Filesystem.h>

namespace Filesystem {
  class DirectoryI;
  typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

  class NodeI : virtual public Node {
  public:
    virtual std::string name(const Ice::Current&);
    NodeI(const Ice::CommunicatorPtr&,
          const std::string&,
          const DirectoryIPtr&);
    void activate(const Ice::ObjectAdapterPtr&);
  private:
    std::string _name;
    Ice::Identity _id;
    DirectoryIPtr _parent;
    NodeI(const NodeI&);            // Copy forbidden
    void operator=(const NodeI&);   // Assignment forbidden
  };

  typedef IceUtil::Handle<NodeI> NodeIPtr;

  class FileI : virtual public File,
                virtual public NodeI {
  public:
    virtual Lines read(const Ice::Current&);
    virtual void write(const Lines&,
```

```
    const Ice::Current& = Ice::Current());
    FileI(const Ice::CommunicatorPtr&,
          const std::string&,
          const DirectoryIPtr&);
  private:
    Lines _lines;
  };

  typedef IceUtil::Handle<FileI> FileIPtr;

  class DirectoryI : virtual public Directory,
                     virtual public NodeI {
  public:
    virtual NodeSeq list(const Ice::Current&);
    DirectoryI(const Ice::CommunicatorPtr&,
               const std::string&,
               const DirectoryIPtr&);
    void addChild(const Filesystem::NodePrx&);
  private:
    Filesystem::NodeSeq _contents;
  };
}
```

9.2.3 The Servant Implementation

The implementation of our servants is mostly trivial, following from the class definitions in our `FilesystemI.h` header file.

Implementing `FileI`

The implementation of the `read` and `write` operations for files is trivial: we simply store the passed file contents in the `_lines` data member. The constructor is equally trivial, simply passing its arguments through to the `NodeI` base class constructor:

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&)
{
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
```

```
    _lines = text;
}

Filesystem::FileI::FileI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent
                        ) : NodeI(communicator, name, parent)
{
}
```

Implementing `DirectoryI`

The implementation of `DirectoryI` is equally trivial: the `list` operation simply returns the `_contents` data member and the constructor passes its arguments through to the `NodeI` base class constructor:

```
Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current&)
{
    return _contents;
}

Filesystem::DirectoryI::DirectoryI(
                const Ice::CommunicatorPtr& communicator,
                const string& name,
                const DirectoryIPtr& parent
            ) : NodeI(name, parent)
{
}

void
Filesystem::DirectoryI::addChild(const NodePrx child)
{
    _contents.push_back(child);
}
```

The only noteworthy thing is the implementation of `addChild`: when a new directory or file is created, the constructor of the `NodeI` base class calls `addChild` on its own parent, passing it the proxy to the newly-created child. The implementation of `addChild` appends the passed reference to the contents list of the directory it is invoked on (which is the parent directory).

Implementing `NodeI`

The name operation of our `NodeI` class is again trivial: it simply returns the `_name` data member:

```
std::string
Filesystem::NodeI::name(const Ice::Current&)
{
    return _name;
}
```

The `NodeI` constructor creates an identity for the servant:

```
Filesystem::NodeI::NodeI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent)
    : _name(name), _parent(parent)
{
    _id.name = parent ? IceUtil::generateUUID() : "RootDir";
}
```

For the root directory, we use the fixed identity `"RootDir"`. This allows the client to create a proxy for the root directory (see Section 7.2). For directories other than the root directory, we use a UUID as the identity (see page 298).

Finally, `NodeI` provides the `activate` member function that adds the servant to the ASM and connects the child node to its parent directory:

```
void
Filesystem::NodeI::activate(const Ice::ObjectAdapterPtr& a)
{
    NodePrx thisNode = NodePrx::uncheckedCast(a->add(this, _id));
    if(_parent)
    {
        _parent->addChild(thisNode);
    }
}
```

This completes our servant implementation. The complete source code is shown here once more:

```
#include <IceUtil/IceUtil.h>
#include <FilesystemI.h>

using namespace std;

// Slice Node::name() operation

std::string
Filesystem::NodeI::name(const Ice::Current&)
{
    return _name;
}
```

```
// NodeI constructor

Filesystem::NodeI::NodeI(const Ice::CommunicatorPtr& communicator,
                         const string& name,
                         const DirectoryIPtr& parent)
    : _name(name), _parent(parent)
{
    // Create an identity. The root directory has the fixed identi
ty "RootDir"
    //
    _id.name = parent ? IceUtil::generateUUID() : "RootDir";
}

// NodeI activate() member function

void
Filesystem::NodeI::activate(const Ice::ObjectAdapterPtr& a)
{
    NodePrx thisNode = NodePrx::uncheckedCast(a->add(this, _id));
    if(_parent)
    {
        _parent->addChild(thisNode);
    }
}

// Slice File::read() operation

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&)
{
    return _lines;
}

// Slice File::write() operation

void
Filesystem::FileI::write(const Filesystem::Lines& text, const Ice:
:Current&)
{
    _lines = text;
}

// FileI constructor

Filesystem::FileI::FileI(const Ice::CommunicatorPtr& communicator,
```

```
                        const string& name,
                        const DirectoryIPtr& parent)
    : NodeI(communicator, name, parent)
{
}

// Slice Directory::list() operation

Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current& c)
{
    return _contents;
}

// DirectoryI constructor

Filesystem::DirectoryI::DirectoryI(
                    const Ice::CommunicatorPtr& communicator,
                    const string& name,
                    const DirectoryIPtr& parent)
    : NodeI(communicator, name, parent)
{
}

// addChild is called by the child in order to add
// itself to the _contents member of the parent

void
Filesystem::DirectoryI::addChild(const NodePrx& child)
{
    _contents.push_back(child);
}
```

9.3 Summary

This chapter showed how to implement a complete server for the file system we defined in Chapter 5. Note that the server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

Note that the server code we presented here is not quite correct as it stands: if two clients access the same file in parallel, each via a different thread, one thread may read the `_lines` data member while another thread updates it. Obviously, if that happens, we may write or return garbage or, worse, crash the server. However, it is trivial to make the `read` and `write` operations thread-safe: a single data member and two lines of source code are sufficient to achieve this. We discuss how to write thread-safe servant implementations in Chapter 31.

Part IV

Java Mapping

Chapter 10
Client-Side Slice-to-Java Mapping

10.1 Chapter Overview

In this chapter, we present the client-side Slice-to-Java mapping (see Chapter 12 for the server-side mapping). One part of the client-side Java mapping concerns itself with rules for representing each Slice data type as a corresponding Java type; we cover these rules in Section 10.3 to Section 10.10. Another part of the mapping deals with how clients can invoke operations, pass and receive parameters, and handle exceptions. These topics are covered in Section 10.11 to Section 10.13. Slice classes have the characteristics of both data types and interfaces and are covered in Section 10.14. In Section 10.16, we show how you can customize the Slice-to-Java mapping using metadata. Section 10.17 describes asynchronous method invocation. Section 10.18 lists the command-line options for the Slice-to-Java compiler and describes how to use the Slice compiler in ant projects. Finally, Section 10.19 covers the use of Slice checksums in the Java mapping.

10.2 Introduction

The client-side Slice-to-Java mapping defines how Slice data types are translated to Java types, and how clients invoke operations, pass parameters, and handle errors. Much of the Java mapping is intuitive. For example, Slice sequences map

to Java arrays, so there is essentially nothing new you have to learn in order to use Slice sequences in Java.

The Java API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data—the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least Section 10.9 to Section 10.13 in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

A word of advice before you start: in order to use the Java mapping, you should need no more than the Slice definition of your application and knowledge of the Java mapping rules. In particular, looking through the generated code in order to discern how to use the Java mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

10.3 Mapping for Identifiers

Slice identifiers map to an identical Java identifier. For example, the Slice identifier `Clock` becomes the Java identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a Java keyword or is an identifier reserved by the Ice run time (such as `checkedCast`), the corresponding Java identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.[1]

A single Slice identifier often results in several Java identifiers. For example, for a Slice interface named `Foo`, the generated Java code uses the identifiers `Foo` and `FooPrx` (among others). If the interface has the name `while`, the generated

1. As suggested in Section 4.5.3 on page 92, you should try to avoid such identifiers as much as possible.

identifiers are `_while` and `whilePrx` (*not* `_whilePrx`), that is, the underscore prefix is applied only to those generated identifiers that actually require it.

10.4 Mapping for Modules

Slice modules map to Java packages with the same name as the Slice module. The mapping preserves the nesting of the Slice definitions. For example:

```
// Definitions at global scope here...

module M1 {
    // Definitions for M1 here...
    module M2 {
        // Definitions for M2 here...
    };
};

// ...

module M1 {     // Reopen M1
    // More definitions for M1 here...
};
```

This definition maps to the corresponding Java definitions:

```
package M1;
// Definitions for M1 here...

package M1.M2;
// Definitions for M2 here...

package M1;
// Definitions for M1 here...
```

Note that these definitions appear in the appropriate source files; source files for definitions in module `M1` are generated in directory `M1` underneath the top-level directory, and source files for definitions for module `M2` are generated in directory `M1/M2` underneath the top-level directory. You can set the top-level output directory using the **`--output-dir`** option with **`slice2java`** (see Section 4.20).

10.5 The `Ice` Package

All of the APIs for the Ice run time are nested in the `Ice` package, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` package are generated from Slice definitions; other parts of the `Ice` package provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` package throughout the remainder of the book.

10.6 Mapping for Simple Built-in Types

The Slice built-in types are mapped to Java types as shown in Table 10.1.

Table 10.1. Mapping of Slice built-in types to Java.

Slice	Java
`bool`	`boolean`
`byte`	`byte`
`short`	`short`
`int`	`int`
`long`	`long`
`float`	`float`
`double`	`double`
`string`	`String`

10.7 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

10.7.1 Mapping for Enumerations

A Slice `enum` type maps to the Java `enum` type. Consider the following example:

```
enum Fruit { Apple, Pear, Orange };
```

The Java mapping for `Fruit` is shown below:

```
public enum Fruit implements java.io.Serializable {
    Apple,
    Pear,
    Orange;

    // ...
}
```

Given the above definitions, we can use enumerated values as follows:

```
Fruit favoriteFruit = Fruit.Apple;
Fruit otherFavoriteFruit = Fruit.Orange;

if (favoriteFruit == Fruit.Apple) // Compare with constant
    // ...

if (f1 == f2)                     // Compare two enums
    // ...

switch (f2) {             // Switch on enum
case Fruit.Apple:
    // ...
    break;
case Fruit.Pear
    // ...
    break;
case Fruit.Orange
    // ...
    break;
}
```

Note that the generated class contains a number of other members, which we have not shown. These members are internal to the Ice run time and you must not use them in your application code (because they may change from release to release).

10.7.2 Mapping for Structures

Slice structures map to Java classes with the same name. For each Slice data member, the Java class contains a corresponding public data member. For example, here is our `Employee` structure from Section 4.9.4 once more:

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The Slice-to-Java compiler generates the following definition for this structure:

```
public final class Employee implements java.lang.Cloneable,
                                       java.io.Serializable {
    public long number;
    public String firstName;
    public String lastName;

    public Employee {}

    public Employee(long number,
                    String firstName,
                    String lastName) {
        this.number = number;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public boolean equals(java.lang.Object rhs) {
        // ...
    }
    public int hashCode() {
        // ...
    }

    public java.lang.Object clone()
        java.lang.Object o;
        try
        {
            o = super.clone();
        }
        catch(java.lang.CloneNotSupportedException ex)
        {
            assert false; // impossible
```

```
        }
        return o;
    }
}
```

For each data member in the Slice definition, the Java class contains a corresponding public data member of the same name. Refer to Section 10.16.3 for additional information on data members.

The `equals` member function compares two structures for equality. Note that the generated class also provides the usual `hashCode` and `clone` methods. (`clone` has the default behavior of making a shallow copy.)

Constructors

Structures have a default constructor that default-constructs each data member. This means members of primitive type are initialized to the equivalent of zero, and members of reference type are initialized to null. Note that applications must always explicitly initialize members of structure and enumerated types because the Ice run time does not accept null as a legal value for these types.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition (see Section 4.9.2). The default constructor initializes each of these data members to its declared value.

Structures also have a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement (instead of first having to construct the instance and then assigning to its members).

10.7.3 Mapping for Sequences

Slice sequences map to Java arrays. This means that the Slice-to-Java compiler does not generate a separate named type for a Slice sequence. For example:

```
sequence<Fruit> FruitPlatter;
```

This definition simply corresponds to the Java type `Fruit[]`. Naturally, because Slice sequences are mapped to Java arrays, you can take advantage of all the array functionality provided by Java, such as initialization, assignment, cloning, and the `length` member. For example:

```
Fruit[] platter = { Fruit.Apple, Fruit.Pear };
assert(platter.length == 2);
```

See Section 10.16 for information on alternate mappings for sequence types.

10.7.4 Mapping for Dictionaries

Here is the definition of our `EmployeeMap` from Section 4.9.4 once more:

```
dictionary<long, Employee> EmployeeMap;
```

As for sequences, the Java mapping does not create a separate named type for this definition. Instead, the dictionary is simply an instance of the generic type `java.util.Map<K, V>`, where *K* is the mapping of the key type and *V* is the mapping of the value type. In the example above, `EmployeeMap` is mapped to the Java type `java.util.Map<Long, Employee>`. The following code demonstrates how to allocate and use an instance of `EmployeeMap`:

```
java.util.Map<Long, Employee> em =
    new java.util.HashMap<Long, Employee>();

Employee e = new Employee();
e.number = 31;
e.firstName = "James";
e.lastName = "Gosling";

em.put(e.number, e);
```

The typesafe nature of the mapping makes iterating over the dictionary quite convenient:

```
for (java.util.Map.Entry<Long, Employee> i : em.entrySet()) {
    long num = i.getKey();
    Employee emp = i.getValue();
    System.out.println(emp.firstName + " was employee #" + num);
}
```

See Section 10.16 for information on alternate mappings for dictionary types.

10.8 Mapping for Constants

Here are the constant definitions we saw in Section 4.9.5 on page 103 once more:

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
```

```
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

```
public interface AppendByDefault {
    boolean value = true;
}

public interface LowerNibble {
    byte value = 15;
}

public interface Advice {
    String value = "Don't Panic!";
}

public interface TheAnswer {
    short value = 42;
}

public interface PI {
    double value = 3.1416;
}

public interface FavoriteFruit {
    Fruit value = Fruit.Pear;
}
```

As you can see, each Slice constant is mapped to a Java interface with the same name as the constant. The interface contains a member named `value` that holds the value of the constant.

10.9 Mapping for Exceptions

Here is a fragment of the Slice definition for our world time server from Section 4.10.5 on page 120 once more:

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

These exception definitions map as follows:

```
public class GenericError extends Ice.UserException {
    public String reason;

    public GenericError() {}

    public GenericError(String reason)
    {
        this.reason = reason;
    }

    public String ice_name() {
        return "GenericError";
    }
}

public class BadTimeVal extends GenericError {
    public BadTimeVal() {}

    public BadTimeVal(String reason)
    {
        super(reason);
    }

    public String ice_name() {
        return "BadTimeVal";
    }
}

public class BadZoneName extends GenericError {
    public BadZoneName() {}

    public BadZoneName(String reason)
    {
        super(reason);
    }
```

```
    public String ice_name() {
        return "BadZoneName";
    }
}
```

Each Slice exception is mapped to a Java class with the same name. For each data member, the corresponding class contains a public data member. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.) Refer to Section 10.16.3 for additional information on data members.

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception also defines the `ice_name` member function, which returns the name of the exception.

All user exceptions are derived from the base class `Ice.UserException`. This allows you to catch all user exceptions generically by installing a handler for `Ice.UserException`. `Ice.UserException`, in turn, derives from `java.lang.Exception`.

`Ice.UserException` implements a `clone` method that is inherited by its derived exceptions, so you can make memberwise shallow copies of exceptions.

Note that the generated exception classes contain other member functions that are not shown. However, those member functions are internal to the Java mapping and are not meant to be called by application code.

Constructors

Exceptions have a default constructor that default-constructs each data member. This means members of primitive type are initialized to the equivalent of zero, and members of reference type are initialized to null. Note that applications must always explicitly initialize members of structure and enumerated types because the Ice run time does not accept null as a legal value for these types.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition (see Section 4.10.2). The default constructor initializes each of these data members to its declared value.

Exceptions also have a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement (instead of first having to construct the instance and then assigning to its members). For derived exceptions, this constructor accepts one

argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

10.10 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice.LocalException` (which, in turn, derives from `java.lang.RuntimeException`).

`Ice.LocalExceptions` implements a `clone` method that is inherited by its derived exceptions, so you can make memberwise shallow copies of exceptions.

An inheritance diagram for user and run-time exceptions appears in Figure 4.4 on page 117. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice.LocalException`

  This is the root of the inheritance tree for run-time exceptions.

- `Ice.UserException`

  This is the root of the inheritance tree for user exceptions.

- `Ice.TimeoutException`

  This is the base exception for both operation-invocation and connection-establishment timeouts.

- `Ice.ConnectTimeoutException`

  This exception is raised when the initial attempt to establish a connection to a server times out.

You will probably have little need to catch the remaining run-time exceptions; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. However, there is one exception you will probably be interested in specifically: `Ice.ObjectNotExistException`. This exception is raised if a client invokes an operation on an Ice object that no longer exists. In other words, the client holds a dangling reference to an object that probably existed some time in the past but has since been permanently destroyed.

10.11 Mapping for Interfaces

Slice interfaces map to proxies on the client side. A proxy is simply a Java interface with operations that correspond to the operations defined in the Slice interface.

The compiler generates quite few source files for each Slice interface. In general, for an interface *<interface-name>*, the following source files are created by the compiler:

- *<interface-name>*`.java`

  This source file declares the *<interface-name>* Java interface.

- *<interface-name>*`Holder.java`

  This source file defines a holder type for the interface (see page 353).

- *<interface-name>*`Prx.java`

  This source file defines the *<interface-name>*`Prx` interface (see page 342).

- *<interface-name>*`PrxHelper.java`

  This source file defines the helper type for the interface's proxy (see page 345).

- *<interface-name>*`PrxHolder.java`

  This source file defines the holder type for the interface's proxy (see page 353).

- `_`*<interface-name>*`Operations.java`
  `_`*<interface-name>*`OperationsNC.java`

  These source files each define an interface that contains the operations corresponding to the Slice interface.

These are the files that contain code that is relevant to the client side. The compiler also generates a file that is specific to the server side, plus three additional files:

- `_`*<interface-name>*`Disp.java`

  This file contains the definition of the server-side skeleton class.

- `_`*<interface-name>*`Del.java`
- `_`*<interface-name>*`DelD.java`

- _<interface-name>DelM.java

  These files contain code that is internal to the Java mapping; they do not contain any functions of relevance to application programmers.

10.11.1 Proxy Interfaces

On the client side, Slice interfaces map to Java interfaces with member functions that correspond to the operations on those interfaces. Consider the following simple interface:

```
interface Simple {
    void op();
};
```

The Slice compiler generates the following definition for use by the client:

```
public interface SimplePrx extends Ice.ObjectPrx {
    public void op();
    public void op(java.util.Map<String, String> __context);
}
```

As you can see, the compiler generates a *proxy interface* SimplePrx. In general, the generated name is <interface-name>Prx. If an interface is nested in a module M, the generated class is part of package M, so the fully-qualified name is M.<interface-name>Prx.

In the client's address space, an instance of SimplePrx is the local ambassador for a remote instance of the Simple interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that SimplePrx inherits from Ice.ObjectPrx. This reflects the fact that all Ice interfaces implicitly inherit from Ice::Object.

For each operation in the interface, the proxy class has a member function of the same name. For the preceding example, we find that the operation op has been mapped to the member function op. Also note that op is overloaded: the second version of op has a parameter __context of type java.util.Map<String, String>. This parameter is for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the __context parameter in detail in Chapter 32. The parameter is also used by IceStorm—see Chapter 44.)

Because all the <interface-name>Prx types are interfaces, you cannot instantiate an object of such a type. Instead, proxy instances are always instanti-

ated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.The proxy references handed out by the Ice run time are always of type `<interface-name>Prx`; the concrete implementation of the interface is part of the Ice run time and does not concern application code.

A value of `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

10.11.2 The `Ice.ObjectPrx` Interface

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice.ObjectPrx`. `ObjectPrx` provides a number of methods:

```
package Ice;

public interface ObjectPrx {
    boolean equals(java.lang.Object r);
    Identity ice_getIdentity();
    boolean ice_isA(String __id);
    boolean ice_isA(String __id,
                    java.util.Map<String, String> ctx);
    String[] ice_ids();
    String[] ice_ids(java.util.Map<String, String> ctx);
    String ice_id();
    String ice_id(java.util.Map<String, String> ctx);
    void ice_ping();
    void ice_ping(java.util.Map<String, String> ctx);
    // ...
}
```

The methods behave as follows:

- `equals`

  This operation compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

- `ice_getIdentity`

  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
};
```

To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

```
Ice.ObjectPrx o1 = ...;
Ice.ObjectPrx o2 = ...;
Ice.Identity i1 = o1.ice_getIdentity();
Ice.Identity i2 = o2.ice_getIdentity();

if (i1.equals(i2))
    // o1 and o2 denote the same object
else
    // o1 and o2 denote different objects
```

- `ice_isA`

  This method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID (see Section 4.13). For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

```
Ice.ObjectPrx o = ...;
if (o != null && o.ice_isA("::Printer"))
    // o denotes a Printer object
else
    // o denotes some other type of object
```

  Note that we are testing whether the proxy is null before attempting to invoke the `ice_isA` method. This avoids getting a `NullPointerException` if the proxy is null.

- `ice_ids`

  This method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.

- `ice_id`

  This method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type

BasePrx, with a static type ID of ::Base, the return value of ice_id might be ::Base, or it might something more derived, such as ::Derived.

- ice_ping

  This method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as ObjectNotExistException or ConnectTimeoutException.

The ice_isA, ice_ids, ice_id, and ice_ping methods are remote operations and therefore support an additional overloading that accepts a request context. Also note that there are other methods in ObjectPrx (not shown here) that provide different ways to dispatch a call. We discuss these topics in Chapter 32.

10.11.3 Proxy Helpers

For each Slice interface, apart from the proxy interface, the Slice-to-Java compiler creates a helper class: for an interface Simple, the name of the generated helper class is SimplePrxHelper. The helper classes contains two methods that support down-casting:

```
public final class SimplePrxHelper
        extends Ice.ObjectPrxHelper implements SimplePrx {
    public static SimplePrx checkedCast(Ice.ObjectPrx b) {
        // ...
    }

    public static SimplePrx checkedCast(Ice.ObjectPrx b,
                                        Ice.Context ctx) {
        // ...
    }

    public static SimplePrx uncheckedCast(Ice.ObjectPrx b) {
        // ...
    }

    // ...
}
```

Both the checkedCast and uncheckedCast methods implement a down-cast: if the passed proxy is a proxy for an object of type Simple, or a proxy for an object with a type derived from Simple, the cast returns a non-null reference to a

proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the cast returns a null reference.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

```
Ice.ObjectPrx obj = ...;        // Get a proxy from somewhere...

SimplePrx simple = SimplePrxHelper.checkedCast(obj);
if (simple != null)
    // Object supports the Simple interface...
else
    // Object is not of type Simple...
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of a proxy in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`. (This also explains the need for the helper class: the Ice run time must contact the server, so we cannot use a Java down-cast.)

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather non-sensical things. To illustrate this, consider the following two interfaces:

```
interface Process {
    void launch(int stackSize, int dataSize);
};

// ...

interface Rocket {
    void launch(float xCoord, float yCoord);
};
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

```
Ice.ObjectPrx obj = ...;                    // Get proxy...
ProcessPrx process
    = ProcessPrxHelper.uncheckedCast(obj);  // No worries...
process.launch(40, 60);                     // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

A final warning about down-casts: you must use either a `checkedCast` or an `uncheckedCast` to down-cast a proxy. If you use a Java cast, the behavior is undefined.

10.11.4 Using Proxy Methods

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy (see Section 32.11). Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second timeout as shown below:

```
Ice.ObjectPrx proxy = communicator.stringToProxy(...);
proxy = proxy.ice_timeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a `checkedCast` or `uncheckedCast` after using a factory method. However, a regular cast is still required, as shown in the example below:

```
Ice.ObjectPrx base = communicator.stringToProxy(...);
HelloPrx hello = HelloPrxHelper.checkedCast(base);
hello = (HelloPrx)hello.ice_timeout(10000); # Type is preserved
hello.sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

10.11.5 Object Identity and Proxy Comparison

Proxies provide an `equals` method that compares proxies:

```
interface ObjectPrx {
    boolean equals(java.lang.Object r);
}
```

Note that proxy comparison with `equals` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `equals` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

```
Ice.ObjectPrx p1 = ...;        // Get a proxy...
Ice.ObjectPrx p2 = ...;        // Get another proxy...

if (p1.equals(p2)) {
    // p1 and p2 denote different objects       // WRONG!
} else {
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `equals`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function in the `Ice.Util` class:

```
package Ice;

public final class Util {
    public static int proxyIdentityCompare(ObjectPrx lhs,
                                           ObjectPrx rhs);
    public static int proxyIdentityAndFacetCompare(ObjectPrx lhs,
                                                   ObjectPrx rhs);
    // ...
}
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

```
Ice.ObjectPrx p1 = ...;        // Get a proxy...
Ice.ObjectPrx p2 = ...;        // Get another proxy...

if (Ice.Util.proxyIdentityCompare(p1, p2) != 0) {
    // p1 and p2 denote different objects       // Correct
} else {
    // p1 and p2 denote the same object         // Correct
}
```

The function returns 0 if the identities are equal, –1 if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name (see Chapter 33).

In addition, the Java mapping provides two wrapper classes that allow you to wrap a proxy for use as the key of a hashed collection:

```
package Ice;

public class ProxyIdentityKey {
    public ProxyIdentityKey(Ice.ObjectPrx proxy);
    public int hashCode();
    public boolean equals(java.lang.Object obj);
    public Ice.ObjectPrx getProxy();
}

public class ProxyIdentityFacetKey {
    public ProxyIdentityFacetKey(Ice.ObjectPrx proxy);
    public int hashCode();
    public boolean equals(java.lang.Object obj);
    public Ice.ObjectPrx getProxy();
}
```

The constructor caches the identity and the hash code of the passed proxy, so calls to `hashCode` and `equals` can be evaluated efficiently. The `getProxy` method returns the proxy that was passed to the constructor.

As for the comparison functions, `ProxyIdentityKey` only uses the proxy's identity, whereas `ProxyIdentityFacetKey` also includes the facet name.

10.11.6 Deserializing Proxies

Proxy objects implement the `java.io.Serializable` interface that enables serialization of proxies to and from a byte stream. You can use the standard class `java.io.ObjectInputStream` to deserialize all Slice types *except* proxies; proxies are a special case because they must be created by a communicator.

To supply a communicator for use in deserializing proxies, an application must use the class `Ice.ObjectInputStream`:

```
package Ice;

public class ObjectInputStream extends java.io.ObjectInputStream
{
    public ObjectInputStream(Communicator communicator,
                             java.io.InputStream stream)
        throws java.io.IOException;

    public Communicator getCommunicator();
}
```

The code shown below demonstrates how to use this class:

```
Ice.Communicator communicator = ...
byte[] bytes = ... // data to be deserialized
java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream(bytes);
Ice.ObjectInputStream in =
    new Ice.ObjectInputStream(communicator, byteStream);
Ice.ObjectPrx proxy = (Ice.ObjectPrx)in.readObject();
```

Ice raises `java.io.IOException` if an application attempts to deserialize a proxy without supplying a communicator.

10.12 Mapping for Operations

As we saw in Section 10.11, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system from Section 5.4:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

```
NodePrx node = ...;             // Initialize proxy
String name = node.name();      // Get name via RPC
```

This illustrates the typical pattern for receiving return values: return values are returned by reference for complex types, and by value for simple types (such as `int` or `double`).

10.12.1 Normal and `idempotent` Operations

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

```
interface Example {
                string op1();
    idempotent  string op2();
};
```

The proxy interface for this is:

```
public interface ExamplePrx extends Ice.ObjectPrx {
    public String op1();
    public String op2();
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to be mapped the same.

10.12.2 Passing Parameters

In-Parameters

The parameter passing rules for the Java mapping are very simple: parameters are passed either by value (for simple types) or by reference (for complex types and type `String`). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation (with some caveats—see page 1093).

Here is an interface with operations that pass parameters of various types from client to server:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following proxy for this definition:

```
public interface ClientToServerPrx extends Ice.ObjectPrx {
    public void op1(int i, float f, boolean b, String s);
    public void op2(NumberAndString ns,
                    String[] ss,
                    java.util.Map st);
    public void op3(ClientToServerPrx proxy);
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

```
ClientToServerPrx p = ...;              // Get proxy...

p.op1(42, 3.14f, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14f;
```

```
boolean b = true;
String s = "Hello world!";
p.op1(i, f, b, s);                      // Pass simple variables

NumberAndString ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
String[] ss = { "Hello world!" };
java.util.HashMap st = new java.util.HashMap();
st.put(new Long(0), ns);
p.op2(ns, ss, st);                      // Pass complex variables

p.op3(p);                               // Pass proxy
```

Out-Parameters

Java does not have pass-by-reference: parameters are always passed by value. For a function to modify one of its arguments, we must pass a reference (by value) to an object; the called function can then modify the object's contents via the passed reference.

To permit the called function to modify a parameter, the Java mapping uses so-called *holder* classes. For example, for each of the built-in Slice types, such as `int` and `string`, the `Ice` package contains a corresponding holder class. Here are the definitions for the holder classes `Ice.IntHolder` and `Ice.StringHolder`:

```
package Ice;

public final class IntHolder {
    public IntHolder() {}
    public IntHolder(int value)
        this.value = value;
    }
    public int value;
}

public final class StringHolder {
    public StringHolder() {}
    public StringHolder(String value) {
        this.value = value;
    }
    public String value;
}
```

A holder class has a public `value` member that stores the value of the parameter; the called function can modify the value by assigning to that member. The class also has a default constructor and a constructor that accepts an initial value.

For user-defined types, such as structures, the Slice-to-Java compiler generates a corresponding holder type. For example, here is the generated holder type for the `NumberAndString` structure we defined on page 352:

```
public final class NumberAndStringHolder {
    public NumberAndStringHolder() {}

    public NumberAndStringHolder(NumberAndString value) {
        this.value = value;
    }

    public NumberAndString value;
}
```

This looks exactly like the holder classes for the built-in types: we get a default constructor, a constructor that accepts an initial value, and the public `value` member.

Note that holder classes are generated for *every* Slice type you define. For example, for sequences, such as the `FruitPlatter` sequence we saw on page 335, the compiler does not generate a special Java `FruitPlatter` type because sequences map to Java arrays. However, the compiler *does* generate a `FruitPlatterHolder` class, so we can pass a `FruitPlatter` array as an out-parameter.

To pass an out-parameter to an operation, we simply pass an instance of a holder class and examine the `value` member of each out-parameter when the call completes. Here is the same Slice definition we saw on page 352 once more, but this time with all parameters being passed in the `out` direction:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
```

```
            out StringSeq ss,
            out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Slice compiler generates the following code for this definition:

```
public interface ClientToServerPrx extends Ice.ObjectPrx {
    public void op1(Ice.IntHolder i, Ice.FloatHolder f,
                    Ice.BooleanHolder b, Ice.StringHolder s);
    public void op2(NumberAndStringHolder ns,
                    StringSeqHolder ss, StringTableHolder st);
    public void op3(ClientToServerPrxHolder proxy);
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

```
ClientToServerPrx p = ...;              // Get proxy...

Ice.IntHolder ih = new Ice.IntHolder();
Ice.FloatHolder fh = new Ice.FloatHolder();
Ice.BooleanHolder bh = new Ice.BooleanHolder();
Ice.StringHolder sh = new Ice.StringHolder();
p.op1(ih, fh, bh, sh);

NumberAndStringHolder nsh = new NumberAndString();
StringSeqHolder ssh = new StringSeqHolder();
StringTableHolder sth = new StringTableHolder();
p.op2(nsh, ssh, sth);

ServerToClientPrxHolder stcph = new ServerToClientPrxHolder();
p.op3(stch);

System.out.writeln(ih.value);   // Show one of the values
```

Again, there are no surprises in this code: the various holder instances contain values once the operation invocation completes and the `value` member of each instance provides access to those values.

Null Parameters

Some Slice types naturally have “empty” or “not there” semantics. Specifically, sequences, dictionaries, and strings all can be `null`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `null` as a parameter or return value of type sequence,

dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid `NullPointerException`s. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as `null` or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

10.13 Exception Handling

Any operation invocation may throw a run-time exception (see Section 10.10 on page 340) and, if the operation has an exception specification, may also throw user exceptions (see Section 10.9 on page 337). Suppose we have the following simple interface:

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as Java exceptions, so you can simply enclose one or more operation invocations in a `try–catch` block:

```
ChildPrx child = ...;   // Get child proxy...

try {
    child.askToCleanUp();
} catch (Tantrum t) {
    System.out.write("The child says: ");
    System.out.writeln(t.reason);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

```
public class Client {
    static void run() {
        ChildPrx child = ...;   // Get child proxy...
        try {
            child.askToCleanUp();
        } catch (Tantrum t) {
            System.out.print("The child says: ");
            System.out.println(t.reason);
            child.scold();          // Recover from error...
        }
        child.praise();             // Give positive feedback...
    }

    public static void
    main(String[] args)
    {
        try {
            // ...
            run();
            // ...
        } catch (Ice.LocalException e) {
            e.printStackTrace();
        } catch (Ice.UserException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Chapter 3.)

Exceptions and Out-Parameters

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out-parameters, Ice provides the weak exception guarantee [21] but does not provide the strong exception guarantee.[2]

10.14 Mapping for Classes

Slice classes are mapped to Java classes with the same name. The generated class contains a public data member for each Slice data member (just as for structures and exceptions), and a member function for each operation. Consider the following class definition:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler generates the following code for this definition:

```
public interface _TimeOfDayOperations {
    String format(Ice.Current current);
}

public interface _TimeOfDayOperationsNC {
    String format();
}

public abstract class TimeOfDay extends Ice.ObjectImpl
                                implements _TimeOfDayOperations,
                                           _TimeOfDayOperationsNC
{
    public short hour;
    public short minute;
    public short second;

    public TimeOfDay();
    public TimeOfDay(short hour, short minute, short second);
    // ...
}
```

There are a number of things to note about the generated code:

2. This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

1. The compiler generates "operations interfaces" called `_TimeOfDayOperations` and `_TimeOfDayOperationsNC`. These interfaces contain a method for each Slice operation of the class.
2. The generated class `TimeOfDay` inherits (indirectly) from `Ice.Object`. This means that all classes implicitly inherit from `Ice.Object`, which is the ultimate ancestor of all classes. Note that `Ice.Object` is *not* the same as `Ice.ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.

   If a class has only data members, but no operations, the compiler generates a non-abstract class.
3. The generated class contains a public member for each Slice data member.
4. The generated class inherits member functions for each Slice operation from the operations interfaces.
5. The generated class contains two constructors.

There is quite a bit to discuss here, so we will look at each item in turn.

10.14.1 Operations Interfaces

The methods in the `_<interface-name>Operations` interface have an additional trailing parameter of type `Ice.Current`, whereas the methods in the `_<interface-name>OperationsNC` interface lack this additional trailing parameter. The methods without the `Current` parameter simply forward to the methods with a `Current` parameter, supplying a default `Current`. For now, you can ignore this parameter and pretend it does not exist. (We look at it in more detail in Section 32.6.)

If a class has only data members, but no operations, the compiler omits generating the `_<interface-name>Operations` and `_<interface-name>OperationsNC` interfaces.

10.14.2 Inheritance from `Ice.Object`

Like interfaces, classes implicitly inherit from a common base class, `Ice.Object`. However, as shown in Figure 10.1, classes inherit from `Ice.Object` instead of `Ice.ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is

expected (and vice versa) because the base types for classes and proxies are not compatible.

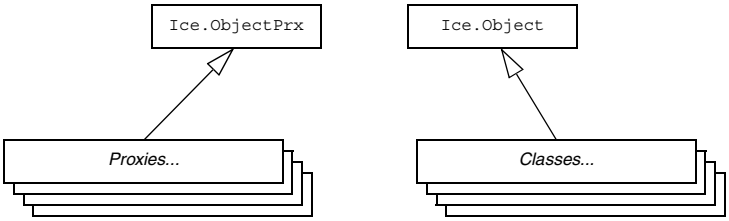


Figure 10.1. Inheritance from `Ice.ObjectPrx` and `Ice.Object`.

`Ice.Object` contains a number of member functions:

```
package Ice;

public interface Object
{
    boolean ice_isA(String s);
    boolean ice_isA(String s, Current current);

    void ice_ping();
    void ice_ping(Current current);

    String[] ice_ids();
    String[] ice_ids(Current current);

    String ice_id();
    String ice_id(Current current);

    void ice_preMarshal();
    void ice_postUnmarshal();

    DispatchStatus ice_dispatch(
        Request request,
        DispatchInterceptorAsyncCallback cb);
}
```

The member functions of `Ice.Object` behave as follows:

- `ice_isA`

  This function returns `true` if the object supports the given type ID, and `false` otherwise.

- `ice_ping`

  As for interfaces, `ice_ping` provides a basic reachability test for the class.
- `ice_ids`

  This function returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.
- `ice_id`

  This function returns the actual run-time type ID for a class. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.
- `ice_preMarshal`

  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`

  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_dispatch`

  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors (see Section 32.23).

Note that the generated class does *not* override `hashCode` and `equals`. This means that classes are compared using shallow reference equality, not value equality (as is used for structures).

All Slice classes derive from `Ice.Object` via the `Ice.ObjectImpl` abstract base class. `ObjectImpl` implements the `java.io.Serializable` interface to support Java's serialization facility (see Section 10.15). `ObjectImpl` also supplies an implementation of `clone` that returns a shallow member-wise copy.

10.14.3 Data Members of Classes

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result,

the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

```
class TimeOfDay {
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler produces the following generated code for this definition:

```
public abstract class TimeOfDay extends Ice.ObjectImpl
                                implements _TimeOfDayOperations,
                                           _TimeOfDayOperationsNC
{
    protected short hour;
    protected short minute;
    protected short second;

    public TimeOfDay();
    public TimeOfDay(short hour, short minute, short second);
    // ...
}
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

```
["protected"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

Refer to Section 10.16.3 for additional information on data members.

10.14.4 Operations of Classes

Operations of classes are mapped to abstract member functions in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

```
public class TimeOfDayI extends TimeOfDay {
    public String format(Ice.Current current) {
        DecimalFormat df
            = (DecimalFormat)DecimalFormat.getInstance();
        df.setMinimumIntegerDigits(2);
        return new String(df.format(hour) + ":" +
                          df.format(minute) + ":" +
                          df.format(second));
    }
}
```

Class Factories

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

```
class ObjectFactory implements Ice.ObjectFactory {
    public Ice.Object create(String type) {
        if (type.equals(M.TimeOfDay.ice_staticId())) {
            return new TimeOfDayI();
        }
        assert(false);
        return null;
    }

    public void destroy() {
        // Nothing to do
    }
}
```

The object factory's `create` method is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` method is called by the Ice run time when its communicator is destroyed.

The `create` method is passed the type ID (see Section 4.13) of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::M::TimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `TimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoObjectFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

```
Ice.Communicator ic = ...;
ic.addObjectFactory(new ObjectFactory(),
                    M.TimeOfDay.ice_staticId());
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::M::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator—if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called. However, calls to `create` can be made concurrently.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

10.14.5 Class Constructors

Classes have a default constructor that default-constructs each data member. This means members of primitive type are initialized to the equivalent of zero, and members of reference type are initialized to null. Note that applications must always explicitly initialize members of structure and enumerated types because the Ice run time does not accept null as a legal value for these types.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition (see Section 4.11.1). The default constructor initializes each of these data members to its declared value.

The generated class also contains a second constructor that accepts one argument for each member of the class. This allows you to create and initialize a class in a single statement, for example:

```
TimeOfDayI tod = new TimeOfDayI(14, 45, 00); // 14:45pm
```

For derived classes, the constructor requires an argument for every member of the class, including inherited members. For example, consider the the definition from Section 4.11.2 once more:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

class DateTime extends TimeOfDay {
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
};
```

The constructors for the generated classes are as follows:

```
public class TimeOfDay extends Ice.ObjectImpl {
    public TimeOfDay() {}

    public TimeOfDay(short hour, short minute, short second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // ...
}

public class DateTime extends TimeOfDay
{
    public DateTime()
    {
        super();
    }

    public DateTime(short hour, short minute, short second,
                    short day, short month, short year)
    {
        super(hour, minute, second);
        this.day = day;
        this.month = month;
        this.year = year;
    }

    // ...
}
```

If you want to instantiate and initialize a `DateTime` instance, you must either use the default constructor or provide values for all of the data members of the instance, including data members of any base classes.

10.15 Serializable Objects

In Java terminology, a *serializable object* typically refers to an object that implements the `java.io.Serializable` interface and therefore supports serialization to and from a byte stream. All Java classes generated from Slice definitions implement the `java.io.Serializable` interface.

In addition to serializing Slice types, applications may also need to incorporate foreign types into their Slice definitions. As discussed in Section 4.18, you can pass Java serializable objects directly as operation parameters or as fields of another data type. For example:

```
["java:serializable:SomePackage.JavaClass"]
sequence<byte> JavaObj;
struct MyStruct {
    int i;
    JavaObj o;
};

interface Example {
    void op(JavaObj inObj, MyStruct s, out JavaObj outObj);
};
```

The generated code for `MyStruct` contains a member `i` of type `int` and a member `o` of type `SomePackage.JavaClass`:

```
public final class MyStruct implements java.lang.Cloneable {
    public int i;
    public SomePackage.JavaClass o;

    // ...
}
```

Similarly, the signature for `op` has parameters of type `JavaClass` and `MyStruct` for the in-parameters, and `Ice.Holder<SomePackage.JavaClass>` for the out-parameter. (Out-parameters are always passed as `Ice.Holder<`*`class`*`>`.)

```
void op(SomePackage.JavaClass inObj,
        MyStruct s,
        Ice.Holder<SomePackage.JavaClass> outObj);
```

Of course, your client and server code must have an implementation of `JavaClass` that derives from `java.io.Serializable`:

```
package SomePackage;

public class JavaClass implements java.io.Serializable {
    // ...
}
```

You can implement this class in any way you see fit—the Ice run time does not place any other requirements on the implementation.

10.16 Customizing the Java Mapping

You can customize the code that the Slice-to-Java compiler produces by annotating your Slice definitions with metadata (see Section 4.17). This section describes how metadata influences several aspects of the generated Java code.

10.16.1 Packages

By default, the scope of a Slice definition determines the package of its mapped Java construct. A Slice type defined in a module hierarchy is mapped to a type residing in the equivalent Java package (see Section 10.4 for more information on the module mapping).

There are times when applications require greater control over the packaging of generated Java classes. For instance, a company may have software development guidelines that require all Java classes to reside in a designated package. One way to satisfy this requirement is to modify the Slice module hierarchy so that the generated code uses the required package by default. In the example below, we have enclosed the original definition of `Workflow::Document` in the modules `com::acme` so that the compiler will create the class in the `com.acme` package:

```
module com {
    module acme {
        module Workflow {
            class Document {
                // ...
```

```
            };
        };
    };
};
```

There are two problems with this workaround:

1. It incorporates the requirements of an implementation language into the application's interface specification.
2. Developers using other languages, such as C++, are also affected.

The Slice-to-Java compiler provides a better way to control the packages of generated code through the use of global metadata (see Section 4.17). The example above can be converted as follows:

```
[["java:package:com.acme"]]
module Workflow {
    class Document {
        // ...
    };
};
```

The global metadata directive `java:package:com.acme` instructs the compiler to generate all of the classes resulting from definitions in this Slice file into the Java package `com.acme`. The net effect is the same: the class for `Document` is generated in the package `com.acme.Workflow`. However, we have addressed the two shortcomings of the first solution by reducing our impact on the interface specification: the Slice-to-Java compiler recognizes the package metadata directive and modifies its actions accordingly, whereas the compilers for other language mappings simply ignore it.

Package Configuration Properties

Using global metadata to alter the default package of generated classes has ramifications for the Ice run time when unmarshaling exceptions and concrete class types. The Ice run time dynamically loads generated classes by translating their Slice type ids into Java class names. For example, the Ice run time translates the Slice type id `::Workflow::Document` into the class name `Workflow.Document`.

However, when the generated classes are placed in a user-specified package, the Ice run time can no longer rely on the direct translation of a Slice type id into a Java class name, and therefore must be configured in order to successfully locate the generated classes. Two configuration properties are supported:

- `Ice.Package.`*`Module=package`*

  Associates a top-level[3] Slice module with the package in which it was generated.

- `Ice.Default.Package=`*`package`*

  Specifies a default package to use if other attempts to load a class have failed.

The behavior of the Ice run time when unmarshaling an exception or concrete class is described below:

1. Translate the Slice type id into a Java class name and attempt to load the class.
2. If that fails, extract the top-level module from the type id and check for an `Ice.Package` property with a matching module name. If found, prepend the specified package to the class name and try to load the class again.
3. If that fails, check for the presence of `Ice.Default.Package`. If found, prepend the specified package to the class name and try to load the class again.
4. If the class still cannot be loaded, the instance may be sliced according to the rules described in Section 37.2.11.

Continuing our example from the previous section, we can define the following property:

```
Ice.Package.Workflow=com.acme
```

Alternatively, we could achieve the same result with this property:

```
Ice.Default.Package=com.acme
```

10.16.2 Custom Types

One of the more powerful applications of metadata is the ability to tailor the Java mapping for sequence and dictionary types to match the needs of your application.

Metadata

The metadata for specifying a custom type has the following format:

```
java:type:instance-type[:formal-type]
```

3. Only top-level module names are allowed; the semantics of global metadata prevent a nested module from being generated into a different package than its enclosing module.

The formal type is optional; the compiler uses a default value if one is not defined. The instance type must satisfy an *is-A* relationship with the formal type: either the same class is specified for both types, or the instance type must be derived from the formal type.

The Slice-to-Java compiler generates code that uses the formal type for all occurrences of the modified Slice definition except when the generated code must instantiate the type, in which case the compiler uses the instance type instead.

The compiler performs no validation on your custom types. Misspellings and other errors will not be apparent until you compile the generated code.

Defining a Custom Sequence Type

Although the default mapping of a sequence type to a native Java array is efficient and typesafe, it is not always the most convenient representation of your data. To use a different representation, specify the type information in a metadata directive, as shown in the following example:

```
["java:type:java.util.LinkedList<String>"]
sequence<string> StringList;
```

It is your responsibility to use a type parameter for the Java class (`String` in the example above) that is the correct mapping for the sequence's element type.

The compiler requires the formal type to implement `java.util.List<`*E*`>`, where *E* is the Java mapping of the element type. If you do not specify a formal type, the compiler uses `java.util.List<`*E*`>` by default.

Note that extra care must be taken when defining custom types that contain nested generic types, such as a custom sequence whose element type is also a custom sequence. The Java compiler strictly enforces type safety, therefore any compatibility issues in the custom type metadata will be apparent when the generated code is compiled.

Defining a Custom Dictionary Type

The default instance type for a dictionary is `java.util.HashMap<`*K*`, `*V*`>`, where *K* is the Java mapping of the key type and *V* is the Java mapping of the value type. If the semantics of a `HashMap` are not suitable for your application, you can specify an alternate type using metadata as shown in the example below:

```
["java:type:java.util.TreeMap<String, String>"]
dictionary<string, string> StringMap;
```

It is your responsibility to use type parameters for the Java class (`String` in the example above) that are the correct mappings for the dictionary's key and value types.

The compiler requires the formal type to implement `java.util.Map<K, V>`. If you do not specify a formal type, the compiler uses this type by default.

Note that extra care must be taken when defining dictionary types that contain nested generic types, such as a dictionary whose element type is a custom sequence. The Java compiler strictly enforces type safety, therefore any compatibility issues in the custom type metadata will be apparent when the generated code is compiled.

Usage

You can define custom type metadata in a variety of situations. The simplest scenario is specifying the metadata at the point of definition:

```
["java:type:java.util.LinkedList<String>"]
sequence<string> StringList;
```

Defined in this manner, the Slice-to-Java compiler uses `java.util.List<String>` (the default formal type) for all occurrences of `StringList`, and `java.util.LinkedList<String>` when it needs to instantiate `StringList`.

You may also specify a custom type more selectively by defining metadata for a data member, parameter or return value. For instance, the mapping for the original Slice definition might be sufficient in most situations, but a different mapping is more convenient in particular cases. The example below demonstrates how to override the sequence mapping for the data member of a structure as well as for several operations:

```
sequence<string> StringSeq;

struct S {
    ["java:type:java.util.LinkedList<String>"] StringSeq seq;
};

interface I {
    ["java:type:java.util.ArrayList<String>"] StringSeq
    modifiedReturnValue();

    void modifiedInParam(
        ["java:type:java.util.ArrayList<String>"] StringSeq seq);
```

```
    void modifiedOutParam(
        out ["java:type:java.util.ArrayList<String>"]
        StringSeq seq);
};
```

As you might expect, modifying the mapping for an operation's parameters or return value may require the application to manually convert values from the original mapping to the modified mapping. For example, suppose we want to invoke the `modifiedInParam` operation. The signature of its proxy operation is shown below:

```
void modifiedInParam(java.util.List<String> seq, Ice.Current curr)
```

The metadata changes the mapping of the `seq` parameter to `java.util.List`, which is the default formal type. If a caller has a `StringSeq` value in the original mapping, it must convert the array as shown in the following example:

```
String[] seq = new String[2];
seq[0] = "hi";
seq[1] = "there";
IPrx proxy = ...;
proxy.modifiedInParam(java.util.Arrays.asList(seq));
```

Although we specified the instance type `java.util.ArrayList<String>` for the parameter, we are still able to pass the result of `asList` because its return type (`java.util.List<String>`) is compatible with the parameter's formal type declared by the proxy method. In the case of an operation parameter, the instance type is only relevant to a servant implementation, which may need to make assumptions about the actual type of the parameter.

Mapping for Modified Out Parameters

The mapping for an `out` parameter uses a generated "holder" class to convey the parameter value (see Section 10.12.2). If you modify the mapping of an `out` parameter, as discussed in the previous section, it is possible that the holder class for the parameter's unmodified type is no longer compatible with the custom type you have specified. The holder class generated for `StringSeq` is shown below:

```
public final class StringSeqHolder
{
    public
    StringSeqHolder()
    {
    }
```

```
    public
    StringSeqHolder(String[] value)
    {
        this.value = value;
    }

    public String[] value;
}
```

An `out` parameter of type `StringSeq` would normally map to a proxy method that used `StringSeqHolder` to hold the parameter value. When the parameter is modified, as is the case with the `modifiedOutParam` operation, the Slice-to-Java compiler cannot use `StringSeqHolder` to hold an instance of `java.util.List<String>`, because `StringSeqHolder` is only appropriate for the default mapping to a native array.

As a result, the compiler handles these situations using instances of the generic class `Ice.Holder<`*T*`>`, where *T* is the parameter's formal type. Consider the following example:

```
sequence<string> StringSeq;

interface I {
    void modifiedOutParam(
        out ["java:type:java.util.ArrayList<String>"]
        StringSeq seq);
};
```

The compiler generates the following mapping for the `modifiedOutParam` proxy method:

```
void modifiedOutParam(
    Ice.Holder<java.util.List<java.lang.String> > seq,
    Ice.Current curr)
```

The formal type of the parameter is `java.util.List<String>`, therefore the holder class becomes `Ice.Holder<java.util.List<String>>`.

10.16.3 JavaBean Mapping

The Java mapping optionally generates JavaBean-style methods for the data members of class, structure and exception types.

Generated Methods

For each data member *val* of type *T*, the mapping generates the following methods:

```
public T getVal();
public void setVal(T v);
```

The mapping generates an additional method if *T* is the `bool` type:

```
public boolean isVal();
```

Finally, if *T* is a sequence type with an element type *E*, two methods are generated to provide direct access to elements:

```
public E getVal(int index);
public void setVal(int index, E v);
```

Note that these element methods are only generated for sequence types that use the default mapping.

The Slice-to-Java compiler considers it a fatal error for a JavaBean method of a class data member to conflict with a declared operation of the class. In this situation, you must rename the operation or the data member, or disable the generation of JavaBean methods for the data member in question.

Metadata

The JavaBean methods are generated for a data member when the member or its enclosing type is annotated with the `java:getset` metadata. The following example demonstrates both styles of usage:

```
sequence<int> IntSeq;

class C {
    ["java:getset"] int i;
    double d;
};

["java:getset"]
struct S {
    bool b;
    string str;
};

["java:getset"]
exception E {
    IntSeq seq;
};
```

JavaBean methods are generated for all members of struct S and exception E, but for only one member of class C. Relevant portions of the generated code are shown below:

```
public class C extends Ice.ObjectImpl
{
    ...

    public int i;

    public int
    getI()
    {
        return i;
    }

    public void
    setI(int _i)
    {
        i = _i;
    }

    public double d;
}

public final class S implements java.lang.Cloneable
{
    public boolean b;

    public boolean
    getB()
    {
        return b;
    }

    public void
    setB(boolean _b)
    {
        b = _b;
    }

    public boolean
    isB()
    {
        return b;
    }
```

```
    public String str;

    public String
    getStr()
    {
        return str;
    }

    public void
    setStr(String _str)
    {
        str = _str;
    }

    ...
}

public class E extends Ice.UserException
{
    ...

    public int[] seq;

    public int[]
    getSeq()
    {
        return seq;
    }

    public void
    setSeq(int[] _seq)
    {
        seq = _seq;
    }

    public int
    getSeq(int _index)
    {
        return seq[_index];
    }

    public void
    setSeq(int _index, int _val)
    {
        seq[_index] = _val;
```

```
        }

        ...
    }
```

10.17 Asynchronous Method Invocation (AMI)

NOTE: As of version 3.4, Ice provides a new API for asynchronous method invocation. This section describes this API. You can find documentation for the previous API in Appendix K. Note that the old API is deprecated and will be removed in a future release.

Asynchronous Method Invocation (*AMI*) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

10.17.1 Basic Asynchronous API

Consider the following simple Slice definition:

```
module Demo {
    interface Employees {
        string getName(int number);
    };
};
```

Proxy Methods

Besides the synchronous proxy methods, **`slice2java`** generates the following asynchronous proxy methods:[4]

4. There are four additional overloads of `begin_getName` that we discuss in Sections 10.17.4 and 10.17.6.

```
public interface EmployeesPrx extends Ice.ObjectPrx
{
    // ...

    public Ice.AsyncResult begin_getName(int number);
    public Ice.AsyncResult begin_getName(
                           int number,
                           java.util.Map<String, String> __ctx);

    public String end_getName(Ice.AsyncResult __result);
}
```

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. (The `begin_` method is overloaded so you can pass a per-invocation context—see Section 32.12).

- The `begin_getName` method sends (or queues) an invocation of `getName`. This method does not block the calling thread.
- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

```
EmployeesPrx e = ...;
Ice.AsyncResult r = e.begin_getName(99);

// Continue to do other things here...

String name = e.end_getName(r);
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `AsyncResult`. This value contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResult` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. Similarly, the `end_` method has one out-parameter for each out-parameter of the corresponding Slice operation (plus the `AsyncResult` parameter). For example, consider the following operation:

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

```
Ice.AsyncResult begin_op(int inp1, String inp2);
Ice.AsyncResult begin_op(int inp1, String inp2,
                         java.util.Map<String, String> __ctx);
double end_op(Ice.BooleanHolder outp1, Ice.LongHolder outp2,
              Ice.AsyncResult r);
```

Exception Handling

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` method is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `CommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `java.lang.IllegalArgumentException`. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_` method, or if the `AsyncResult` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

10.17.2 The `AsyncResult` Class

The `AsyncResult` that is returned by the `begin_` method encapsulates the state of the asynchronous invocation:

```
public class AsyncResult {
    public Communicator getCommunicator();
    public Connection getConnection();
    public ObjectPrx getProxy();
    public String getOperation();

    public boolean isCompleted();
    public void waitForCompleted();
```

```
    public boolean isSent();
    public void waitForSent();

    public boolean sentSynchronously();
}
```

The methods have the following semantics:

- `Communicator getCommunicator()`

  This method returns the communicator that sent the invocation.

- `Connection getConnection()`

  This method returns the connection that was used for the invocation.

- `ObjectPrx getProxy()`

  This method returns the proxy that was used to call the `begin_` method.

- `String getOperation()`

  This method returns the name of the operation.

- `boolean isCompleted()`

  This method returns true if, at the time it is called, the result of an invocation is available, indicating that a call to the `end_` method will not block the caller. Otherwise, if the result is not yet available, the method returns false.

- `void waitForCompleted()`

  This method blocks the caller until the result of an invocation becomes available.

- `boolean isSent()`

  When you call the `begin_` method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued, `isSent` returns false.

- `void waitForSent()`

  This method blocks the calling thread until a request has been written to the client-side transport.

- `boolean sentSynchronously()`

  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSyn-`

chronously returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

10.17.3 Polling for Completion

The AsyncResult methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
};
```

The client repeatedly calls send to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;
while (!file.eof()) {
    byte[] bs;
    bs = file.read(chunkSize); // Read a chunk
    ft.send(offset, bs);       // Send the chunk
    offset += bs.length;
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing—the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
int chunkSize = ...;
int offset = 0;

LinkedList<Ice.AsyncResult> results =
    new LinkedList<Ice.AsyncResult>();
int numRequests = 5;
```

```
while (!file.eof()) {
    byte[] bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    Ice.AsyncResult r = ft.begin_send(offset, bs);
    offset += bs.length;

    // Wait until this request has been passed to the transport.
    r.waitForSent();
    results.add(r);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while (results.size() > numRequests) {
        Ice.AsyncResult r = results.getFirst();
        results.removeFirst();
        r.waitForCompleted();
    }
}

// Wait for any remaining requests to complete.
while (results.size() > 0) {
    Ice.AsyncResult r = results.getFirst();
    results.removeFirst();
    r.waitForCompleted();
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

10.17.4 Generic Completion Callbacks

The `begin_` method is overloaded to allow you to provide completion callbacks. Here are the corresponding methods for the `getName` operation:

```
Ice.AsyncResult begin_getName(int number, Ice.Callback __cb);

Ice.AsyncResult begin_getName(int number,
                              java.util.Map<String, String> __ctx,
                              Ice.Callback __cb);
```

The second version of `begin_getName` lets you override the default context. Following the in-parameters, the `begin_` method accepts a parameter of type `Ice.Callback`, which is a callback class with a `completed` method that you must provide. The Ice run time invokes the `completed` method when an asynchronous operation completes. For example:

```
public class MyCallback extends Ice.Callback
{
    public void completed(Ice.AsyncResult r)
    {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try {
            String name = e.end_getName(r);
            System.out.println("Name is: " + name);
        } catch (Ice.LocalException ex) {
            System.err.println("Exception is: " + ex);
        }
    }
}
```

Note that your callback class must derive from `Ice.Callback`. The implementation of your callback method must call the `end_` method. The proxy for the call is available via the `getProxy` method on the `AsyncResult` that is passed by the Ice run time. The return type of `getProxy` is `Ice.ObjectPrx`, so you must down-cast the proxy to its correct type.

Your callback method should catch and handle any exceptions that may be thrown by the `end_` method. If an operation can throw user exceptions, this means that you need an additional catch handler for `Ice.UserException` (or catch all possible user exceptions explicitly). If you allow an exception to escape from the callback method, the Ice run time produces a log entry by default and ignores the exception. (You can disable the log message by setting the property `Ice.Warn.AMICallback` to zero.)

To inform the Ice run time that you want to receive a callback for the completion of the asynchronous call, you pass the callback instance to the `begin_` method:

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
e.begin_getName(99, cb);
```

A more terse way to make an invocation is to use an anonymous class:

```
EmployeesPrx e = ...;

e.begin_getName(
        99,
        new Ice.AsyncCallback()
        {
            public void completed(Ice.AsyncResult r)
            {
                EmployeesPrx p = (EmployeesPrx)r.getProxy();
                try {
                    String name = p.end_getName(r);
                    System.out.println("Name is: " + name);
                } catch (Ice.LocalException ex) {
                    System.err.println("Exception: " + ex);
                }
            }
        });
```

This style is useful particularly for callbacks that do only a small amount of work because the code that starts the call and the code that processes the results are physically close together.

10.17.5 Sharing State Between the `begin_` and `end_` Method

It is common for the `end_` method to require access to some state that is established by the code that calls the `begin_` method. As an example, consider an application that asynchronously starts a number of operations and, as each operation completes, needs to update different user interface elements with the results. In this case, the `begin_` method knows which user interface element should receive the update, and the `end_` method needs access to that element.

Assuming that we have a `Widget` class that designates a particular user interface element, you could pass different widgets by storing the widget to be used as a member of your callback class:

```
public class MyCallback extends Ice.AsyncCallback
{
    public MyCallback(Widget w)
    {
        _w = w;
    }

    private Widget _w;

    public void completed(Ice.AsyncResult r)
    {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try {
            String name = e.end_getName(r);
            _w.writeString(name);
        } catch (Ice.LocalException ex) {
            System.err.println("Exception is: " + ex);
        }
    }
}
```

For this example, we assume that widgets have a `writeString` method that updates the relevant UI element.

When you call the `begin_` method, you pass the appropriate callback instance to inform the `end_` method how to update the display:

```
EmployeesPrx e = ...;
Widget widget1 = ...;
Widget widget2 = ...;

// Invoke the getName operation with different widget callbacks.
e.begin_getName(99, new MyCallback(widget1));
e.begin_getName(24, new MyCallback(widget2));
```

The callback class provides a simple and effective way for you to pass state between the point where an operation is invoked and the point where its results are processed. Moreover, if you have a number of operations that share common state, you can pass the same callback instance to multiple invocations. (If you do this, your callback methods may need to use synchronization.)

10.17.6 Type-Safe Completion Callbacks

The generic callback API we saw in Section 10.17.4 is not entirely type-safe:

- You must down-cast the return value of `getProxy` to the correct proxy type before you can call the `end_` method.
- You must call the correct `end_` method to match the operation called by the `begin_` method.
- You must remember to catch exceptions when you call the `end_` method; if you forget to do this, you will not know that the operation failed.

slice2java generates an additional type-safe API that takes care of these chores for you. To use type-safe callbacks, you must implement a callback class that provides two callback methods:

- a `response` method that is called if the operation succeeds
- an `exception` method that is called if the operation raises an exception

Your callback class must derive from the base class that is generated by **slice2java**. The name of this base class is *<module>*`.Callback_`*<interface>*`_`*<operation>*. Here is a callback class for an invocation of the `getName` operation:

```
public class MyCallback extends Demo.Callback_Employees_getName
{
    public void response(String name)
    {
        System.out.println("Name is: " + name);
    }

    public void exception(Ice.LocalException ex)
    {
        System.err.println("Exception is: " + ex);
    }
}
```

The `response` callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the `response` callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

The `exception` callback is invoked if the invocation fails because of an Ice run time exception. If the Slice operation can also raise user exceptions, your callback class must supply an additional overloading of `exception` that accepts an argument of type `Ice.UserException`.

The proxy methods are overloaded to accept this callback instance:

```
Ice.AsyncResult begin_getName(int number,
                              Callback_Employees_getName __cb);

Ice.AsyncResult begin_getName(int number,
                              java.util.Map<String, String> __ctx,
                              Callback_Employees_getName __cb);
```

You pass the callback to an invocation as you would with the generic API:

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
e.begin_getName(99, cb);
```

10.17.7 Oneway Invocations

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws an `IllegalArgumentException`.

The callback methods looks exactly as for a twoway invocation. For the generic API, the Ice run time does not call the `completed` callback method unless the invocation raised an exception during the `begin_` method ("on the way out"). For the type-safe API, the `response` method is never called.

10.17.8 Flow Control

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `AsyncResult.sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `AsyncResult.sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the

client stops invoking more operations until some of the queued operations have drained out of the local transport.

For the generic API, you can override the `sent` method:

```
public class MyCallback extends Ice.AsyncCallback
{
    public void completed(Ice.AsyncResult r)
    {
        // ...
    }

    public void sent(Ice.AsyncResult r)
    {
        // ...
    }
}
```

You inform the Ice run time that you want to be informed when a call has been passed to the local transport as usual:

```
e.begin_getName(99, new MyCallback());
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the `sent` method from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the `sent` method from a different thread once it has written the request to the local transport. In addition, you can find out from the `AsyncResult` that is returned by the `begin_` method whether the request was sent synchronously or was queued, by calling `sentSynchronously`.

For the generic API, the `sent` method has the following signature:

```
void sent(Ice.AsyncResult r);
```

For the type-safe API, the signature is:

```
void sent(boolean sentSynchronously);
```

For the generic API, you can find out whether the request was sent synchronously by calling `sentSynchronously` on the `AsyncResult`. For the type-safe API, the boolean `sentSynchronously` parameter provides the same information.

The `sent` methods allow you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

10.17.9 Batch Requests

Applications that send batched requests (see Section 32.16) can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `AsyncResult.getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

10.17.10 Concurrency

The Ice run time always invokes your callback methods from a separate thread, with one exception: it calls the `sent` callback from the thread calling the `begin_` method if the request could be sent synchronously. In the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` member or parameter.

10.17.11 Limitations

AMI invocations cannot be sent using collocated optimization. If you attempt to invoke an AMI operation using a proxy that is configured to use collocation optimization, the Ice run time raises `CollocationOptimizationException` if the servant happens to be collocated; the request is sent normally if the servant is not collocated. Section 32.21 provides more information about this optimization and describes how to disable it when necessary.

10.18 Slice Compiler

10.18.1 `slice2java` Command-Line Options

The Slice-to-Java compiler, **`slice2java`**, offers the following command-line options in addition to the standard options described in Section 4.20:

- **`--tie`**

  Generate tie classes (see Section 12.7).

- **`--impl`**

  Generate sample implementation files. This option will not overwrite an existing file.

- **`--impl-tie`**

  Generate sample implementation files using ties (see Section 12.7). This option will not overwrite an existing file.

- **`--checksum`** ***`CLASS`***

  Generate checksums for Slice definitions into the class ***`CLASS`***. The given class name may optionally contain a package specifier. The generated class contains checksums for all of the Slice files being translated by this invocation of the compiler. For example, the command below causes **`slice2java`** to generate the file `Checksums.java` containing the checksums for the Slice definitions in `File1.ice` and `File2.ice`:

  ```
  slice2java --checksum Checksums File1.ice File2.ice
  ```

- **`--stream`**

  Generate streaming helper functions for Slice types (see Section 35.2).

- **`--meta`** ***`META`***

  Define the global metadata directive ***`META`***. Using this option is equivalent to defining the global metadata ***`META`*** in each named Slice file, as well as in any file included by a named Slice file.

  Global metadata specified with **`--meta`** overrides any corresponding global metadata directive in the files being compiled.

10.18.2 Ant Task

The Ice for Java build system makes extensive use of an ant task named `Slice2JavaTask` that automates the Slice-to-Java compiler and may also be useful for Ice developers. The task and its supporting classes reside in the JAR file named `ant-ice.jar`, which normally can be found in the `lib` subdirectory of your Ice installation.

Execution Environment

The `Slice2JavaTask` must be able to locate and spawn the **`slice2java`** executable. You can specify the directory of your Ice installation by defining the `ice.home` ant property or the `ICE_HOME` environment variable, in which case the task assumes that the Slice compiler's executable is located in the `bin` subdirectory of the specified installation directory. For example, if `ICE_HOME` is set to `/opt/Ice` on Linux, the task assumes that the executable path name is `/opt/Ice/bin/slice2java`. Furthermore, the task also configures its shared library search path (if necessary for your platform) to ensure the executable can resolve its library dependencies.

If both `ice.home` and `ICE_HOME` are defined, `ice.home` takes precedence. If neither are defined, the task assumes that the executable can already be found in your `PATH` and that your shared library search path is configured correctly.

Finally, you can use a task parameter to specify the full path name of the Slice compiler. Again, the task assumes that your shared library search path is configured correctly.

Dependencies

The task minimizes recompilation by maintaining dependencies between Slice files. The task stores this information in a file named `.depend` in the output directory and updates the dependencies after each invocation. (You can specify a different name for this file using a task parameter.)

Note that the task does not maintain dependencies between a Slice file and its generated Java source files. Consequently, removing the generated Java source files does not cause the task to recompile a Slice file. In fact, the task only compiles a Slice file when any of the following conditions are true:

- no dependency file exists
- no dependency information is found for the Slice file

- the modification time of the Slice file is later than the modification time of the dependency file
- the Slice file includes another Slice file that is eligible for compilation

The simplest way to force the task to recompile all of your Slice files is to remove the dependency file.

Parameters

The task supports the parameters listed in Table 10.2:

Table 10.2. Ant task parameters

Attribute	Description	Required
`checksum`	Specifies the name of a class to contain the Slice checksums. See Section 10.19 for more information.	No
`dependencyfile`	Specifies an alternate name for the dependency file. If you specify a relative filename, it is relative to ant's current working directory. If not specified, the task uses the name `.depend` by default. If you do not define this attribute and `outputdir` is defined, the task creates the `.depend` file in the designated output directory (see `outputdir`).	No
`ice`	Instructs the Slice compiler to permit symbols that have the reserved prefix `Ice`. This parameter is used in the Ice build system and is not normally required by applications.	No
`outputdir`	Specifies the directory in which the Slice compiler generates Java source files. If not specified, the task uses ant's current working directory.	No
`stream`	Indicates whether to generate streaming support (see Section 35.2). If not specified, streaming support is not generated.	No
`tie`	Indicates whether to generate TIE classes (see Section 12.7). If not specified, TIE classes are not generated.	No
`translator`	Specifies the path name of the Slice compiler. If not specified, the task locates the Slice compiler in its execution environment as described on page 393.	No

For the flag parameters (`ice`, `stream`, and `tie`), legal positive values are `on`, `true`, or `yes`; negative values are `off`, `false`, or `no`.

Nested Elements

Several Slice compiler options must be defined as nested elements of the task:

- `define`

  Defines a preprocessor macro. The element supports the attributes `name` and (optionally) `value`, as shown below:

  ```
  <define name="FOO">
  <define name="BAR" value="5">
  ```

  These definitions are equivalent to the command-line options `-DFOO` and `-DBAR=5`, respectively.

- `fileset`

  Specifies the set of Slice files to be compiled. Refer to the ant documentation of its `FileSet` type for more information.

- `includepath`

  Specifies the include file search path for Slice files. In ant terminology, `includepath` is a *path-like structure*. Refer to the ant documentation of its `Path` type for more information.

- `meta`

  Defines a global metadata directive in each Slice file as well as in each included Slice file. The element supports `name` and `value` attributes.

Using the Task

Define the following `taskdef` element in your project's build file to enable the task:

```
<taskdef name="slice2java" classname="Slice2JavaTask"/>
```

This configuration assumes that `ant-ice.jar` is already present in ant's class path. Alternatively, you can specify the JAR explicitly as follows:

```
<taskdef name="slice2java" classpath="/opt/Ice/lib/ant-ice.jar"
    classname="Slice2JavaTask"/>
```

Once activated, you can invoke the task to translate your Slice files. The example shown below is a simplified version of the ant project for the `hello` demo:

```
<target name="generate" depends="init">
    <mkdir dir="generated"/>
    <slice2java outputdir="generated">
        <fileset dir="." includes="Hello.ice"/>
    </slice2java>
</target>

<target name="compile" depends="generate">
    <mkdir dir="classes"/>
    <javac srcdir=".:generated" destdir="classes">
        <exclude name="generated/**"/>
        ...
    </javac>
</target>

<target name="all" depends="compile"/>

<target name="clean">
    <delete dir="generated"/>
    <delete dir="classes"/>
</target>
```

This project demonstrates some practices that we encourage you to adopt in your own projects. First, it is helpful to keep the source files generated by the Slice compiler separate from your application's source files by dedicating an output directory for the exclusive use of the Slice compiler. Doing so helps to minimize confusion and makes it easier to configure a source-code management system to ignore generated files.

Next, we also recommend that you include a `clean` target in your ant project that removes this output directory. Assuming that the dependency file (`.depend`) is also stored in this directory, removing the output directory is an efficient way to clean up your project's source tree and guarantees that all of your Slice files are recompiled in the next build.

Finally, after seeing the `exclude` element in the invocation of `javac` you might infer that the generated code was not being compiled, but the presence of the output directory in the `srcdir` attribute ensures that the generated code is included in the build. The purpose of the `exclude` element is to prevent ant from including the generated files twice in its target list.

10.19 Using Slice Checksums

As described in Section 4.21, the Slice compilers can optionally generate checksums of Slice definitions. For **slice2java**, the **--checksum** option causes the compiler to generate a new Java class that adds checksums to a static map member. Assuming we supplied the option **--checksum Checksums** to **slice2java**, the generated class `Checksums.java` looks like this:

```
public class Checksums {
    public static java.util.Map checksums;
}
```

The read-only map `checksums` is initialized automatically prior to first use; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the dictionaries using the `equals` method. However, this is not feasible if it is possible that the server might return a superset of the client's checksums. A more general solution is to iterate over the local checksums as demonstrated below:

```
java.util.Map serverChecksums = ...
java.util.Iterator i = Checksums.checksums.entrySet().iterator();
while(i.hasNext()) {
    java.util.Map.Entry e = (java.util.Map.Entry)i.next();
    String id = (String)e.getKey();
    String checksum = (String)e.getValue();
    String serverChecksum = (String)serverChecksums.get(id);
    if (serverChecksum == null) {
        // No match found for type id!
    } else if (!checksum.equals(serverChecksum)) {
        // Checksum mismatch!
    }
}
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

Chapter 11
Developing a File System Client in Java

11.1 Chapter Overview

In this chapter, we present the source code for a Java client that accesses the file system we developed in Chapter 5 (see Chapter 13 for the corresponding server).

11.2 The Java Client

We now have seen enough of the client-side Java mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
```

```
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

```
import Filesystem.*;

public class Client {

    // Recursively print the contents of directory "dir" in
    // tree fashion.  For files, show the contents of each file.
    // The "depth" parameter is the current nesting level
    // (for indentation).

    static void
    listRecursive(DirectoryPrx dir, int depth)
    {
        char[] indentCh = new char[++depth];
        java.util.Arrays.fill(indentCh, '\t');
        String indent = new String(indentCh);

        NodePrx[] contents = dir.list();

        for (int i = 0; i < contents.length; ++i) {
            DirectoryPrx subdir
                = DirectoryPrxHelper.checkedCast(contents[i]);
            FilePrx file
                = FilePrxHelper.uncheckedCast(contents[i]);
            System.out.println(indent + contents[i].name() +
                (subdir != null ? " (directory):" : " (file):"));
            if (subdir != null) {
                listRecursive(subdir, depth);
            } else {
```

```
                String[] text = file.read();
                for (int j = 0; j < text.length; ++j)
                    System.out.println(indent + "\t" + text[j]);
            }
        }
    }

    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            // Create a communicator
            //
            ic = Ice.Util.initialize(args);

            // Create a proxy for the root directory
            //
            Ice.ObjectPrx base
                = ic.stringToProxy("RootDir:default -p 10000");
            if (base == null)
                throw new RuntimeException("Cannot create proxy");

            // Down-cast the proxy to a Directory proxy
            //
            DirectoryPrx rootDir
                = DirectoryPrxHelper.checkedCast(base);
            if (rootDir == null)
                throw new RuntimeException("Invalid proxy");

            // Recursively list the contents of the root directory
            //
            System.out.println("Contents of root directory:");
            listRecursive(rootDir, 0);
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
```

```
                ic.destroy();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                status = 1;
            }
        }
        System.exit(status);
    }
}
```

After importing the `Filesystem` package, the `Client` class defines two methods: `listRecursive`, which is a helper function to print the contents of the file system, and `main`, which is the main program. Let us look at `main` first:

1. The structure of the code in `main` follows what we saw in Chapter 3. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.

   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.

- If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:

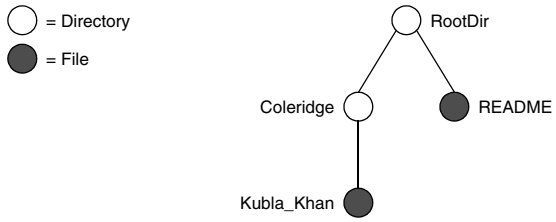


Figure 11.1. A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in Chapter 38 and Chapter 34.

11.3 Summary

This chapter presented a very simple client to access a server that implements the file system we developed in Chapter 5. As you can see, the Java code hardly differs from the code you would write for an ordinary Java program. This is one of

the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Java object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. As we will see in Chapter 13, this is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

Chapter 12
Server-Side Slice-to-Java Mapping

12.1 Chapter Overview

In this chapter, we present the server-side Slice-to-Java mapping (see Chapter 10 for the client-side mapping). Section 12.3 discusses how to initialize and finalize the server-side run time, sections 12.4 to 12.7 show how to implement interfaces and operations, and Section 12.8 discusses how to register objects with the server-side Ice run time. Finally, Section 12.9 shows how to implement operations asynchronously.

12.2 Introduction

The mapping for Slice data types to Java is identical on the client side and server side. This means that everything in Chapter 10 also applies to the server side. However, for the server side, there are a few additional things you need to know, specifically:

- how to initialize and finalize the server-side run time
- how to implement servants
- how to pass parameters and throw exceptions
- how to create servants and register them with the Ice run time.

We discuss these topics in the remainder of this chapter.

12.3 The Server-Side `main` Method

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. As for the client side, you must initialize the Ice run time by calling `Ice.Util.initialize` before you can do anything else in your server. `Ice.Util.initialize` returns a reference to an instance of an `Ice.Communicator`:

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // ...
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        }
        // ...
    }
}
```

`Ice.Util.initialize` accepts the argument vector that is passed to `main` by the operating system. The function scans the argument vector for any command-line options that are relevant to the Ice run time, but does not remove those options.[1] If anything goes wrong during initialization, `initialize` throws an exception.

Before leaving your `main` function, you *must* call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, `destroy` waits for any operation implementations that are still executing in the server to complete. In addition, `destroy` ensures that any outstanding threads

1. The semantics of Java arrays prevents `Ice.Util.initialize` from modifying the size of the argument vector. However, another overloading of `Ice.Util.initialize` is provided that allows the application to obtain a new argument vector with the Ice options removed.

are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your main function to terminate without calling destroy first; doing so has undefined behavior.

The general shape of our server-side main function is therefore as follows:

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // ...
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        }
        if (ic != null) {
            try {
                ic.destroy();
            } catch (Exception e) {
                e.printStackTrace();
                status = 1;
            }
        }
        System.exit(status);
    }
}
```

Note that the code places the call to Ice::initialize into a try block and takes care to return the correct exit status to the operating system. Also note that an attempt to destroy the communicator is made only if the initialization succeeded.

12.3.1 The Ice.Application Class

The preceding structure for the main function is so common that Ice offers a class, Ice.Application, that encapsulates all the correct initialization and finalization activities. The synopsis of the class is as follows (with some detail omitted for now):

```
package Ice;

public enum SignalPolicy { HandleSignals, NoSignalHandling }

public abstract class Application {
    public Application()

    public Application(SignalPolicy signalPolicy)

    public final int main(String appName, String[] args)

    public final int main(String appName, String[] args,
                          String configFile)

    public final int main(String appName, String[] args,
                          InitializationData initData)

    public abstract int run(String[] args);

    public static String appName()

    public static Communicator communicator()

    // ...
}
```

The intent of this class is that you specialize `Ice.Application` and implement the abstract `run` method in your derived class. Whatever code you would normally place in `main` goes into the `run` method instead. Using `Ice.Application`, our program looks as follows:

```
public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Server code here...

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server();
```

```
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

Note that `Application.main` is overloaded: you can pass an optional file name or an `InitializationData` structure (see Section 32.3 and Section 30.9).

If you pass a configuration file name to `main`, the settings in this file are overridden by settings in a file identified by the `ICE_CONFIG` environment variable (if defined). Property settings supplied on the command line take precedence over all other settings.

The `Application.main` function does the following:

1. It installs an exception handler for `java.lang.Exception`. If your code fails to handle an exception, `Application.main` prints the name of an exception and a stack trace on `System.err` before returning with a non-zero return value.
2. It initializes (by calling `Ice.Util.initialize`) and finalizes (by calling `Communicator.destroy`) a communicator. You can get access to the communicator for your server by calling the static `communicator` accessor.
3. It scans the argument vector for options that are relevant to the Ice run time and removes any such options. The argument vector that is passed to your `run` method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.
4. It provides the name of your application via the static `appName` member function. The return value from this call is the first argument in the call to `Application.main`, so you can get at this name from anywhere in your code by calling `Ice.Application.appName` (which is usually required for error messages). In the example above, the return value from `appName` would be `Server`.
5. It installs a shutdown hook that properly shuts down the communicator.
6. It installs a per-process logger (see Section 32.19.5) if the application has not already configured one. The per-process logger uses the value of the `Ice.ProgramName` property (see Section 30.8) as a prefix for its messages and sends its output to the standard error channel. An application can specify an alternate logger as described in Section 32.19.

Using `Ice.Application` ensures that your program properly finalizes the Ice run time, whether your server terminates normally or in response to an exception.

We recommend that all your programs use this class; doing so makes your life easier. In addition `Ice.Application` also provides features for signal handling and configuration that you do not have to implement yourself when you use this class.

Using `Ice.Application` on the Client Side

You can use `Ice.Application` for your clients as well: simply implement a class that derives from `Ice.Application` and place the client code into its `run` method. The advantage of this approach is the same as for the server side: `Ice.Application` ensures that the communicator is destroyed correctly even in the presence of exceptions.

Catching Signals

The simple server we developed in Chapter 3 had no way to shut down cleanly: we simply interrupted the server from the command line to force it to exit. Terminating a server in this fashion is unacceptable for many real-life server applications: typically, the server has to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is particularly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

Java does not provide direct support for signals, but it does allow an application to register a *shutdown hook* that is invoked when the JVM is shutting down. There are several events that trigger JVM shutdown, such as a call to `System.exit` or an interrupt signal from the operating system, but the shutdown hook is not provided with the reason for the shut down.

`Ice.Application` registers a shutdown hook by default, allowing you to cleanly terminate your application prior to JVM shutdown.

```
package Ice;

public abstract class Application {
    // ...

    synchronized public static void destroyOnInterrupt()
    synchronized public static void shutdownOnInterrupt()
    synchronized public static void setInterruptHook(Thread t)
    synchronized public static void defaultInterrupt()
    synchronized public static boolean interrupted()
}
```

The functions behave as follows:

- `destroyOnInterrupt`

  This function installs a shutdown hook that calls `destroy` on the communicator. This is the default behavior.

- `shutdownOnInterrupt`

  This function installs a shutdown hook that calls `shutdown` on the communicator.

- `setInterruptHook`

  This function installs a custom shutdown hook that takes responsibility for performing whatever action is necessary to terminate the application. Refer to the Java documentation for `Runtime.addShutdownHook` for more information on the semantics of shutdown hooks.

- `defaultInterrupt`

  This function removes the shutdown hook.

- `interrupted`

  This function returns true if the shutdown hook caused the communicator to shut down, false otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by the JVM. This is useful, for example, for logging purposes.

By default, `Ice.Application` behaves as if `destroyOnInterrupt` was invoked, therefore our server `main` function requires no change to ensure that the program terminates cleanly on JVM shutdown. (You can disable this default shutdown hook by passing the enumerator `NoSignalHandling` to the constructor. In that case, shutdown is not intercepted and terminates the VM.) However, we add a diagnostic to report the occurrence, so our `main` function now looks like:

```
public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Server code here...

        if (interrupted())
            System.err.println(appName() + ": terminating");

        return 0;
    }

    public static void
    main(String[] args)
```

```
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

During the course of normal execution, the JVM does not terminate until all non-daemon threads have completed. If an interrupt occurs, the JVM ignores the status of active threads and terminates as soon as it has finished invoking all of the installed shutdown hooks.

In a subclass of `Ice.Application`, the default shutdown hook (as installed by `destroyOnInterrupt`) blocks until the application's main thread completes. As a result, an interrupted application may not terminate successfully if the main thread is blocked. For example, this can occur in an interactive application when the main thread is waiting for console input. To remedy this situation, the application can install an alternate shutdown hook that does not wait for the main thread to finish:

```
public class Server extends Ice.Application {
    class ShutdownHook extends Thread {
        public void
        run()
        {
            try
            {
                communicator().destroy();
            }
            catch(Ice.LocalException ex)
            {
                ex.printStackTrace();
            }
        }
    }

    public int
    run(String[] args)
    {
        setInterruptHook(new ShutdownHook());

        // ...
    }
}
```

After replacing the default shutdown hook using `setInterruptHook`, the JVM will terminate as soon as the communicator is destroyed.

`Ice.Application` and Properties

Apart from the functionality shown in this section, `Ice.Application` also takes care of initializing the Ice run time with property values. Properties allow you to configure the run time in various ways. For example, you can use properties to control things such as the thread pool size or port number for a server. The `main` function of `Ice.Application` is overloaded; the second version allows you to specify the name of a configuration file that will be processed during initialization. We discuss Ice properties in more detail in Chapter 30.

Limitations of `Ice.Application`

`Ice.Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use `Ice.Application`. Instead, you must structure your code as we saw in Chapter 3 (taking care to always destroy the communicator).

12.4 Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing member functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

12.4.1 Skeleton Classes

On the client side, interfaces map to proxy classes (see Section 5.12). On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has a pure virtual member function for each operation on the corresponding interface. For example, consider the Slice definition for the `Node` interface we defined in Chapter 5 once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The Slice compiler generates the following definition for this interface:

```
package Filesystem;

public interface _NodeOperations
{
    String name(Ice.Current current);
}

public interface _NodeOperationsNC
{
    String name();
}

public interface Node extends Ice.Object,
                              _NodeOperations,
                              _NodeOperationsNC {}

public abstract class _NodeDisp extends Ice.ObjectImpl
                                implements Node
{
    // Mapping-internal code here...
}
```

The important points to note here are:

- As for the client side, Slice modules are mapped to Java packages with the same name, so the skeleton class definitions are part of the `Filesystem` package.
- For each Slice interface *<interface-name>*, the compiler generates Java interfaces `_`*<interface-name>*`Operations` and `_`*<interface-name>*`OperationsNC` (`_NodeOperations` and `_NodeOperationsNC` in this example). These interfaces contains a member function for each operation in the Slice interface. (You can ignore the `Ice.Current` parameter for the time being—we discuss it in detail in Section 32.6.)

- For each Slice interface *<interface-name>*, the compiler generates a Java interface *<interface-name>* (`Node` in this example). That interface extends `Ice.Object` and the two operations interfaces.
- For each Slice interface *<interface-name>*, the compiler generates an abstract class `_`*<interface-name>*`Disp` (`_NodeDisp` in this example). This abstract class is the actual skeleton class; it is the base class from which you derive your servant class.

12.4.2 Servant Classes

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

```
package Filesystem;

public final class NodeI extends _NodeDisp {

    public NodeI(String name)
    {
        _name = name;
    }

    public String name(Ice.Current current)
    {
        return _name;
    }

    private String _name;
}
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.) Note that `NodeI` extends `_NodeDisp`, that is, it derives from its skeleton class.

As far as Ice is concerned, the `NodeI` class must implement only a single method: the `name` method that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a

constructor. (Obviously, the constructor initializes the `_name` member and the `name` function returns its value.)

Normal and `idempotent` Operations

Whether an operation is an ordinary operation or an `idempotent` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

```
interface Example {
               void   normalOp();
    idempotent void   idempotentOp();
    idempotent string readonlyOp();
};
```

The operations class for this interface looks like this:

```
public interface _ExampleOperations
{
    void normalOp(Ice.Current current);
    void idempotentOp(Ice.Current current);
    String readonlyOp(Ice.Current current);
}
```

Note that the signatures of the member functions are unaffected by the `idempotent` qualifier.

12.5 Parameter Passing

For each parameter of a Slice operation, the Java mapping generates a corresponding parameter for the corresponding method in the `_<interface-name>Operations` interface. In addition, every operation has an additional, trailing parameter of type `Ice.Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` member function of the `_NodeOperations` interface has a single parameter of type `Ice.Current`. We explain the purpose of this parameter in Section 32.6 and will ignore it for now.

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

```
module M {
    interface Example {
        string op(string sin, out string sout);
    };
};
```

The generated skeleton class for this interface looks as follows:

```
public interface _ExampleOperations
{
    String op(String sin, Ice.StringHolder sout,
              Ice.Current current);
}
```

As you can see, there are no surprises here. For example, we could implement `op` as follows:

```
public final class ExampleI extends M._ExampleDisp {

    public String op(String sin, Ice.StringHolder sout,
                     Ice.Current current)
    {
        System.out.println(sin);     // In params are initialized
        sout.value = "Hello World!"; // Assign out param
        return "Done";
    }
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal Java rules and do not require special-purpose API calls.

12.6 Raising Exceptions

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

```
// ...

public void
write(String[] text, Ice.Current current)
```

```
    throws GenericError

{
    // Try to write file contents here...
    // Assume we are out of space...
    if (error) {
        GenericError e = new GenericError();
        e.reason = "file too large";
        throw e;
    }
}
```

If you throw an arbitrary Java run-time exception (such as a `ClassCastException`), the Ice run time catches the exception and then returns an `UnknownException` to the client. Similarly, if you throw an "impossible" user exception (a user exception that is not listed in the exception specification of the operation), the client receives an `UnknownUserException`.

If you throw an Ice run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`.[2] For that reason, you should never throw system exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

12.7 Tie Classes

The mapping to skeleton classes we saw in Section 12.4 requires the servant class to inherit from its skeleton class. Occasionally, this creates a problem: some class libraries require you to inherit from a base class in order to access functionality provided by the library; because Java does not support multiple inheritance, this means that you cannot use such a class library to implement your servants because your servants cannot inherit from both the library class and the skeleton class simultaneously.

To allow you to still use such class libraries, Ice provides a way to write servants that replaces inheritance with delegation. This approach is supported by *tie classes*. The idea is that, instead of inheriting from the skeleton class, you

2. There are three run-time exceptions that are not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`. We discuss these exceptions in more detail in Chapter 33.

simply create a class (known as an *implementation class* or *delegate class*) that contains methods corresponding to the operations of an interface. You use the **`--tie`** option with the **`slice2java`** compiler to create a tie class. For example, for the `Node` interface we saw in Section 12.4.1, the **`--tie`** option causes the compiler to create exactly the same code as we saw previously, but to also emit an additional tie class. For an interface *`<interface-name>`*, the generated tie class has the name `_`*`<interface-name>`*`Tie`:

```
package Filesystem;

public class _NodeTie extends _NodeDisp implements Ice.TieBase {

    public _NodeTie() {}

    public
    _NodeTie(_NodeOperations delegate)
    {
        _ice_delegate = delegate;
    }

    public java.lang.Object
    ice_delegate()
    {
        return _ice_delegate;
    }

    public void
    ice_delegate(java.lang.Object delegate)
    {
        _ice_delegate = (_NodeOperations)delegate;
    }

    public boolean
    equals(java.lang.Object rhs)
    {
        if (this == rhs)
        {
            return true;
        }
        if (!(rhs instanceof _NodeTie))
        {
            return false;
        }

        return _ice_delegate.equals(((_NodeTie)rhs)._ice_delegate)
```

```
;
    }

    public int
    hashCode()
    {
        return _ice_delegate.hashCode();
    }

    public String
    name(Ice.Current current)
    {
        return _ice_delegate.name(current);
    }

    private _NodeOperations _ice_delegate;
}
```

This looks a lot worse than it is: in essence, the generated tie class is simply a servant class (it extends `_NodeDisp`) that delegates each invocation of a method that corresponds to a Slice operation to your implementation class (see Figure 12.1).

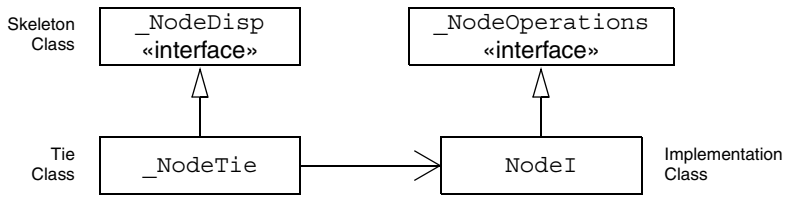


Figure 12.1. A skeleton class, tie class, and implementation class.

The generated tie class also implements the `Ice.TieBase` interface, which defines methods for obtaining and changing the delegate object:

```
package Ice;

public interface TieBase {
    java.lang.Object ice_delegate();
    void ice_delegate(java.lang.Object delegate);
}
```

The delegate has type `java.lang.Object` in these methods in order to allow a tie object's delegate to be manipulated without knowing its actual type. However,

the `ice_delegate` modifier raises `ClassCastException` if the given delegate object is not of the correct type.

Given this machinery, we can create an implementation class for our `Node` interface as follows:

```
package Filesystem;

public final class NodeI implements _NodeOperations {
{
    public NodeI(String name)
    {
        _name = name;
    }

    public String name(Ice.Current current)
    {
        return _name;
    }

    private String _name;
}
```

Note that this class is identical to our previous implementation, except that it implements the `_NodeOperations` interface and does not extend `_NodeDisp` (which means that you are now free to extend any other class to support your implementation).

To create a servant, you instantiate your implementation class and the tie class, passing a reference to the implementation instance to the tie constructor:

```
NodeI fred = new NodeI("Fred");          // Create implementation
_NodeTie servant = new _NodeTie(fred);   // Create tie
```

Alternatively, you can also default-construct the tie class and later set its delegate instance by calling `ice_delegate`:

```
_NodeTie servant = new _NodeTie();       // Create tie
// ...
NodeI fred = new NodeI("Fred");          // Create implementation
// ...
servant.ice_delegate(fred);              // Set delegate
```

When using tie classes, it is important to remember that the tie instance is the servant, not your delegate. Furthermore, you must not use a tie instance to incarnate (see Section 12.8) an Ice object until the tie has a delegate. Once you have set

the delegate, you must not change it for the lifetime of the tie; otherwise, undefined behavior results.

You should use the tie approach only if you need to, that is, if you need to extend some base class in order to implement your servants: using the tie approach is more costly in terms of memory because each Ice object is incarnated by two Java objects instead of one, the tie and the delegate. In addition, call dispatch for ties is marginally slower than for ordinary servants because the tie forwards each operation to the delegate, that is, each operation invocation requires two function calls instead of one.

Also note that, unless you arrange for it, there is no way to get from the delegate back to the tie. If you need to navigate back to the tie from the delegate, you can store a reference to the tie in a member of the delegate. (The reference can, for example, be initialized by the constructor of the delegate.)

12.8 Object Incarnation

Having created a servant class such as the rudimentary `NodeI` class in Section 12.4.2, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

12.8.1 Instantiating a Servant

Instantiating a servant means to allocate an instance:

```
Node servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance and assigns its address to a reference of type `Node`. This works because `NodeI` is derived from `Node`, so a `Node` reference can refer to an instance of type `NodeI`. However, if we want to invoke a member function of the `NodeI` class at this point, we must use a `NodeI` reference:

```
NodeI servant = new NodeI("Fred");
```

Whether you use a `Node` or a `NodeI` reference depends purely on whether you want to invoke a member function of the `NodeI` class: if not, a `Node` reference works just as well as a `NodeI` reference.

12.8.2 Creating an Identity

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.[3] An Ice object identity is a structure with the following Slice definition:

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
    // ...
};
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (See Section 32.6 for a discussion of the `category` field.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

```
Ice.Identity id = new Ice.Identity();
id.name = "Fred"; // Not unique, but good enough for now
```

12.8.3 Activating a Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

```
_adapter.add(servant, id);
```

3. The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity. See Chapter 34 for further discussion.

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state (see Section 32.4), client requests are dispatched to the servant as soon as you call `add`.

12.8.4 UUIDs as Identities

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) [14] as identities. The `Ice.Util` package contains a helper function to create such identities:

```
public class Example {
    public static void
    main(String[] args)
    {
        System.out.println(Ice.Util.generateUUID());
    }
}
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` creates a string that differs from all previous ones.[4] You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant

4. Well, almost: eventually, the UUID algorithm wraps around and produces strings that repeat themselves, but this will not happen until approximately the year 3400.

map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

```
_adapter.addWithUUID(new NodeI("Fred"));
```

12.8.5 Creating Proxies

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Chapter 3. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

```
NodePrx proxy = NodePrxHelper.uncheckedCast(
                    _adapter.addWithUUID(new NodeI("Fred")));
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice.ObjectPrx`.

Direct Proxy Creation

The object adapter offers an operation to create a proxy for a given identity:

```
module Ice {
    local interface ObjectAdapter {
        Object* createProxy(Identity id);
        // ...
    };
};
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

```
Ice.Identity id = new Ice.Identity();
id.name = Ice.Util.generateUUID();
Ice.ObjectPrx o = _adapter.createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Chapter 34.)

12.9 Asynchronous Method Dispatch (AMD)

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool (see Section 32.10). If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (*AMD*), the server-side equivalent of AMI (see Section 10.17), addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server sends a response explicitly using a callback object provided by the Ice run time.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data (i.e., the callback object and operation arguments) for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results while still in the dispatch thread, and then continues using the dispatch thread for additional work.

12.9.1 Enabling AMD with Metadata

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive *replaces* synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

```
["amd"] interface I {
  bool isValid();
  float computeRate();
};

interface J {
  ["amd"] void startProcess();
  int endProcess();
};
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface or class level) minimizes the amount of generated code and, more importantly, minimizes complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

12.9.2 AMD Mapping

The Java mapping emits the following code for each AMD operation:

1. A callback interface used by the implementation to notify the Ice run time about the completion of an operation. The name of this interface is formed using the pattern `AMD_`*`class_op`*. For example, an operation named `foo` defined in interface `I` results in an interface named `AMD_I_foo`. The interface is generated in the same scope as the interface or class containing the operation. Two methods are provided:

```
public void ice_response(<params>);
```

The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-`void` return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's `out` parameters follow the return value, in the order of declaration.

```
public void ice_exception(java.lang.Exception ex);
```

The `ice_exception` method allows the server to raise an exception. With respect to exceptions, there is less compile-time type safety in an AMD implementation because there is no `throws` clause on the dispatch method and any exception type could conceivably be passed to `ice_exception`. However, the Ice run time validates the exception value using the same semantics as for synchronous dispatch (see Section 4.10.4).

Neither `ice_response` nor `ice_exception` throw any exceptions to the caller.

2. The dispatch method, whose name has the suffix `_async`. This method has a `void` return type. The first parameter is a reference to an instance of the callback interface described above. The remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
  ["amd"] int foo(short s, out long l);
};
```

The callback interface generated for operation `foo` is shown below:

```
public interface AMD_I_foo {
    void ice_response(int __ret, long l);
    void ice_exception(java.lang.Exception ex);
}
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

```
void foo_async(AMD_I_foo __cb, short s);
```

12.9.3 Exceptions

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread

that receives the invocation), and the response thread (the thread that sends the response).[5] Although we recommend that the callback object be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the callback object. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

Whether raised in a dispatch thread or reported via the callback object, user exceptions are validated as described in Section 4.10.2, and local exceptions may undergo the translation described in Section 4.10.4.

12.9.4 Example

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

```
module Demo {
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {};

    interface Model {
        ["amd"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    };
};
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo._ModelDisp` and supplies a definition for the `interpolate_async` method that creates a `Job` to hold the callback

5. These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

object and arguments, and adds the `Job` to a queue. The method is synchronized to guard access to the queue:

```
public final class ModelI extends Demo._ModelDisp {
    synchronized public void interpolate_async(
        Demo.AMD_Model_interpolate cb,
        float[][] data,
        float factor,
        Ice.Current current)
            throws RangeError
    {
        _jobs.add(new Job(cb, data, factor));
    }

    java.util.LinkedList _jobs = new java.util.LinkedList();
}
```

After queuing the information, the operation returns control to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute`, which uses `interpolateGrid` (not shown) to perform the computational work:

```
class Job {
    Job(Demo.AMD_Model_interpolate cb,
        float[][] grid,
        float factor)
    {
        _cb = cb;
        _grid = grid;
        _factor = factor;
    }

    void execute()
    {
        if (!interpolateGrid()) {
            _cb.ice_exception(new Demo.RangeError());
            return;
        }
        _cb.ice_response(_grid);
    }

    private boolean interpolateGrid() {
        // ...
    }
```

```
    private Demo.AMD_Model_interpolate _cb;
    private float[][] _grid;
    private float _factor;
}
```

If `interpolateGrid` returns `false`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

12.10 Summary

This chapter presented the server-side Java mapping. Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanism to the client side: a small API to initialize and finalize the run time, plus a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Even though the examples in this chapter are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers (which we discuss in Chapter 32), you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any Java mapping rules beyond those we described here.

Chapter 13
Developing a File System Server in Java

13.1 Chapter Overview

In this chapter, we present the source code for a Java server that implements the file system we developed in Chapter 5 (see Chapter 11 for the corresponding client). The code we present here is fully functional, apart from the required interlocking for threads. (We examine threading issues in detail in Section 32.10.)

13.2 Implementing a File System Server

We have now seen enough of the server-side Java mapping to implement a server for the file system we developed in Chapter 5. (You may find it useful to review the Slice definition for our file system in Section 5.4 before studying the source code.)

Our server is composed of three source files:

- `Server.java`

  This file contains the server main program.

- `Filesystem/DirectoryI.java`

  This file contains the implementation for the `Directory` servants.

- `Filesystem/FileI.java`

  This file contains the implementation for the `File` servants.

13.2.1 The Server `main` Program

Our server main program, in the file `Server.java`, uses the `Ice.Application` class we discussed in Section 12.3.1. The `run` method installs a shutdown hook, creates an object adapter, instantiates a few servants for the directories and files in the file system, and then activates the adapter. This leads to a `main` program as follows:

```
import Filesystem.*;

public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Create an object adapter (stored in the _adapter
        // static members)
        //
        Ice.ObjectAdapter adapter
            = communicator().createObjectAdapterWithEndpoints(
                        "SimpleFilesystem", "default -p 10000");
        DirectoryI._adapter = adapter;
        FileI._adapter = adapter;

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI root = new DirectoryI("/", null);

        // Create a file "README" in the root directory
        //
        File file = new FileI("README", root);
        String[] text;
        text = new String[] {
            "This file system contains a collection of poetry."
        };
        try {
            file.write(text, null);
        } catch (GenericError e) {
            System.err.println(e.reason);
        }

        // Create a directory "Coleridge" in the root directory
```

```
        //
        DirectoryI coleridge
            = new DirectoryI("Coleridge", root);

        // Create a file "Kubla_Khan" in the Coleridge directory
        //
        file = new FileI("Kubla_Khan", coleridge);
        text = new String[]{ "In Xanadu did Kubla Khan",
                             "A stately pleasure-dome decree:",
                             "Where Alph, the sacred river, ran",
                             "Through caverns measureless to man",
                             "Down to a sunless sea." };
        try {
            file.write(text, null);
        } catch (GenericError e) {
            System.err.println(e.reason);
        }

        // All objects are created, allow client requests now
        //
        adapter.activate();

        // Wait until we are done
        //
        communicator().waitForShutdown();

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server();
        System.exit(app.main("Server", args));
    }
}
```

The code imports the contents of the `Filesystem` package. This avoids having to continuously use fully-qualified identifiers with a `Filesystem.` prefix.

The next part of the source code is the definition of the `Server` class, which derives from `Ice.Application` and contains the main application logic in its `run` method. Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown in Figure 13.1.

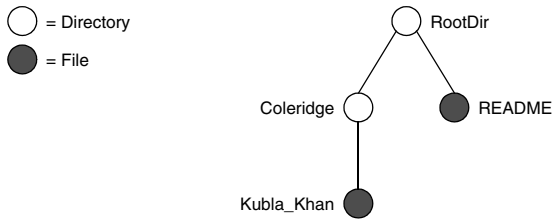


Figure 13.1. A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters, the name of the directory or file to be created and a reference to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent.) Thus, the statement

```
DirectoryI root = new DirectoryI("/", null);
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in Figure 13.1:

```
// Create the root directory (with name "/" and no parent)
//
DirectoryI root = new DirectoryI("/", null);

// Create a file "README" in the root directory
//
File file = new FileI("README", root);
String[] text;
text = new String[] {
    "This file system contains a collection of poetry."
};
try {
    file.write(text, null);
} catch (GenericError e) {
    System.err.println(e.reason);
}

// Create a directory "Coleridge" in the root directory
//
DirectoryI coleridge
```

```
        = new DirectoryI("Coleridge", root);

    // Create a file "Kubla_Khan" in the Coleridge directory
    //
    file = new FileI("Kubla_Khan", coleridge);
    text = new String[]{ "In Xanadu did Kubla Khan",
                         "A stately pleasure-dome decree:",
                         "Where Alph, the sacred river, ran",
                         "Through caverns measureless to man",
                         "Down to a sunless sea." };
    try {
        file.write(text, null);
    } catch (GenericError e) {
        System.err.println(e.reason);
    }
```

We first create the root directory and a file `README` within the root directory. (Note that we pass a reference to the root directory as the parent when we create the new node of type `FileI`.)

The next step is to fill the file with text:

```
String[] text;
text = new String[] {
    "This file system contains a collection of poetry."
};
try {
    file.write(text, null);
} catch (GenericError e) {
    System.err.println(e.reason);
}
```

Recall from Section 10.7.3 that Slice sequences by default map to Java arrays. The Slice type `Lines` is simply an array of strings; we add a line of text to our `README` file by initializing the `text` array to contain one element.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

```
    file.write(text, null);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a reference to the servant (of type `FileI`) and *not* via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place—such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary Java function call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in Figure 13.1.

13.2.2 The `FileI` Servant Class

Our `FileI` servant class has the following basic structure:

```
public class FileI extends _FileDisp
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private String[] _lines;
}
```

The class has a number of data members:

- `_adapter`

  This static member stores a reference to the single object adapter we use in our server.
- `_name`

  This member stores the name of the file incarnated by the servant.
- `_parent`

  This member stores the reference to the servant for the file's parent directory.
- `_lines`

  This member holds the contents of the file.

The `_name` and `_parent` data members are initialized by the constructor:

```
public
FileI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    assert(_parent != null);

    // Create an identity
    //
    Ice.Identity myID = new Ice.Identity();
```

```
    myID.name = Ice.Util.generateUUID();

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);

    // Create a proxy for the new node and
    // add it as a child to the parent
    //
    NodePrx thisNode
        = NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));
    _parent.addChild(thisNode);
}
```

After initializing the `_name` and `_parent` members, the code verifies that the reference to the parent is not null because every file must have a parent directory. The constructor then generates an identity for the file by calling `Ice.Util.generateUUID` and adds itself to the servant map by calling `ObjectAdapter.add`. Finally, the constructor creates a proxy for this file and calls the `addChild` method on its parent directory. `addChild` is a helper function that a child directory or file calls to add itself to the list of descendant nodes of its parent directory. We will see the implementation of this function on page 441.

The remaining methods of the `FileI` class implement the Slice operations we defined in the `Node` and `File` Slice interfaces:

```
// Slice Node::name() operation

public String
name(Ice.Current current)
{
    return _name;
}

// Slice File::read() operation

public String[]
read(Ice.Current current)
{
    return _lines;
}

// Slice File::write() operation

public void
```

```
write(String[] text, Ice.Current current)
    throws GenericError
{
    _lines = text;
}
```

The `name` method is inherited from the generated `Node` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived). It simply returns the value of the `_name` member.

The `read` and `write` methods are inherited from the generated `File` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived) and simply return and set the `_lines` member.

13.2.3 The DirectoryI Servant Class

The `DirectoryI` class has the following basic structure:

```
package Filesystem;

public final class DirectoryI extends _DirectoryDisp
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private java.util.ArrayList _contents
        = new java.util.ArrayList();
}
```

As for the `FileI` class, we have data members to store the object adapter, the name, and the parent directory. (For the root directory, the `_parent` member holds a null reference.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

```
public
DirectoryI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    // Create an identity. The parent has the
    // fixed identity "RootDir"
    //
```

```
    Ice.Identity myID = new Ice.Identity();
    myID.name = _parent != null ? Ice.Util.generateUUID() : "RootD
ir";

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);

    // Create a proxy for the new node and add it as a
    // child to the parent
    //
    NodePrx thisNode
        = NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));
    if (_parent != null)
        _parent.addChild(thisNode);
}
```

The constructor creates an identity for the new directory by calling `Ice.Util.generateUUID`. (For the root directory, we use the fixed identity `"RootDir"`.) The servant adds itself to the servant map by calling `ObjectAdapter.add` and then creates a reference to itself and passes it to the `addChild` helper function.

`addChild` simply adds the passed reference to the `_contents` list:

```
void
addChild(NodePrx child)
{
    _contents.add(child);
}
```

The remainder of the operations, `name` and `list`, are trivial:

```
public String
name(Ice.Current current)
{
    return _name;
}

// Slice Directory::list() operation

public NodePrx[]
list(Ice.Current current)
{
```

```
    NodePrx[] result = new NodePrx[_contents.size()];
    _contents.toArray(result);
    return result;
}
```

Note that the `_contents` member is of type `java.util.ArrayList`, which is convenient for the implementation of the `addChild` method. However, this requires us to convert the list into a Java array in order to return it from the `list` operation.

13.3 Summary

This chapter showed how to implement a complete server for the file system we defined in Chapter 5. Note that the server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

Note that the server code we presented here is not quite correct as it stands: if two clients access the same file in parallel, each via a different thread, one thread may read the `_lines` data member while another thread updates it. Obviously, if that happens, we may write or return garbage or, worse, crash the server. However, it is trivial to make the `read` and `write` operations thread-safe. We discuss thread safety in Section 32.10.

Part V

C# Mapping

Chapter 14
Client-Side Slice-to-C# Mapping

14.1 Chapter Overview

In this chapter, we present the client-side Slice-to-C# mapping (see Chapter 16 for the server-side mapping). One part of the client-side C# mapping concerns itself with rules for representing each Slice data type as a corresponding C# type; we cover these rules in Section 14.3 to Section 14.9. Another part of the mapping deals with how clients can invoke operations, pass and receive parameters, and handle exceptions. These topics are covered in Section 14.10 to Section 14.12. Slice classes have the characteristics of both data types and interfaces and are covered in Section 14.13. Sections 14.14 and 14.15 cover serializable objects and metadata directives, and Section 14.16 discusses asynchronous invocations. Finally, Sections 14.17 and 14.18 explain how to use the Slice compiler and Slice checksums.

14.2 Introduction

The client-side Slice-to-C# mapping defines how Slice data types are translated to C# types, and how clients invoke operations, pass parameters, and handle errors. Much of the C# mapping is intuitive. For example, by default, Slice sequences

map to C# arrays, so there is little you have learn in order to use Slice dictionaries in C#.

The C# API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data—the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least Section 14.9 to Section 14.12 in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

A word of advice before you start: in order to use the C# mapping, you should need no more than the Slice definition of your application and knowledge of the C# mapping rules. In particular, looking through the generated code in order to discern how to use the C# mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

14.3 Mapping for Identifiers

Slice identifiers map to an identical C# identifier. For example, the Slice identifier `Clock` becomes the C# identifier `Clock`. If a Slice identifier is the same as a C# keyword, the corresponding C# identifier is a *verbatim identifier* (an identifier prefixed with `@`). For example, the Slice identifier `while` is mapped as `@while`.[1]

The Slice-to-C# compiler generates classes that inherit from interfaces or base classes in the .NET framework. These interfaces and classes introduce a number of methods into derived classes. To avoid name clashes between Slice identifiers that happen to be the same as an inherited method, such identifiers are prefixed with `ice_` and suffixed with _ in the generated code. For example, the Slice iden-

1. As suggested in Section 4.5.3 on page 92, you should try to avoid such Slice identifiers as much as possible.

tifier `Clone` maps to the C# identifier `ice_Clone_` if it would clash with an inherited `Clone`. The complete list of identifiers that are so changed is:

`Clone`	`Equals`	`Finalize`
`GetBaseException`	`GetHashCode`	`GetObjectData`
`GetType`	`MemberwiseClone`	`ReferenceEquals`
`ToString`	`checkedCast`	`uncheckedCast`

Note that Slice identifiers in this list are translated to the corresponding C# identifier only where necessary. For example, structures do not derive from `ICloneable`, so if a Slice structure contains a member named `Clone`, the corresponding C# structure's member is named `Clone` as well. On the other hand, classes do derive from `ICloneable`, so, if a Slice class contains a member named `Clone`, the corresponding C# class's member is named `ice_Clone_`.

Also note that, for the purpose of prefixing, Slice identifiers are case-insensitive, that is, both `Clone` and `clone` are escaped and map to `ice_Clone_` and `ice_clone_`, respectively.

14.4 Mapping for Modules

Slice modules map to C# namespaces with the same name as the Slice module. The mapping preserves the nesting of the Slice definitions. For example:

```
module M1 {
    // Definitions for M1 here...
    module M2 {
        // Definitions for M2 here...
    };
};

// ...

module M1 {     // Reopen M1
    // More definitions for M1 here...
};
```

This definition maps to the corresponding C# definitions:

```
namespace M1
{
    namespace M2
    {
        // ...
```

```
    }
    // ...
}

// ...

namespace M1    // Reopen M1
{
    // ...
}
```

If a Slice module is reopened, the corresponding C# namespace is reopened as well.

14.5 The `Ice` Namespace

All of the APIs for the Ice run time are nested in the `Ice` namespace, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` namespace are generated from Slice definitions; other parts of the `Ice` namespace provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` namespace throughout the remainder of the book.

14.6 Mapping for Simple Built-in Types

The Slice built-in types are mapped to C# types as shown in Table 14.1.

Table 14.1. Mapping of Slice built-in types to C#.

Slice	C#
`bool`	`bool`
`byte`	`byte`
`short`	`short`
`int`	`int`

Table 14.1. Mapping of Slice built-in types to C#.

Slice	C#
`long`	`long`
`float`	`float`
`double`	`double`
`string`	`string`

14.7 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

14.7.1 Mapping for Enumerations

Enumerations map to the corresponding enumeration in C#. For example:

```
enum Fruit { Apple, Pear, Orange };
```

Not surprisingly, the generated C# definition is identical:

```
enum Fruit { Apple, Pear, Orange };
```

14.7.2 Mapping for Structures

Ice for .NET supports two different mappings for structures. By default, Slice structures map to C# structures if they (recursively) contain only value types. If a Slice structure (recursively) contains a string, proxy, class, sequence, or dictionary member, it maps to a C# class. A metadata directive (see Section 4.17) allows you to force the mapping to a C# class for Slice structures that contain only value types.

In addition, for either mapping, you can control whether Slice data members are mapped to fields or to properties (see page 453).

Structure Mapping for Structures

Consider the following structure:

```
struct Point {
    double x;
    double y;
};
```

This structure consist of only value types and so, by default, maps to a C# partial structure:

```
public partial struct Point
{
    public double x;
    public double y;

    public Point(double x, double y);

    public override int GetHashCode();
    public override bool Equals(object other);

    public static bool operator==(Point lhs, Point rhs);
    public static bool operator!=(Point lhs, Point rhs);
}
```

For each data member in the Slice definition, the C# structure contains a corresponding public data member of the same name.

The generated constructor accepts one argument for each structure member, in the order in which they are defined in the Slice definition. This allows you to construct and initialize a structure in a single statement:

```
Point p = new Point(5.1, 7.8);
```

Note that C# does not allow a value type to declare a default constructor or to assign default values to data members.

The structure overrides the `GetHashCode` and `Equals` methods to allow you to use it as the key type of a dictionary. (Note that the static two-argument version of `Equals` is inherited from `System.Object`.) Two structures are equal if (recursively) all their data members are equal. Otherwise, they are not equal. For structures that contain reference types, `Equals` performs a deep comparison; that is, reference types are compared for value equality, not reference equality.

Class Mapping for Structures

The mapping for Slice structures to C# structures provides value semantics. Usually, this is appropriate, but there are situations where you may want to change this:

- If you use structures as members of a collection, each access to an element of the collection incurs the cost of boxing or unboxing. Depending on your situation, the performance penalty may be noticeable.
- On occasion, it is useful to be able to assign null to a structure, for example, to support "not there" semantics (such as when implementing parameters that are conceptually optional).

To allow you to choose the correct performance and functionality trade-off, the Slice-to-C# compiler provides an alternative mapping of structures to classes, for example:

```
["clr:class"] struct Point {
    double x;
    double y;
};
```

The `"clr:class"` metadata directive instructs the Slice-to-C# compiler to generate a mapping to a C# partial class for this structure. The generated code is almost identical, except that the keyword `struct` is replaced by the keyword `class`[2] and that the class has a default constructor and inherits from `ICloneable`:

```
public partial class Point : _System.ICloneable
{
    public double x;
    public double y;

    public Point();
    public Point(double x, double y);

    public object Clone();

    public override int GetHashCode();
    public override bool Equals(object other);
```

2. Some of the generated marshaling code differs for the class mapping of structures, but this is irrelevant to application code.

```
    public static bool operator==(Point lhs, Point rhs);
    public static bool operator!=(Point lhs, Point rhs);
}
```

The class has a default constructor that default-constructs each data member. This means members of primitive type are initialized to the equivalent of zero, and members of reference type are initialized to null. Note that applications must always explicitly initialize a member whose type is a class-mapped structure because the Ice run time does not accept null as a legal value for these types.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition (see Section 4.9.2). The default constructor initializes each of these data members to its declared value.

The class also provides a second constructor that has one parameter for each data member. This allows you to construct and initialize a class instance in a single statement:

```
Point p = new Point(5.1, 7.8);
```

The `Clone` method performs a shallow memberwise copy, and the comparison methods have the usual semantics (they perform value comparison).

Note that you can influence the mapping for structures only at the point of definition of a structure, that is, for a particular structure type, you must decide whether you want to use the structure or the class mapping. (You cannot override the structure mapping elsewhere, for example, for individual structure members or operation parameters.)

As we mentioned previously, if a Slice structure (recursively) contains a member of reference type, it is automatically mapped to a C# class. (The compiler behaves as if you had explicitly specified the "`clr:class`" metadata directive for the structure.)

Here is our `Employee` structure from Section 4.9.4 once more:

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The structure contains two strings, which are reference types, so the Slice-to-C# compiler generates a C# class for this structure:

```
public partial class Employee : _System.ICloneable
{
    public long number;
    public string firstName;
    public string lastName;

    public Employee();
    public Employee(long number,
                    string firstName,
                    string lastName);

    public object Clone();

    public override int GetHashCode();
    public override bool Equals(object other);

    public static bool operator==(Employee lhs, Employee rhs);
    public static bool operator!=(Employee lhs, Employee rhs);
}
```

Property Mapping for Structures

You can instruct the compiler to emit property definitions instead of public data members. For example:

```
["clr:property"] struct Point {
    double x;
    double y;
};
```

The "`clr:property`" metadata directive causes the compiler to generate a property for each Slice data member:

```
public partial struct Point
{
    private double x_prop;
    public double x {
        get {
            return x_prop;
        }
        set {
            x_prop = value;
        }
    }

    private double y_prop;
    public double y {
```

```
        get {
            return y_prop;
        }
        set {
            y_prop = value;
        }
    }

    // Other methods here...
}
```

Note that the properties are non-virtual because C# structures cannot have virtual properties. However, if you apply the "`clr:property`" directive to a structure that contains a member of reference type, or if you combine the "`clr:property`" and "`clr:class`" directives, the generated properties are virtual. For example:

```
["clr:property", "clr:class"]
struct Point {
    double x;
    double y;
};
```

This generates the following code:

```
public partial class Point : System.ICloneable
{
    private double x_prop;
    public virtual double x {
        get {
            return x_prop;
        }
        set {
            x_prop = value;
        }
    }

    private double y_prop;
    public virtual double y {
        get {
            return y_prop;
        }
        set {
            y_prop = value;
        }
```

```
    }

    // Other methods here...
}
```

14.7.3 Mapping for Sequences

Ice for .NET supports several different mappings for sequences. By default, sequences are mapped to arrays. You can use metadata directives (see Section 4.17) to map sequences to a number of alternative types:

- `System.Collections.Generic.List`
- `System.Collections.Generic.LinkedList`
- `System.Collections.Generic.Queue`
- `System.Collections.Generic.Stack`
- Types derived from `Ice.CollectionBase` (which is a drop-in replacement for `System.Collections.CollectionBase`)[3]
- User-defined custom types that derive from `System.Collections.Generic.IEnumerable<T>`.

The different mappings allow you to map sequences to a container type that provides the correct performance trade-off for your application.

Array Mapping for Sequences

By default, the Slice-to-C# compiler maps sequences to arrays. Interestingly, no code is generated in this case; you simply define an array of elements to model the Slice sequence. For example:

```
sequence<Fruit> FruitPlatter;
```

Given this definition, to create a sequence containing an apple and an orange, you could write:

```
Fruit[] fp = { Fruit.Apple, Fruit.Orange };
```

Or, alternatively:

3. This mapping is provided mainly for compatibility with Ice versions prior to 3.3.

```
Fruit fp[] = new Fruit[2];
fp[0] = Fruit.Apple;
fp[1] = Fruit.Orange;
```

The array mapping for sequences is both simple and efficient, especially for sequences that do not need to provide insertion or deletion other than at the end of the sequence.

Mapping to Predefined Generic Containers for Sequences

With metadata directives, you can change the default mapping for sequences to use generic containers provided by .NET. For example:

```
["clr:generic:List"] sequence<string> StringSeq;
["clr:generic:LinkedList"] sequence<Fruit> FruitSeq;
["clr:generic:Queue"] sequence<int> IntQueue;
["clr:generic:Stack"] sequence<double> DoubleStack;
```

The "`clr:generic:<type>`" metadata directive causes the **`slice2cs`** compiler to the map the corresponding sequence to one of the containers in the `System.Collections.Generic` namespace. For example, the `Queue` sequence maps to `System.Collections.Generic.Queue<int>` due to its metadata directive.

The predefined containers allow you to select an appropriate space–performance trade-off, depending on how your application uses a sequence. In addition, if a sequence contains value types, such as `int`, the generic containers do not incur the cost of boxing and unboxing and so are quite efficient. (For example, `System.Collections.Generic.List<int>` performs within a few percentage points of an integer array for insertion and deletion at the end of the sequence, but has the advantage of providing a richer set of operations.)

Generic containers can be used for sequences of any element type except objects. For sequences of objects, only `List` is supported because it provides the functionality required for efficient unmarshaling. Metadata that specifies any other generic type is ignored with a warning:

```
class MyClass {
    // ...
};

["clr:generic:List"]
sequence<MyClass> MyClassList; // OK

["clr:generic:LinkedList"]
sequence<MyClass> MyClassLinkedList; // Ignored
```

In this example, sequence type `MyClassList` maps to the generic container `System.Collections.Generic.List<MyClass>`, but sequence type `MyClassLinkedList` uses the default array mapping.

Mapping to Custom Types for Sequences

If the array mapping and the predefined containers are unsuitable for your application (for example, because may need a priority queue, which does not come with .NET), you can implement your own custom containers and direct **`slice2cs`** to map sequences to these custom containers. For example:

```
["clr:generic:MyTypes.PriorityQueue"] sequence<int> Queue;
```

This metadata directive causes the Slice `Queue` sequence to be mapped to the type `MyTypes.PriorityQueue`. You must specify the fully-qualified name of your custom type following the `clr:generic:` prefix. This is because the generated code prepends a `global::` qualifier to the type name you provide; for the preceding example, the generated code refers to your custom type as `global::MyTypes.PriorityQueue<int>`.

Your custom type can have whatever interface you deem appropriate, but it must meet the following requirements:

- The custom type must derive from `System.Collections.Generic.IEnumerable<T>`.
- The custom type must provide a readable `Count` property that returns the number of elements in the collection.
- The custom type must provide an `Add` method that appends an element to the end of the collection.
- If (and only if) the Slice sequence contains elements that are Slice classes, the custom type must provide an indexer that sets the value of an element at a specific index. (Indexes, as usual, start at zero.)

As an example, here is a minimal class (omitting implementation) that meets these criteria:

```
public class PriorityQueue<T> : IEnumerable<T>
{
    public IEnumerator<T> GetEnumerator();

    public int Count
        get;

    public void Add(T elmt);
```

```
    public T this[int index] // Needed for class elements only.
        set;


    // Other methods and data members here...
}
```

`CollectionBase` Mapping for Sequences

The `CollectionBase` mapping is provided mainly for compatibility with Ice versions prior to 3.3. Internally, `CollectionBase` is implemented using `System.Collections.Generic.List<T>`, so it offers the same performance trade-offs as `List<T>`. (For value types, `Ice.CollectionBase` is considerably faster than `System.Collections.CollectionBase`, however.)

`Ice.CollectionBase` is not as type-safe as `List<T>` because, in order to remain source code compatible with `System.Collections.CollectionBase`, it provides methods that accept elements of type `object`. This means that, if you pass an element of the wrong type, the problem will be diagnosed only at run time, instead of at compile time. For this reason, we suggest that you do not use the `CollectionBase` mapping for new code.

To enable the `CollectionBase` mapping, you must use the `"clr:collection"` metadata directive:

```
["clr:collection"] sequence<Fruit> FruitPlatter;
```

With this directive, **`slice2cs`** generates a type that derives from `Ice.CollectionBase`:

```
public class FruitPlatter : Ice.CollectionBase<M.Fruit>,
                            System.ICloneable
{
    public FruitPlatter();
    public FruitPlatter(int capacity);
    public FruitPlatter(Fruit[] a);
    public FruitPlatter(
        System.Collections.Generic.IEnumerable<Fruit> l);

    public static implicit operator
        _System.Collections.Generic.List<Fruit>(FruitPlatter s);

    public virtual FruitPlatter GetRange(int index, int count);
```

```
    public static FruitPlatter Repeat(Fruit value, int count);

    public object Clone();
}
```

The generated `FruitPlatter` class provides the following methods:

- `FruitPlatter();`
  `FruitPlatter(int capacity);`
  `FruitPlatter(Fruit[] a);`
  `FruitPlatter(IEnumerable<Fruit> l);`

  Apart from calling the default constructor, you can also specify an initial capacity for the sequence or, using the array constructor, initialize a sequence from an array. In addition, you can initialize the class to contain the same elements as any enumerable collection with the same element type.

- `FruitPlatter GetRange(int index, int count);`

  This method returns a new sequence with `count` elements that are copied from the source sequence beginning at `index`.

- `FruitPlatter Repeat(Fruit value, int count);`

  This method returns a sequence with `count` elements that are initialized to `value`.

- `object Clone()`

  The `Clone` method returns a shallow copy of the source sequence.

- `static implicit operator List<Fruit>`
  `(FruitPlatter s);`

  This operator performs an implicit conversion of a `FruitPlatter` instance to a `List<Fruit>`, so you can pass a `FruitPlatter` sequence where a `List<Fruit>`, `IEnumerable<Fruit>`, or `System.Collections.IEnumerable` is expected.

The remaining methods are provided by the generic `Ice.CollectionBase` base class. This class provides the following methods:

- `CollectionBase();`
  `CollectionBase(int capacity);`
  `CollectionBase(T[] a);`
  `CollectionBase(IEnumerable<T> l);`

  The constructors initialize the sequence as for the concrete derived class.

- `int Count { get; }`

This property returns the number of elements of the sequence.

- `int Capacity { get; set; }`

  This property controls the capacity of the sequence. Its semantics are as for the corresponding property of `List<T>`.

- `virtual void TrimToSize();`

  This method sets the capacity of the sequence to the actual number of elements.

- ```
  int Add(object o);
  int Add(T value);
  ```

  These methods append `value` at the end of the sequence. They return the index at which the element is inserted (which always is the value of `Count` prior the call to `Add`.)

- ```
  void Insert(int index, object o);
  void insert(int index, T value);
  ```

  These methods insert an element at the specified index.

- ```
  virtual void InsertRange(int index,
                           CollectionBase<T> c);
  virtual void InsertRange(int index, T[] c);
  ```

  These methods insert a range of values into the sequence starting at the given index.

- ```
  virtual void SetRange(int index,
                        CollectionBase<T> c);
  virtual void SetRange(int index, T[] c);
  ```

  These methods copy the provided sequence over a range of elements in the target sequence, starting at the provided index, with semantics as for `System.Collections.ArrayList`

- `void RemoveAt(int index);`

  This method deletes the element at the specified index.

- ```
  void Remove(object o);
  void Remove(T value);
  ```

  These methods search for the specified element and, if present, delete that element. If the element is not in the sequence, the methods do nothing.

- `virtual void RemoveRange(int index, int count);`

  This method removes `count` elements, starting at the given index.
  ```

- `void Clear();`

  This method deletes all elements of the sequence.

- `bool Contains(object o);`
  `bool Contains(T value);`

  These methods return true if the sequence contains `value`; otherwise, they return false.

- `int IndexOf(object o);`
  `int IndexOf(T value);`

  These methods return the index of the specified element. If the element is not in the sequence, the return value is `-1`.

- `virtual int LastIndexOf(T value);`
  `virtual int LastIndexOf(T value, int startIndex);`
  `virtual int LastIndexOf(T value, int startIndex, int count);`

  These methods search for the provided element and return its last occurrence in the sequence, as for `System.Collections.ArrayList.LastIndexOf`.

- `object this[int index] { get; set; }`
  `T this[int index] { get; set; }`

  The indexers allow you to read and write elements using array subscript notation.

- `IEnumerator<T> GetEnumerator();`

  This method returns an enumerator that you can use to iterate over the collection.

- `static implicit operator List<T> (CollectionBase<T> s);`

  As for the derived class, this operator permits implicit conversion to a `List<T>`.

- `void CopyTo(T[] a);`
  `void CopyTo(T[] a, int i);`
  `void CopyTo(int i, T[] a, int ai, int c);`
  `void CopyTo(System.Array a, int i);`

  These methods copy the contents of a sequence into an array. The semantics are the same as for the corresponding methods of `List<T>`.

- `T[] ToArray();`

  The `ToArray` method returns the contents of the sequence as an array.

- `void AddRange(CollectionBase<T> s);`
  `void AddRange(T[] a);`

  The `AddRange` methods append the contents of a sequence or an array to the current sequence, respectively.

- `virtual void Sort();`
  `virtual void Sort(System.Collections.IComparer comparer);`
  `virtual void Sort(int index, int count, System.Collections.IComparer comparer);`

  These methods sort the sequence.

- `virtual void Reverse();`
  `virtual void Reverse(int index, int count);`

  These methods reverse the order of elements of the sequence.

- `virtual int BinarySearch(T value);`
  `virtual int BinarySearch(T value, System.Collections.IComparer comparer);`
  `virtual int BinarySearch(int index, int count, T value, System.Collections.IComparer comparer);`

  The methods perform a binary search on the sequence, with semantics as for `System.Collections.ArrayList`.

- `static FruitPlatter Repeat(Fruit value, int count);`

  This method returns a sequence with `count` elements that are initialized to `value`.

Note that for all methods that return sequences, these methods perform a shallow copy, that is, if you have a sequence whose elements have reference type, what is copied are the references, not the objects denoted by those references.

`Ice.CollectionBase` also provides the usual `GetHashCode` and `Equals` methods, as well as the comparison operators for equality and inequality. (Two sequences are equal if they have the same number of elements and all elements in corresponding positions are equal, as determined by the `Equals` method of the elements.)

`Ice.CollectionBase` also implements the inherited `IsFixedSize`, `IsReadOnly`, and `IsSynchronized` properties (which return false), and the inherited `SyncRoot` property (which returns `this`).

Creating a sequence containing an apple and an orange is simply a matter of writing:

```
FruitPlatter fp = new FruitPlatter();
fp.Add(Fruit.Apple);
fp.Add(Fruit.Orange);
```

Multi-Dimensional Sequences

Slice permits you to define sequences of sequences, for example:

```
enum Fruit { Apple, Orange, Pear };
["clr:generic:List"] sequence<Fruit> FruitPlatter;
["clr:generic:LinkedList"] sequence<FruitPlatter> Cornucopia;
```

If we use these definitions as shown, the type of FruitPlatter in the generated code is:

```
System.Collections.Generic.LinkedList<
    System.Collections.Generic.List<Fruit>
>
```

As you can see, the outer sequence contains elements of type `List<Fruit>`, as you would expect.

Now let us modify the definition to change the mapping of `FruitPlatter` to an array:

```
enum Fruit { Apple, Orange, Pear };
sequence<Fruit> FruitPlatter;
["clr:LinkedList"] sequence<FruitPlatter> Cornucopia;
```

With this definition, the type of `Cornucopia` becomes:

```
System.Collections.Generic.LinkedList<Fruit[]>
```

As you can see, the generated code now no longer mentions the type `FruitPlatter` anywhere and deals with the outer sequence elements as an array of `Fruit` instead.

14.7.4 Mapping for Dictionaries

Ice for .NET supports three different mappings for dictionaries. By default, dictionaries are mapped to `System.Collec-`

tions.Generic.Dictionary<T>. You can use metadata directives (see Section 4.17) to map dictionaries to two other types:

- System.Collections.Generic.SortedDictionary
- Types derived from Ice.DictionaryBase (which is a drop-in replacement for System.Collections.DictionaryBase)[4]

Mapping to Predefined Containers for Dictionaries

Here is the definition of our EmployeeMap from Section 4.9.4 once more:

```
dictionary<long, Employee> EmployeeMap;
```

By default, he Slice-to-C# compiler maps the dictionary to the following type:

```
System.Collections.Generic.Dictionary<Employee>
```

You can use the "clr:generic:SortedDictionary" metadata directive to change the mapping to a sorted dictionary:

```
["clr:generic:SortedDictionary"]
dictionary<long, Employee> EmployeeMap;
```

With this definition, the type of the dictionary becomes:

```
System.Collections.Generic.SortedDictionary<Employee>
```

DictionaryBase mapping for Dictionaries

The DictionaryBase mapping is provided mainly for compatibility with Ice versions prior to 3.3. Internally, DictionaryBase is implemented using System.Collections.Generic.Dictionary<T>, so it offers the same performance trade-offs as Dictionary<T>. (For value types, Ice.DictionaryBase is considerably faster than System.Collections.DictionaryBase, however.)

Ice.DictionaryBase is not as type-safe as Dictionary<T> because, in order to remain source code compatible with System.Collections.DictionaryBase, it provides methods that accept elements of type object. This means that, if you pass an element of the wrong type, the problem will be diagnosed only at run time, instead of at compile time. For this reason, we suggest that you do not use the DictionaryBase mapping for new code.

4. This mapping is provided mainly for compatibility with Ice versions prior to 3.3.

To enable the `DictionaryBase` mapping, you must use the `"clr:collection"` metadata directive:

```
["clr:collection"] dictionary<long, Employee> EmployeeMap;
```

With this directive, **`slice2cs`** generates a type that derives from `Ice.CollectionBase`:

```
public class EmployeeMap : Ice.DictionaryBase<long, Employee>,
                           System.ICloneable
{
    public void AddRange(EmployeeMap m);
    public object Clone();
}
```

Note that the generated `EmployeeMap` class derives from `Ice.DictionaryBase`, which provides a super-set of the interface of the .NET `System.Collections.DictionaryBase` class. Apart from methods inherited from `DictionaryBase`, the class provides a `Clone` method and an `AddRange` method that allows you to append the contents of one dictionary to another. If the target dictionary contains a key that is also in the source dictionary, the target dictionary's value is preserved. For example:

```
Employee e1 = new Employee();
e1.number = 42;
e1.firstName = "Herb";
e1.lastName = "Sutter";

EmployeeMap em1 = new EmployeeMap();
em[42] = e;

Employee e2 = new Employee();
e2.number = 42;
e2.firstName = "Stan";
e2.lastName = "Lipmann";

EmployeeMap em2 = new EmployeeMap();
em[42] = e2;

// Add contents of em2 to em1
//
em1.AddRange(em2);

// Equal keys preserve the original value
//
Debug.Assert(em1[42].firstName.Equals("Herb"));
```

The `DictionaryBase` class provides the following methods:

```
public abstract class DictionaryBase<KT, VT>
    : System.Collections.IDictionary
{
    public DictionaryBase();

    public int Count { get; }

    public void Add(KT key, VT value);
    public void Add(object key, object value);

    public void CopyTo(System.Array a, int index);

    public void Remove(KT key);
    public void Remove(object key);

    public void Clear();

    public System.Collections.ICollection Keys { get; }
    public System.Collections.ICollection Values { get; }

    public VT this[KT key] { get; set; }
    public object this[object key] { get; set; }

    public bool Contains(KT key);
    public bool Contains(object key);

    public override int GetHashCode();
    public override bool Equals(object other);
    public static bool operator==(DictionaryBase<KT, VT> lhs,
                                  DictionaryBase<KT, VT> rhs);
    public static bool operator!=(DictionaryBase<KT, VT> lhs,
                                  DictionaryBase<KT, VT> rhs);

    public System.Collections.IEnumerator GetEnumerator();

    public bool IsFixedSize { get; }
    public bool IsReadOnly { get; }
    public bool IsSynchronized { get; }
    public object SyncRoot { get; }
}
```

The methods have the same semantics as the corresponding methods in the .NET Framework. The `Equals` method returns true if two dictionaries contain the

same number of entries and, for each entry, the key and value are the same (as determined by their `Equals` methods).

The `Clone` method performs a shallow copy.

The class also implements the inherited `IsFixedSize`, `IsReadOnly`, and `IsSynchronized` properties (which return false), and the `SyncRoot` property (which returns `this`).

14.7.5 Collection Comparison

A utility class, `Ice.CollectionComparer` allows you to compare collections for equality:

```
public class CollectionComparer {
    public static bool
    Equals(System.Collections.IDictionary d1,
           System.Collections.IDictionary d2);

    public static bool
    Equals(System.Collections.ICollection c1,
           System.Collections.ICollection c2);

    public static bool
    Equals(System.Collections.IEnumerable c1,
           System.Collections.IEnumerable c2);
}
```

Equality of the elements in a collection is determined by calling the elements' `Equals` method.

Two dictionaries are equal if they contain the same number of entries with identical keys and values.

Two collections that derive from `ICollection` or `IEnumerable` are equal if they contain the same number of entries and entries compare equal. Note that order is significant, so corresponding entries must not only be equal but must also appear in the same position.

14.8 Mapping for Constants

Here are the constant definitions we saw in Section 4.9.5 on page 103 once more:

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

```
public abstract class AppendByDefault
{
    public const bool value = true;
}

public abstract class LowerNibble
{
    public const byte value = 15;
}

public abstract class Advice
{
    public const string value = "Don't Panic!";
}

public abstract class TheAnswer
{
    public const short value = 42;
}

public abstract class PI
{
    public const double value = 3.1416;
}

public enum Fruit { Apple, Pear, Orange }

public abstract class FavoriteFruit
{
    public const Fruit value = Fruit.Pear;
}
```

As you can see, each Slice constant is mapped to a class with the same name as the constant. The class contains a member named `value` that holds the value of the constant.[5]

14.9 Mapping for Exceptions

The mapping for exceptions is based on the inheritance hierarchy shown in Figure 14.1

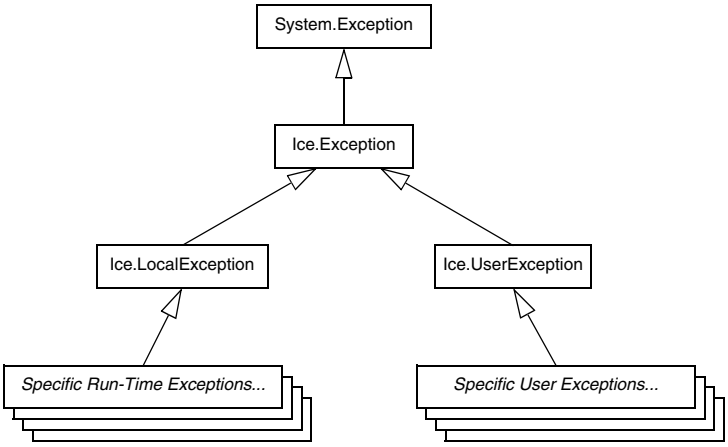


Figure 14.1. Inheritance structure for exceptions.

The ancestor of all exceptions is `System.Exception`. Derived from that is `Ice.Exception`, which provides the definitions of a number of constructors. `Ice.LocalException` and `Ice.UserException` are derived from `Ice.Exception` and form the base of all run-time and user exceptions, respectively.

The constructors defined in `Ice.Exception` have the following signatures:

```
public abstract class Exception : System.Exception
{
    public Exception();
    public Exception(System.Exception ex);
}
```

5. The mapping to classes instead of to plain constants is necessary because C# does not permit constant definitions at namespace scope.

Each concrete derived exception class implements these constructors. The second constructor initializes the `InnerException` property of `System.Exception`. (Both constructors set the `Message` property to the empty string.)

Here is a fragment of the Slice definition for our world time server from Section 4.10.5 on page 120 once more:

```
exception GenericError {
    string reason;
};

exception BadTimeVal extends GenericError {};

exception BadZoneName extends GenericError {};
```

These exception definitions map as follows:

```
public partial class GenericError : Ice.UserException
{
    public string reason;

    public GenericError();
    public GenericError(System.Exception ex__);
    public GenericError(string reason);
    public GenericError(string reason, System.Exception ex__);

    // GetHashCode and comparison methods defined here,
    // as well as mapping-internal methods.
}

public partial class BadTimeVal : M.GenericError
{
    public BadTimeVal();
    public BadTimeVal(System.Exception ex__);
    public BadTimeVal(string reason);
    public BadTimeVal(string reason, System.Exception ex__);

    // GetHashCode and comparison methods defined here,
    // as well as mapping-internal methods.
}

public partial class BadZoneName : M.GenericError
{
    public BadZoneName();
    public BadZoneName(System.Exception ex__);
    public BadZoneName(string reason);
    public BadZoneName(string reason, System.Exception ex__);
```

```
    // GetHashCode and comparison methods defined here,
    // as well as mapping-internal methods.
}
```

Each Slice exception is mapped to a C# partial class with the same name. For each exception member, the corresponding class contains a public data member. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.)

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

All user exceptions are derived from the base class `Ice.UserException`. This allows you to catch all user exceptions generically by installing a handler for `Ice.UserException`. Similarly, you can catch all Ice run-time exceptions with a handler for `Ice.LocalException`, and you can catch all Ice exceptions with a handler for `Ice.Exception`.

If an exception (or one of its base exceptions) contains data members, the mapping generates two additional constructors. These constructors allow you to instantiate and initialize an exception in a single statement, instead of having to first instantiate the exception and then assign to its members. For derived exceptions, the constructors accept one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order. The second of these constructors has a trailing parameter of type `System.Exception` which initializes the `InnerException` property of the `System.Exception` base exception.

All exceptions also provide the usual `GetHashCode` and `Equals` methods, as well as the `==` and `!=` comparison operators.

The generated exception classes also contain other member functions that are not shown here; these member functions are internal to the C# mapping and are not meant to be called by application code.

Constructors

Exceptions have a default constructor that default-constructs each data member. This means members of primitive type are initialized to the equivalent of zero, and members of reference type are initialized to null. Note that applications must always explicitly initialize a member whose type is a class-mapped structure because the Ice run time does not accept null as a legal value for these types.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice defini-

tion (see Section 4.10.2). The default constructor initializes each of these data members to its declared value.

Exceptions also provide constructors that accept one parameter for each data member. This allows you to construct and initialize a class instance in a single statement (instead of first having to construct the instance and then assigning to its members). For derived exceptions, these constructors accept one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

14.10 Mapping for Interfaces

On the client side, Slice interfaces map to C# interfaces with member functions that correspond to the operations on those interfaces. Consider the following simple interface:

```
interface Simple {
    void op();
};
```

The Slice compiler generates the following definition for use by the client:

```
public interface SimplePrx : Ice.ObjectPrx
{
    void op();
    void op(System.Collections.Generic.Dictionary<string, string>
                __context);
}
```

As you can see, the compiler generates a *proxy interface* `SimplePrx`. In general, the generated name is `<interface-name>Prx`. If an interface is nested in a module `M`, the generated interface is part of namespace `M`, so the fully-qualified name is `M.<interface-name>Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice.ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a member function of the same name. For the preceding example, we find that the operation `op` has been

mapped to the method `op`. Also note that `op` is overloaded: the second version of `op` has a parameter `__context`, which is a dictionary of string pairs. This parameter is for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the `__context` parameter in detail in Chapter 32. The parameter is also used by IceStorm—see Chapter 44.)

Because all the *`<interface-name>`*`Prx` types are interfaces, you cannot instantiate an object of such a type. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.The proxy references handed out by the Ice run time are always of type *`<interface-name>`*`Prx`; the concrete implementation of the interface is part of the Ice run time and does not concern application code.

A value of `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

14.10.1 The `Ice.ObjectPrx` Interface

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice.ObjectPrx`. `ObjectPrx` provides a number of methods:

```
namespace Ice
{
    public interface ObjectPrx
    {
        Identity ice_getIdentity();
        bool ice_isA(string id);
        string ice_id();
        void ice_ping();

        int GetHashCode();
        bool Equals(object r);

        // Defined in a helper class:
        //
        public static bool Equals(Ice.ObjectPrx lhs,
                                  Ice.ObjectPrx rhs);
        public static bool operator==(ObjectPrx lhs,
                                      ObjectPrx rhs);
        public static bool operator!=(ObjectPrx lhs,
                                      ObjectPrx rhs);
```

```
        // ...
    }
}
```

Note that the static methods are not actually defined `Ice.ObjectPrx`, but in a helper class that becomes a base class of an instantiated proxy. However, this is simply an internal detail of the C# mapping—conceptually, these methods belong with `Ice.ObjectPrx`, so we discuss them here.

The methods behave as follows:

- `ice_getIdentity`

  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  ```
  module Ice {
      struct Identity {
          string name;
          string category;
      };
  };
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

  ```
  Ice.ObjectPrx o1 = ...;
  Ice.ObjectPrx o2 = ...;
  Ice.Identity i1 = o1.ice_getIdentity();
  Ice.Identity i2 = o2.ice_getIdentity();

  if (i1.Equals(i2))
      // o1 and o2 denote the same object
  else
      // o1 and o2 denote different objects
  ```

- `ice_isA`

  This method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID (see Section 4.13). For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

  ```
  Ice.ObjectPrx o = ...;
  if (o != null && o.ice_isA("::Printer"))
      // o denotes a Printer object
  else
  ```

```
// o denotes some other type of object
```

Note that we are testing whether the proxy is null before attempting to invoke the `ice_isA` method. This avoids getting a `NullReferenceException` if the proxy is null.

- `ice_id`

  This method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type `BasePrx`, with a static type ID of `::Base`, the return value of `ice_id` might be `::Base`, or it might something more derived, such as `::Derived`.

- `ice_ping`

  This method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as `ObjectNotExist-Exception` or `ConnectTimeoutException`.

- `Equals`

  This operation compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

Note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets; we discuss these methods in Chapter 32 and Chapter 33.

14.10.2 Proxy Helpers

For each Slice interface, apart from the proxy interface, the Slice-to-C# compiler creates a helper class: for an interface `Simple`, the name of the generated helper class is `SimplePrxHelper`.[6] The helper class contains two methods of interest:

6. You can ignore the `ObjectPrxHelperBase` base class—it exists for mapping-internal purposes.

```
public class SimplePrxHelper : Ice.ObjectPrxHelperBase, SimplePrx
{
    public static SimplePrx checkedCast(Ice.ObjectPrx b);
    public static SimplePrx checkedCast(Ice.ObjectPrx b,
                        System.Collections.Generic.Dictionary
                            <string, string> ctx);
    public static SimplePrx uncheckedCast(Ice.ObjectPrx b)

    // ...
}
```

Both the `checkedCast` and `uncheckedCast` methods implement a down-cast: if the passed proxy is a proxy for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a non-null reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the cast returns a null reference.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

```
Ice.ObjectPrx obj = ...;        // Get a proxy from somewhere...

SimplePrx simple = SimplePrxHelper.checkedCast(obj);
if (simple != null)
    // Object supports the Simple interface...
else
    // Object is not of type Simple...
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`. (This also explains the need for the helper class: the Ice run time must contact the server, so we cannot use a C# down-cast.)

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end

up sending the invocation to an object of the wrong type; that object may do rather non-sensical things. To illustrate this, consider the following two interfaces:

```
interface Process {
    void launch(int stackSize, int dataSize);
};

// ...

interface Rocket {
    void launch(float xCoord, float yCoord);
};
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

```
Ice.ObjectPrx obj = ...;                    // Get proxy...
ProcessPrx process
    = ProcessPrxHelper.uncheckedCast(obj); // No worries...
process.launch(40, 60);                     // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

A final warning about down-casts: you must use either a `checkedCast` or an `uncheckedCast` to down-cast a proxy. If you use a C# cast, the behavior is undefined.

14.10.3 Using Proxy Methods

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy (see Section 32.11). Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second timeout as shown below:

```
Ice.ObjectPrx proxy = communicator.stringToProxy(...);
proxy = proxy.ice_timeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a `checkedCast` or `uncheckedCast` after using a factory method. However, a regular cast is still required, as shown in the example below:

```
Ice.ObjectPrx base = communicator.stringToProxy(...);
HelloPrx hello = HelloPrxHelper.checkedCast(base);
hello = (HelloPrx)hello.ice_timeout(10000); # Type is preserved
hello.sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

14.10.4 Object Identity and Proxy Comparison

As mentioned on page 475, proxies provide an `Equals` operation. Proxy comparison with `Equals` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `Equals` (or `==` and `!=`) tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

```
Ice.ObjectPrx p1 = ...;        // Get a proxy...
Ice.ObjectPrx p2 = ...;        // Get another proxy...

if (p1.Equals(p2)) {
    // p1 and p2 denote different objects       // WRONG!
} else {
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can

be contacted via several different transport endpoints). In other words, if two proxies compare equal with `Equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `Equals`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you must use a helper function in the `Ice.Util` class:

```
public sealed class Util {
    public static int proxyIdentityCompare(ObjectPrx lhs,
                                           ObjectPrx rhs);
    public static int proxyIdentityAndFacetCompare(ObjectPrx lhs,
                                                   ObjectPrx rhs);
// ...
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

```
Ice.ObjectPrx p1 = ...;        // Get a proxy...
Ice.ObjectPrx p2 = ...;        // Get another proxy...

if (Ice.Util.proxyIdentityCompare(p1, p2) != 0) {
    // p1 and p2 denote different objects       // Correct
} else {
    // p1 and p2 denote the same object         // Correct
}
```

The function returns 0 if the identities are equal, –1 if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name (see Chapter 33).

The C# mapping also provides two helper classes in the `Ice` namespace that allow you to insert proxies into hashtables or ordered collections, based on the identity, or the identity plus the facet name:

```
public class ProxyIdentityKey
    : System.Collections.IHashCodeProvider,
      System.Collections.IComparer {

    public int GetHashCode(object obj);
    public int Compare(object obj1, object obj2);
}

public class ProxyIdentityFacetKey
```

```
    : System.Collections.IHashCodeProvider,
      System.Collections.IComparer {

    public int GetHashCode(object obj);
    public int Compare(object obj1, object obj2);
}
```

Note these classes derive from `IHashCodeProvider` and `IComparer`, so they can be used for both hash tables and ordered collections.

14.11 Mapping for Operations

As we saw in Section 14.10, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system from Section 5.4:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

```
NodePrx node = ...;             // Initialize proxy
string name = node.name();      // Get name via RPC
```

This illustrates the typical pattern for receiving return values: return values are returned by reference for complex types, and by value for simple types (such as `int` or `double`).

14.11.1 Normal and `idempotent` Operations

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned `idempotent` has no effect. For example, consider the following interface:

```
interface Example {
                string op1();
    idempotent  string op2();
};
```

The proxy interface for this is:

```
public interface ExamplePrx : Ice.ObjectPrx
{
    string op1();
    string op2();
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to be mapped the same.

14.11.2 Passing Parameters

In-Parameters

The parameter passing rules for the C# mapping are very simple: parameters are passed either by value (for value types) or by reference (for reference types). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation (with some caveats—see page 1093).

Here is an interface with operations that pass parameters of various types from client to server:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following proxy for this definition:

```
public interface ClientToServerPrx : Ice.ObjectPrx
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, string[] ss,
             Dictionary<long, string[]> st);
    void op3(ClientToServerPrx proxy);
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

```
ClientToServerPrx p = ...;              // Get proxy...

p.op1(42, 3.14f, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14f;
bool b = true;
string s = "Hello world!";
p.op1(i, f, b, s);                      // Pass simple variables

NumberAndString ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
string[] ss = new string[1];
ss[0] = "Hello world!";
Dictionary<long, string[]> st
    = new Dictionary<long, string[]>();
st[0] = ss;
p.op2(ns, ss, st);                      // Pass complex variables

p.op3(p);                               // Pass proxy
```

Out-Parameters

Slice `out` parameters simply map to C# `out` parameters.

Here is the same Slice definition we saw on page 481 once more, but this time with all parameters being passed in the `out` direction:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;
```

```
interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Slice compiler generates the following code for this definition:

```
public interface ServerToClientPrx : Ice.ObjectPrx
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out string[] ss,
             out Dictionary<long, string[]> st);
    void op3(out ServerToClientPrx proxy);
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

```
ClientToServerPrx p = ...;              // Get proxy...

int i;
float f;
bool b;
string s;
p.op1(out i, out f, out b, out s);

NumberAndString ns;
string[] ss;
Dictionary<long, string[]> st;
p.op2(out ns, out ss, out st);

ServerToClientPrx stc;
p.op3(out stc);

System.Console.WriteLine(i);   // Show one of the values
```

Null Parameters

Some Slice types naturally have “empty” or “not there” semantics. Specifically, C# sequences (if mapped to `CollectionBase`), dictionaries, strings, and structures (if mapped to classes) all can be `null`, but the corresponding Slice types do not have the concept of a null value.

- Slice sequences, dictionaries, and strings cannot be null, but can be empty. To make life with these types easier, whenever you pass a C# `null` reference as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.
- If you pass a C# `null` reference to a Slice structure that is mapped to a C# class as a parameter or return value, the Ice run time automatically sends a structure whose elements are default-initialized. This means that all proxy members are initialized to `null`, sequence and dictionary members are initialized to empty collections, strings are initialized to the empty string, and members that have a value type are initialized to their default values.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are structures, sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid `NullReferenceExceptions`. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as `null` or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

14.12 Exception Handling

Any operation invocation may throw a run-time exception (see Section 14.9 on page 469) and, if the operation has an exception specification, may also throw user exceptions (see Section 14.9 on page 469). Suppose we have the following simple interface:

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as C# exceptions, so you can simply enclose one or more operation invocations in a `try–catch` block:

```
ChildPrx child = ...;   // Get child proxy...

try
{
    child.askToCleanUp();
}
catch (Tantrum t)
{
    System.Console.Write("The child says: ");
    System.Console.WriteLine(t.reason);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

```
public class Client
{
    private static void run() {
        ChildPrx child = ...;       // Get child proxy...
        try
        {
            child.askToCleanUp();
        }
        catch (Tantrum t)
        {
            System.Console.Write("The child says: ");
            System.Console.WriteLine(t.reason);
            child.scold();          // Recover from error...
        }
        child.praise();             // Give positive feedback...
    }

    static void Main(string[] args)
    {
        try
        {
            // ...
            run();
            // ...
        }
        catch (Ice.Exception e)
        {
```

```
            System.Console.WriteLine(e);
        }
    }
}
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Chapter 15.)

Note that the `ToString` method of exceptions prints the name of the exception, any inner exceptions, and the stack trace. Of course, you can be more selective in the way exceptions are displayed. For example, `e.GetType().Name` returns the (unscoped) name of an exception.

Exceptions and Out-Parameters

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out-parameters, Ice provides the weak exception guarantee [21] but does not provide the strong exception guarantee.[7]

14.13 Mapping for Classes

Slice classes are mapped to C# classes with the same name. By default, the generated class contains a public data member for each Slice data member (just as for structures and exceptions), and a member function for each operation.

Note that classes also support the `"clr:property"` metadata directive, so you can generate classes with virtual properties instead of data members. (See page 453 for details on "`clr:property`".)

Consider the following class definition:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

7. This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

The Slice compiler generates the following code for this definition:

```
public interface TimeOfDayOperations_
{
    string format(Ice.Current __current);
}

public interface TimeOfDayOperationsNC_
{
    string format();
}

public abstract partial class TimeOfDay
    : Ice.ObjectImpl,
      TimeOfDayOperations_,
      TimeOfDayOperationsNC_
{
    public short hour;
    public short minute;
    public short second;

    public TimeOfDay()
    {
    }

    public TimeOfDay(short hour, short minute, short second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    public string format()
    {
        return format(new Ice.Current());
    }

    public abstract string format(Ice.Current __current);
}
```

There are a number of things to note about the generated code:

1. The compiler generates “operations interfaces” called `TimeOfDayOperations_` and `TimeOfDayOperationsNC_`. These interfaces contain a method for each Slice operation of the class.

2. The generated class `TimeOfDay` inherits (indirectly) from `Ice.Object`. This means that all classes implicitly inherit from `Ice.Object`, which is the ultimate ancestor of all classes. Note that `Ice.Object` is *not* the same as `Ice.ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.

   If a class has only data members, but no operations, the compiler generates a non-abstract class.

3. The generated class contains a public member for each Slice data member.

4. The generated class inherits member functions for each Slice operation from the operations interfaces.

5. The generated class contains two constructors.

There is quite a bit to discuss here, so we will look at each item in turn.

14.13.1 Operations Interfaces

The methods in the `<interface-name>Operations_` interface have an additional trailing parameter of type `Ice.Current`, whereas the methods in the `<interface-name>OperationsNC_` interface lack this additional trailing parameter. The methods without the `Current` parameter simply forward to the methods with a `Current` parameter, supplying a default `Current`. For now, you can ignore this parameter and pretend it does not exist. (We look at it in more detail in Section 32.6.)

If a class has only data members, but no operations, the compiler omits generating the `<interface-name>Operations_` and `<interface-name>OperationsNC_` interfaces.

14.13.2 Inheritance from `Ice.Object`

Like interfaces, classes implicitly inherit from a common base class, `Ice.Object`. However, as shown in Figure 14.2, classes inherit from `Ice.Object` instead of `Ice.ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is

expected (and vice versa) because the base types for classes and proxies are not compatible.

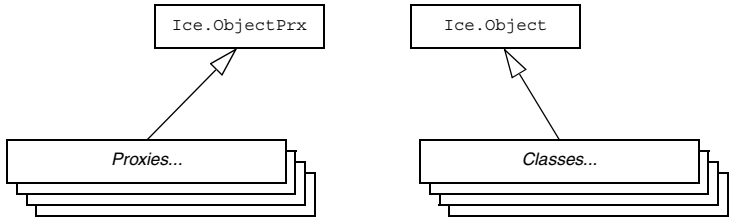


Figure 14.2. Inheritance from `Ice.ObjectPrx` and `Ice.Object`.

`Ice.Object` contains a number of member functions:

```
namespace Ice
{
    public interface Object : System.ICloneable
    {
        bool ice_isA(string s);
        bool ice_isA(string s, Current current);

        void ice_ping();
        void ice_ping(Current current);

        string[] ice_ids();
        string[] ice_ids(Current current);

        string ice_id();
        string ice_id(Current current);

        void ice_preMarshal();
        void ice_postUnmarshal();

        DispatchStatus ice_dispatch(
            Request request,
            DispatchInterceptorAsyncCallback cb);
    }
}
```

The member functions of `Ice.Object` behave as follows:

- `ice_isA`

  This function returns `true` if the object supports the given type ID, and `false` otherwise.

- `ice_ping`

  As for interfaces, `ice_ping` provides a basic reachability test for the class.

- `ice_ids`

  This function returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.

- `ice_id`

  This function returns the actual run-time type ID for a class. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.

- `ice_preMarshal`

  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.

- `ice_postUnmarshal`

  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.

- `ice_dispatch`

  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors (see Section 32.23).

Note that the generated class does *not* override `GetHashCode` and `Equals`. This means that classes are compared using shallow reference equality, not value equality (as is used for structures).

The class also provides a `Clone` method (whose implementation is inherited from `Ice.ObjectImpl`); the `Clone` method returns a shallow memberwise copy.

14.13.3 Data Members of Classes

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For

example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

```
class TimeOfDay {
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler produces the following generated code for this definition:

```
public abstract partial class TimeOfDay
    : Ice.ObjectImpl,
      TimeOfDayOperations_,
      TimeOfDayOperationsNC_
{
    protected short hour;
    protected short minute;
    protected short second;

    public TimeOfDay()
    {
    }

    public TimeOfDay(short hour, short minute, short second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // ...
}
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

```
["protected"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

If a protected data member also has the `clr:property` directive, the generated property has protected visibility. Consider the `TimeOfDay` class once again:

```
["protected", "clr:property"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The effects of combining these two metadata directives are shown in the generated code below:

```
public abstract partial class TimeOfDay
    : Ice.ObjectImpl,
      TimeOfDayOperations_,
      TimeOfDayOperationsNC_
{
    private short hour_prop;
    protected short hour {
        get {
            return hour_prop;
        }
        set {
            hour_prop = value;
        }
    }

    // ...
}
```

See page 453 for more information on the property mapping for data members.

14.13.4 Operations of Classes

Operations of classes are mapped to abstract member functions in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

```
public class TimeOfDayI : TimeOfDay
{
    public string format(Ice.Current current)
    {
        return   hour.ToString("D2") + ":"
```

```
                + minute.ToString("D2") + ":"
                + second.ToString("D2");
    }
}
```

Class Factories

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

```
class ObjectFactory : Ice.ObjectFactory
{
    public Ice.Object create(string type)
    {
```

```
        if (type.Equals(M.TimeOfDay.ice_staticId()))
            return new TimeOfDayI();
        System.Diagnostics.Debug.Assert(false);
        return null;
    }

    public void destroy()
    {
        // Nothing to do
    }
}
```

The object factory's `create` method is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` method is called by the Ice run time when its communicator is destroyed.

The `create` method is passed the type ID (see Section 4.13) of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::M::TimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `TimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoObjectFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

```
Ice.Communicator ic = ...;
ic.addObjectFactory(new ObjectFactory(),
                    M.TimeOfDay.ice_staticId());
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::M::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance, which returns a `TimeOfDayI` instance to the Ice run time.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator—if you do, the Ice run time has no

idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called. However, calls to `create` can be made concurrently.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

14.13.5 Class Constructors

Classes have a default constructor that default-constructs each data member. This means members of primitive type are initialized to the equivalent of zero, and members of reference type are initialized to null. Note that applications must always explicitly initialize a member whose type is a class-mapped structure because the Ice run time does not accept null as a legal value for these types.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition (see Section 4.11.1). The default constructor initializes each of these data members to its declared value.

Classes also provide a constructor that accepts one argument for each member of the class. This allows you to create and initialize a class in a single statement, for example:

```
TimeOfDayI tod = new TimeOfDayI(14, 45, 00); // 2:45pm
```

For derived classes, the constructor requires one argument of all of the members of the class, including inherited members. For example, consider the the definition from Section 4.11.2 once more:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};
```

```
class DateTime extends TimeOfDay {
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
};
```

The constructors for the generated classes are as follows:

```
public partial class TimeOfDay : Ice.ObjectImpl
{
    public TimeOfDay() {}

    public TimeOfDay(short hour, short minute, short second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // ...
}

public partial class DateTime : TimeOfDay
{
    public DateTime() : base() {}

    public DateTime(short hour,
                    short minute,
                    short second,
                    short day,
                    short month,
                    short year) : base(hour, minute, second)
    {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    // ...
}
```

If you want to instantiate and initialize a `DateTime` instance, you must either use the default constructor or provide values for all of the data members of the instance, including data members of any base classes.

14.14 Serializable Objects

As discussed in Section 4.18, you can pass CLR serializable objects directly as operation parameters or as fields of another data type. For example:

```
["clr:serializable:SomeNamespace.CLRClass"]
sequence<byte> CLRObj;
struct MyStruct {
    int i;
    CLRObj o;
};

interface Example {
    void op(CLRObj o, MyStruct s);
};
```

The generated code for `MyStruct` contains member `i` of type `int` and a member `o` of type `SomeNamespace.CLRClass`:

```
public partial class MyStruct : _System.ICloneable {
    public int i;
    SomeNamespace.CLRClass o;

    // ...
}
```

Similarly, the signature for `op` has parameters of type `CLRClass` and `MyStruct`:

```
void op(SomeNamespace.CLRClass o, MyStruct s);
```

Of course, your client and server code must have an implementation of `CLRClass` that sets the `Serializable` attribute:

```
namespace SomeNamespace {
    [Serializable]
    public class CLRClass {
        // ...
    }
}
```

You can implement this class in any way you see fit—the Ice run time does not place any other requirements on the implementation. However, note that the CLR requires the class to reside in the same assembly for client and server.

14.15 C#-Specific Metadata Directives

The **slice2cs** compiler supports metadata directives that allow you inject C# attribute specifications into the generated code. The metadata directive is `cs:attribute:`. For example:

```
["cs:attribute:System.Serializable"]
struct Point {
    double x;
    double y;
};
```

This results in the following code being generated for `S`:

```
[System.Serializable]
public partial struct Point
{
    public double x;
    public double y;
    // ...
}
```

You can apply this metadata directive to any slice construct, such as structure, operation, or parameter definitions.

You can use this directive also at global level. For example:

```
[["cs:attribute:assembly: AssemblyDescription(\"My assembly\")"]]
```

This results in the following code being inserted following any using directives and preceding any definitions:

```
[assembly: AssemblyDescription("My assembly")]
```

14.16 Asynchronous Method Invocation (AMI)

NOTE: As of version 3.4, Ice provides a new API for asynchronous method invocation. This section describes this API. You can find documentation for the previous API in Appendix K. Note that the old API is deprecated and will be removed in a future release.

Asynchronous Method Invocation (*AMI*) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both

oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

14.16.1 Basic Asynchronous API

Consider the following simple Slice definition:

```
module Demo {
    interface Employees {
        string getName(int number);
    };
};
```

Proxy Methods

Besides the synchronous proxy methods, **slice2cs** generates the following asynchronous proxy methods:[8]

```
public interface EmployeesPrx : Ice.ObjectPrx {
    Ice.AsyncResult<Demo.Callback_Employees_getName>
        begin_getName(int number);

    Ice.AsyncResult<Demo.Callback_Employees_getName>
        begin_getName(
            int number,
            _System.Collections.Generic.Dictionary
                                <string, string> ctx__);

    string end_getName(Ice.AsyncResult r__);
}
```

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. (The `begin_` method is overloaded so you can pass a per-invocation context—see Section 32.12).

8. There are two additional overloads of `begin_getName` that we discuss in Section 14.16.4.

- The `begin_getName` method sends (or queues) an invocation of `getName`. This method does not block the calling thread.
- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

```
EmployeesPrx e = ...;
Ice.AsyncResult r = e.begin_getName(99);

// Continue to do other things here...

string name = e.end_getName(r);
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `Ice.AsyncResult`. (The class derives from `System.IAsyncResult`.) This value contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResult` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. Similarly, the `end_` method has one out-parameter for each out-parameter of the corresponding Slice operation (plus the `AsyncResult` parameter). For example, consider the following operation:

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

```
Ice.AsyncResult<Demo.Callback_Employees_op>
    begin_op(int inp1, string inp2);

double end_op(out bool outp1,
              out long outp2,
              Ice.AsyncResult r__);
```

Exception Handling

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all

exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` method is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `CommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `System.ArgumentException`. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_` method, or if the `AsyncResult` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

14.16.2 The `AsyncResult` Class

The `AsyncResult` that is returned by the `begin_` method encapsulates the state of the asynchronous invocation:

```
public interface AsyncResult : System.IAsyncResult
{
    Ice.Communicator getCommunicator();
    Ice.Connection getConnection();
    ObjectPrx getProxy();
    string getOperation();
    object AsyncState { get; }


    bool IsCompleted { get; }
    void waitForCompleted();

    bool isSent();
    void waitForSent();

    bool sentSynchronously();

    AsyncResult whenSent(Ice.AsyncCallback cb);
    AsyncResult whenSent(Ice.SentCallback cb);
    AsyncResult whenCompleted(Ice.ExceptionCallback ex);
}
```

```
public interface AsyncResult<T> : AsyncResult
{
    AsyncResult<T> whenCompleted(T cb,
                                 Ice.ExceptionCallback excb);

    new AsyncResult<T> whenCompleted(Ice.ExceptionCallback excb);
    new AsyncResult<T> whenSent(Ice.SentCallback cb);
}
```

The methods and properties have the following semantics:

- `Communicator getCommunicator()`

  This method returns the communicator that sent the invocation.

- `Connection getConnection()`

  This method returns the connection that was used for the invocation.

- `ObjectPrx getProxy()`

  This method returns the proxy that was used to call the `begin_` method.

- `string getOperation()`

  This method returns the name of the operation.

- `object AsyncState { get; }`

  This property stores an object that you can use to pass shared state from the `begin_` to the `end_` method.

- `bool IsCompleted { get; }`

  This property is true if, at the time it is called, the result of an invocation is available, indicating that a call to the `end_` method will not block the caller. Otherwise, if the result is not yet available, the method returns false.

- `void waitForCompleted()`

  This method blocks the caller until the result of an invocation becomes available.

- `bool isSent()`

  When you call the `begin_` method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still in its queue, `isSent` returns false.

- `void waitForSent()`

  This method blocks the calling thread until a request has been written to the client-side transport.

- `bool sentSynchronously()`

  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSynchronously` returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

- ```
  AsyncResult whenSent(Ice.AsyncCallback cb)
  AsyncResult whenSent(Ice.SentCallback cb)
  AsyncResult<T> whenSent(Ice.SentCallback cb)
  AsyncResult whenCompleted(Ice.ExceptionCallback ex)
  AsyncResult<T> whenCompleted(
                  T cb,
                  Ice.ExceptionCallback excb)
  AsyncResult<T> whenCompleted(
                   Ice.ExceptionCallback excb)
  ```

  These methods allow you to specify callback methods that are called by the Ice run time. The `whenSent` methods set a callback that triggers when an asynchronous invocation is written to the client-side transport. The `whenCompleted` methods set a callback that triggers when an asynchronous invocation completes (see Section 14.16.4).

14.16.3 Polling for Completion

The `AsyncResult` methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
};
```

The client repeatedly calls `send` to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:
```

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
int offset = 0;
while (!file.eof()) {
    byte[] bs;
    bs = file.read(chunkSize); // Read a chunk
    ft.send(offset, bs);       // Send the chunk
    offset += bs.Length;
}
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing—the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

```
FileHandle file = open(...);
FileTransferPrx ft = ...;
const int chunkSize = ...;
int offset = 0;

LinkedList<Ice.AsyncResult> results =
    new LinkedList<Ice.AsyncResult>();
const int numRequests = 5;

while (!file.eof()) {
    byte[] bs;
    bs = file.read(chunkSize);

    // Send up to numRequests + 1 chunks asynchronously.
    Ice.AsyncResult r = ft.begin_send(offset, bs);
    offset += bs.Length;

    // Wait until this request has been passed to the transport.
    r.waitForSent();
    results.AddLast(r);

    // Once there are more than numRequests, wait for the least
    // recent one to complete.
    while (results.Count > numRequests) {
        Ice.AsyncResult r = results.First;
        results.RemoveFirst();
```

```
        r.waitForCompleted();
    }
}

// Wait for any remaining requests to complete.
while (results.Count > 0) {
    Ice.AsyncResult r = results.First;
    results.RemoveFirst();
    r.waitForCompleted();
}
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

14.16.4 Generic Completion Callbacks

The `begin_` method is overloaded to allow you to provide completion callbacks. Here are the corresponding methods for the `getName` operation:

```
Ice.AsyncResult begin_getName(
        int number,
        Ice.AsyncCallback cb__,
        object cookie__);

Ice.AsyncResult begin_getName(
        int number,
        _System.Collections.Generic.Dictionary
                            <string, string> ctx__,
        Ice.AsyncCallback cb__,
        object cookie__);
```

The second version of `begin_getName` lets you override the default context. (We discuss the purpose of the `cookie` parameter in the next section.) Following the in-parameters, the `begin_` method accepts a parameter of type `Ice.AsyncCallback`, which is a delegate for a callback method. The Ice run time invokes the callback method when an asynchronous operation completes. Your callback method must have `void` return type and accept a single parameter of type `AsyncResult`, for example:

```
private class MyCallback
{
    public void finished(Ice.AsyncResult r)
    {
        EmployeesPrx e = (EmployeesPrx)r.getProxy();
        try {
            string name = e.end_getName(r);
            System.Console.WriteLine("Name is: " + name);
        } catch (Ice.Exception ex) {
            System.Console.Err.WriteLine("Exception is: " + ex);
        }
    }
}
```

The implementation of your callback method must call the `end_` method. The proxy for the call is available via the `getProxy` method on the `AsyncResult` that is passed by the Ice run time. The return type of `getProxy` is `Ice.ObjectPrx`, so you must down-cast the proxy to its correct type.

Your callback method should catch and handle any exceptions that may be thrown by the `end_` method. If you allow an exception to escape from the callback method, the Ice run time produces a log entry by default and ignores the exception. (You can disable the log message by setting the property `Ice.Warn.AMICallback` to zero.)

To inform the Ice run time that you want to receive a callback for the completion of the asynchronous call, you pass a delegate for your callback method to the `begin_` method:

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
Ice.AsyncCallback del = new Ice.AsyncCallback(cb.finished);

e.begin_getName(99, del, null);
```

The trailing `null` argument specifies a cookie, which we will discuss shortly.

You can avoid explicit instantiation of the delegate and, more tersely, write:

```
EmployeesPrx e = ...;

MyCallback cb = new MyCallback();
e.begin_getName(99, cb.finished, null);
```

Using Cookies

It is common for the `end_` method to require access to some state that is established by the code that calls the `begin_` method. As an example, consider an application that asynchronously starts a number of operations and, as each operation completes, needs to update different user interface elements with the results. In this case, the `begin_` method knows which user interface element should receive the update, and the `end_` method needs access to that element.

The API allows you to pass such state by providing a cookie. A cookie is any class instance; the class can contain whatever data you want to pass, as well as any methods you may want to add to manipulate that data.

Here is an example implementation that stores a `Widget`. (We assume that this class provides whatever methods are needed by the `end_` method to update the display.) When you call the `begin_` method, you pass the appropriate cookie instance to inform the `end_` method how to update the display:

```
// Invoke the getName operation with different widget cookies.
e.begin_getName(99, getNameCB, widget1);
e.begin_getName(24, getNameCB, widget2);
```

The `end_` method can retrieve the cookie from the `AsyncResult` by reading the `AsyncState` property. For this example, we assume that widgets have a `writeString` method that updates the relevant UI element:

```
public void getNameCB(Ice.AsyncResult r)
{
    EmployeesPrx e = (EmployeesPrx)r.getProxy();
    Widget widget = (Widget)r.AsyncState;
    try {
        string name = e.end_getName(r);
        widget.writeString(name);
    } catch (Ice.Exception ex) {
        handleException(ex);
    }
}
```

The cookie provides a simple and effective way for you to pass state between the point where an operation is invoked and the point where its results are processed.

Moreover, if you have a number of operations that share common state, you can pass the same cookie instance to multiple invocations.

14.16.5 Type-Safe Completion Callbacks

The generic callback API we saw in Section 14.16.4 is not entirely type-safe:

- You must down-cast the return value of `getProxy` to the correct proxy type before you can call the `end_` method.
- You must call the correct `end_` method to match the operation called by the `begin_` method.
- You must remember to catch exceptions when you call the `end_` method; if you forget to do this, you will not know that the operation failed.

`slice2cs` generates an additional type-safe API that takes care of these chores for you. To use type-safe callbacks, you must implement a callback class that provides two callback methods:

- a success callback that is called if the operation succeeds
- a failure callback that is called if the operation raises an exception

Here is a callback class for an invocation of the `getName` operation:

```
public class MyCallback
{
    public void getNameCB(string name)
    {
        System.Console.WriteLine("Name is: " + name);
    }

    public void failureCB(Ice.Exception ex)
    {
        System.Console.Err.WriteLine("Exception is: " + ex);
    }
}
```

The callback methods can have any name you prefer and must have `void` return type. The failure callback always has a single parameter of type `Ice.Exception`. The success callback parameters depend on the operation signature. If the operation has non-`void` return type, the first parameter of the success callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

At the calling end, you call the `begin_` method as follows:

```
MyCallback cb = new MyCallback();

e.begin_getName(99).whenCompleted(cb.getNameCB, cb.failureCB);
```

Note the `whenCompleted` method on the `AsyncResult` that is returned by the `begin_` method. This method establishes the link between the `begin_` method and the callbacks that are called by the Ice run time by setting the delegates for the success and failure methods.

It is legal to pass a null delegate for the success or failure methods. For the success callback, this is legal only for operations that have `void` return type and no out-parameters. This is useful if you do not care when the operation completes but want to know if the call failed. If you pass a null exception delegate, the Ice run time will ignore any exception that is raised by the invocation.

Using Cookies

The type-safe API does not support cookies. If you want to pass state from the `begin_` method to the `end_` method, you must use the generic API or, alternatively, place the state into the callback class containing the callback methods. Here is a simple implementation of a callback class that stores a widget that can be retrieved by the `end_` method:

```
public class MyCallback
{
    public MyCallback(Widget w)
    {
        _w = w;
    }

    private Widget _w;

    public void getNameCB(string name)
    {
        _w.writeString(name);
    }

    public void failureCB(Ice.Exception ex)
    {
        _w.writeError(ex);
    }
}
```

When you call the `begin_` method, you pass the appropriate callback instance to inform the `end_` method how to update the display:

```
EmployeesPrx e = ...;
Widget widget1 = ...;
Widget widget2 = ...;

// Invoke the getName operation with different widget callbacks.

e.begin_getName(99,
                new MyCallback(widget1)).
                    whenCompleted(getNameCB, failureCB);

e.begin_getName(24,
                new MyCallback(widget2)).
                    whenCompleted(getNameCB, failureCB);
```

14.16.6 Oneway Invocations

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws a `System.ArgumentException`.

For the generic API, the callback method looks exactly as for a twoway invocation. However, for oneway invocations, the Ice run time does not call the callback method unless the invocation raised an exception during the `begin_` method ("on the way out").

For the type-safe API, you only specify a delegate for the failure method. For example, here is how you could call `ice_ping` asynchronously:

```
ObjectPrx p = ...;
MyCallback cb = new MyCallback();
p.begin_ice_ping().whenCompleted(cb.failureCB);
```

14.16.7 Flow Control

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `AsyncResult.sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `AsyncResult.sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

For the generic API, you can create an additional callback method:

```
public class MyCallback
{
    public void finished(Ice.AsyncResult r)
    {
        // ...
    }

    public void sent(Ice.AsyncResult r)
    {
        // ...
    }
}
```

As with any other callback method, you are free to choose any name you like. For this example, the name of the callback method is `sent`. You inform the Ice run time that you want to be informed when a call has been passed to the local transport by calling `whenSent`:

```
MyCallback cb = new MyCallback();

e.begin_getName(99).whenCompleted(cb.getNameCB,
                                  cb.failureCB).whenSent(cb.sent);
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the `sent` method from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the `sent` method from a different thread once it has written the request to the local transport. In addition, you can find out from the `AsyncResult` that is returned by the `begin_` method whether the request was sent synchronously or was queued, by calling `sentSynchronously`.

For the generic API, the `sent` method has the following signature:

```
void sent(Ice.AsyncResult r);
```

For the type-safe API, the signature is:

```
void sent(bool sentSynchronously);
```

For the generic API, you can find out whether the request was sent synchronously by calling `sentSynchronously` on the `AsyncResult`. For the type-safe API, the boolean `sentSynchronously` parameter provides the same information.

The `sent` methods allow you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

14.16.8 Batch Requests

Applications that send batched requests (see Section 32.16) can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests sent queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `AsyncResult.getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

14.16.9 Concurrency

The Ice run time always invokes your callback methods from a separate thread, with one exception: it calls the `sent` callback from the thread calling the `begin_` method if the request could be sent synchronously. In the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` member or parameter.

14.16.10 Limitations

AMI invocations cannot be sent using collocated optimization. If you attempt to invoke an AMI operation using a proxy that is configured to use collocation optimization, the Ice run time raises `CollocationOptimizationException` if the

servant happens to be collocated; the request is sent normally if the servant is not collocated. Section 32.21 provides more information about this optimization and describes how to disable it when necessary.

14.17 slice2cs Command-Line Options

The Slice-to-C# compiler, **slice2cs**, offers the following command-line options in addition to the standard options described in Section 4.20:

- **--tie**

  Generate tie classes (see Section 16.7).
- **--impl**

  Generate sample implementation files. This option will not overwrite an existing file.
- **--impl-tie**

  Generate sample implementation files using ties (see Section 16.7). This option will not overwrite an existing file.
- **--checksum**

  Generate checksums for Slice definitions.
- **--stream**

  Generate streaming helper functions for Slice types (see Section 35.2).

14.18 Using Slice Checksums

As described in Section 4.21, the Slice compilers can optionally generate checksums of Slice definitions. For **slice2cs**, the **--checksum** option causes the compiler to generate checksums in each C# source file that are added to a member of the `Ice.SliceChecksums` class:

```
namespace Ice {
    public sealed class SliceChecksums {
        public readonly static SliceChecksumDict checksums;
    };
}
```

The `checksums` map is initialized automatically prior to first use; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the dictionaries using the `Equals` function. However, this is not feasible if it is possible that the server might be linked with more Slice definitions than the client. A more general solution is to iterate over the local checksums as demonstrated below:

```
Ice.SliceChecksumDict serverChecksums = ...
foreach(System.Collections.DictionaryEntry e
        in Ice.SliceChecksums.checksums) {
    string checksum = serverChecksums[e.Key];
    if (checksum == null) {
        // No match found for type id!
    } else if (!checksum.Equals(e.Value)) {
        // Checksum mismatch!
    }
}
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

Chapter 15
Developing a File System Client in C#

15.1 Chapter Overview

In this chapter, we present the source code for a C# client that accesses the file system we developed in Chapter 5 (see Chapter 17 for the corresponding server).

15.2 The C# Client

We now have seen enough of the client-side C# mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
```

```
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

```
using System;
using Filesystem;

public class Client
{
    // Recursively print the contents of directory "dir"
    // in tree fashion. For files, show the contents of
    // each file. The "depth" parameter is the current
    // nesting level (for indentation).

    static void listRecursive(DirectoryPrx dir, int depth)
    {
        string indent = new string('\t', ++depth);

        NodePrx[] contents = dir.list();

        foreach (NodePrx node in contents)
            DirectoryPrx subdir
                = DirectoryPrxHelper.checkedCast(node);
            FilePrx file = FilePrxHelper.uncheckedCast(node);
            Console.WriteLine(indent + node.name() +
                (subdir != null ? " (directory):" : " (file):"));
            if (subdir != null) {
                listRecursive(subdir, depth);
            } else {
                string[] text = file.read();
                for (int j = 0; j < text.Length; ++j)
                    Console.WriteLine(indent + "\t" + text[j]);
```

```
                }
            }
        }

        public static void Main(string[] args)
        {
            int status = 0;
            Ice.Communicator ic = null;
            try {
                // Create a communicator
                //
                ic = Ice.Util.initialize(ref args);

                // Create a proxy for the root directory
                //
                Ice.ObjectPrx obj
                    = ic.stringToProxy("RootDir:default -p 10000");

                // Down-cast the proxy to a Directory proxy
                //
                DirectoryPrx rootDir
                    = DirectoryPrxHelper.checkedCast(obj);

                // Recursively list the contents of the root directory
                //
                Console.WriteLine("Contents of root directory:");
                listRecursive(rootDir, 0);
            } catch (Exception e) {
                Console.Error.WriteLine(e);
                status = 1;
            }
            if (ic != null) {
                // Clean up
                //
                try {
                    ic.destroy();
                } catch (Exception e) {
                    Console.Error.WriteLine(e);
                    status = 1;
                }
            }
            Environment.Exit(status);
        }
    }
```

The `Client` class defines two methods: `listRecursive`, which is a helper function to print the contents of the file system, and `Main`, which is the main program. Let us look at `Main` first:

1. The structure of the code in `Main` follows what we saw in Chapter 3. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.

2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.

   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.

2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.

3. The code checks the type of the node:

   - If it is a directory, the code recurses, incrementing the indent level.

   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:

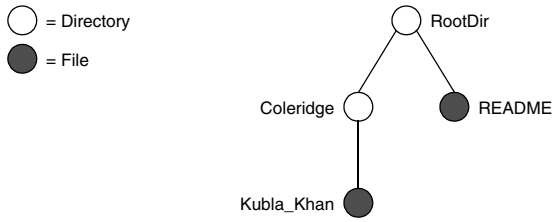


Figure 15.1. A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in Chapter 38 and Chapter 34.

15.3 Summary

This chapter presented a very simple client to access a server that implements the file system we developed in Chapter 5. As you can see, the C# code hardly differs from the code you would write for an ordinary C# program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local C# object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with

arcane networking APIs. As we will see in Chapter 17, this is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

Chapter 16
Server-Side Slice-to-C# Mapping

16.1 Chapter Overview

In this chapter, we present the server-side Slice-to-C# mapping (see Chapter 14 for the client-side mapping). Section 16.3 discusses how to initialize and finalize the server-side run time, sections 16.4 to 16.7 show how to implement interfaces and operations, and Section 16.8 discusses how to register objects with the server-side Ice run time. Finally, Section 16.9 shows how to implement operations asynchronously.

16.2 Introduction

The mapping for Slice data types to C# is identical on the client side and server side. This means that everything in Chapter 14 also applies to the server side. However, for the server side, there are a few additional things you need to know, specifically:

- how to initialize and finalize the server-side run time
- how to implement servants
- how to pass parameters and throw exceptions
- how to create servants and register them with the Ice run time.

We discuss these topics in the remainder of this chapter.

16.3 The Server-Side `Main` Method

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. As for the client side, you must initialize the Ice run time by calling `Ice.Util.initialize` before you can do anything else in your server. `Ice.Util.initialize` returns a reference to an instance of an `Ice.Communicator`:

```
using System;

public class Server
{
    public static void Main(string[] args)
    {
        int status = 0;
        Ice.Communicator communicator = null;

        try {
            communicator = Ice.Util.initialize(ref args);
            // ...
        } catch (Exception ex) {
            Console.Error.WriteLine(ex);
            status = 1;
        }
        // ...
    }
}
```

`Ice.Util.initialize` accepts the argument vector that is passed to `Main` by the operating system. The method scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `Ice.Util.initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

Before leaving your `Main` method, you *must* call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, `destroy` waits for any operation implementations that are still executing in the server to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file

descriptors and memory. Never allow your Main method to terminate without calling destroy first; doing so has undefined behavior.

The general shape of our server-side Main method is therefore as follows:

```
using System;

public class Server
{
    public static void Main(string[] args)
    {
        int status = 0;
        Ice.Communicator communicator = null;

        try {
            communicator = Ice.Util.initialize(ref args);
            // ...
        } catch (Exception ex) {
            Console.Error.WriteLine(ex);
            status = 1;
        }
        if (communicator != null) {
            try {
                communicator.destroy();
            } catch (Exception ex) {
                Console.Error.WriteLine(ex);
                status = 1;
            }
        }
        Environment.Exit(status);
    }
}
```

Note that the code places the call to Ice.Util.initialize into a try block and takes care to return the correct exit status to the operating system. Also note that an attempt to destroy the communicator is made only if the initialization succeeded.

16.3.1 The Ice.Application Class

The preceding structure for the Main method is so common that Ice offers a class, Ice.Application, that encapsulates all the correct initialization and finalization activities. The synopsis of the class is as follows (with some detail omitted for now):

```
namespace Ice
{
    public abstract class Application
    {
        public abstract int run(string[] args);

        public Application();

        public Application(SignalPolicy signalPolicy);

        public int main(string[] args);
        public int main(string[] args, string configFile);
        public int main(string[] args, InitializationData init);

        public static string appName();

        public static Communicator communicator();
    }
}
```

The intent of this class is that you specialize `Ice.Application` and implement the abstract `run` method in your derived class. Whatever code you would normally place in `Main` goes into the `run` method instead. Using `Ice.Application`, our program looks as follows:

```
using System;

public class Server
{
    class App : Ice.Application
    {
        public override int run(string[] args)
        {
            // Server code here...

            return 0;
        }
    }

    public static void Main(string[] args)
    {
        App app = new App();
        Environment.Exit(app.main(args));
    }
}
```

Note that `Application.main` is overloaded: you can pass an optional file name or an `InitializationData` structure (see Section 32.3 and Section 30.9).

If you pass a configuration file name to `main`, the settings in this file are overridden by settings in a file identified by the `ICE_CONFIG` environment variable (if defined). Property settings supplied on the command line take precedence over all other settings.

The `Application.main` method does the following:

1. It installs an exception handler for `System.Exception`. If your code fails to handle an exception, `Application.main` prints the name of the exception and a stack trace on `Console.Error` before returning with a non-zero return value.
2. It initializes (by calling `Ice.Util.initialize`) and finalizes (by calling `Communicator.destroy`) a communicator. You can get access to the communicator for your server by calling he static `communicator` accessor.
3. It scans the argument vector for options that are relevant to the Ice run time and removes any such options. The argument vector that is passed to your `run` method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.
4. It provides the name of your application via the static `appName` method. You can get at the application name from anywhere in your code by calling `Ice.Application.appName` (which is usually required for error messages).
5. It installs a signal handler that properly destroys the communicator.
6. It installs a per-process logger (see Section 32.19.5) if the application has not already configured one. The per-process logger uses the value of the `Ice.ProgramName` property (see Section 30.8) as a prefix for its messages and sends its output to the standard error channel. An application can specify an alternate logger as described in Section 32.19.

Using `Ice.Application` ensures that your program properly finalizes the Ice run time, whether your server terminates normally or in response to an exception. We recommend that all your programs use this class; doing so makes your life easier. In addition `Ice.Application` also provides features for signal handling and configuration that you do not have to implement yourself when you use this class.

Using `Ice.Application` on the Client Side

You can use `Ice.Application` for your clients as well: simply implement a class that derives from `Ice.Application` and place the client code into its `run` method. The advantage of this approach is the same as for the server side: `Ice.Application` ensures that the communicator is destroyed correctly even in the presence of exceptions.

Catching Signals

The simple server we developed in Chapter 3 had no way to shut down cleanly: we simply interrupted the server from the command line to force it to exit. Terminating a server in this fashion is unacceptable for many real-life server applications: typically, the server has to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is particularly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

To make it easier to deal with signals, `Ice.Application` encapsulates the low-level signal handling tasks, allowing you to cleanly shut down on receipt of a signal.

```
namespace Ice
{
    public abstract class Application
    {
        // ...

        public static void destroyOnInterrupt();
        public static void shutdownOnInterrupt();
        public static void ignoreInterrupt();
        public static void callbackOnInterrupt();
        public static void holdInterrupt();
        public static void releaseInterrupt();

        public static bool interrupted();

        public virtual void interruptCallback(int sig);
    }
}
```

The methods behave as follows:

- `destroyOnInterrupt`

  This method installs a handler that destroys the communicator if it is interrupted. This is the default behavior.

- `shutdownOnInterrupt`

  This method installs a handler that shuts down the communicator if it is interrupted.
- `ignoreInterrupt`

  This method causes signals to be ignored.
- `callbackOnInterrupt`

  This method configures `Ice.Application` to invoke `interruptCallback` when a signal occurs, thereby giving the subclass responsibility for handling the signal.
- `holdInterrupt`

  This method temporarily blocks signal delivery.
- `releaseInterrupt`

  This method restores signal delivery to the previous disposition. Any signal that arrives after `holdInterrupt` was called is delivered when you call `releaseInterrupt`.
- `interrupted`

  This method returns `true` if a signal caused the communicator to shut down, `false` otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by a signal. This is useful, for example, for logging purposes.
- `interruptCallback`

  A subclass overrides this method to respond to signals. The method may be called concurrently with any other thread and must not raise exceptions.

By default, `Ice.Application` behaves as if `destroyOnInterrupt` was invoked, therefore our server `Main` method requires no change to ensure that the program terminates cleanly on receipt of a signal. (You can disable the signal-handling functionality of `Ice.Application` by passing the enumerator `NoSignalHandling` to the constructor. In that case, signals retain their default behavior, that is, terminate the process.) However, we add a diagnostic to report the occurrence, so our `run` method now looks like:

```
using System;

public class Server
{
    class App : Ice.Application
    {
```

```
        public override int run(string[] args)
        {
            // Server code here...

            if (interrupted())
                Console.Error.WriteLine(
                    appName() + ": terminating");

            return 0;
        }
    }

    public static void Main(string[] args)
    {
        App app = new App();
        Environment.Exit(app.main(args));
    }
}
```

`Ice.Application` and Properties

Apart from the functionality shown in this section, `Ice.Application` also takes care of initializing the Ice run time with property values. Properties allow you to configure the run time in various ways. For example, you can use properties to control things such as the thread pool size or port number for a server. The `main` method of `Ice.Application` is overloaded; the second version allows you to specify the name of a configuration file that will be processed during initialization. We discuss Ice properties in more detail in Chapter 30.

Limitations of `Ice.Application`

`Ice.Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use `Ice.Application`. Instead, you must structure your code as we saw in Chapter 3 (taking care to always destroy the communicator).

16.4 Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing methods in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

16.4.1 Skeleton Classes

On the client side, interfaces map to proxy classes (see Section 5.12). On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has an abstract method for each operation on the corresponding interface. For example, consider the Slice definition for the `Node` interface we defined in Chapter 5 once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The Slice compiler generates the following definition for this interface:

```
namespace Filesystem
{
    public interface NodeOperations_
    {
        string name(Ice.Current __current);
    }

    public interface NodeOperationsNC_
    {
        string name();
    }

    public interface Node : Ice.Object,
                            NodeOperations_, NodeOperationsNC_
    {
    }

    public abstract class NodeDisp_ : Ice.ObjectImpl, Node
    {
        public string name()
        {
            return name(new Ice.Current());
        }

        public abstract string name(Ice.Current __current);

        // Mapping-internal code here...
    }
}
```

The important points to note here are:

- As for the client side, Slice modules are mapped to C# namespaces with the same name, so the skeleton class definitions are part of the `Filesystem` namespace.
- For each Slice interface *`<interface-name>`*, the compiler generates C# interfaces *`<interface-name>`*`Operations_` and *`<interface-name>`*`OperationsNC_` (`NodeOperations_` and `NodeOperationsNC_` in this example). These interfaces contain a method for each operation in the Slice interface. (You can ignore the `Ice.Current` parameter for the time being—we discuss it in detail in Section 32.6.)
- For each Slice interface *`<interface-name>`*, the compiler generates a C# interface *`<interface-name>`* (`Node` in this example). That interface extends `Ice.Object` and the two operations interfaces.
- For each Slice interface *`<interface-name>`*, the compiler generates an abstract class *`<interface-name>`*`Disp_` (`NodeDisp_` in this example). This abstract class is the actual skeleton class; it is the base class from which you derive your servant class.

16.4.2 Servant Classes

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

```
public class NodeI : NodeDisp_
{
    public NodeI(string name)
    {
        _name = name;
    }

    public override string name(Ice.Current current)
    {
        return _name;
    }

    private string _name;
}
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention

only: as far as the Ice run time is concerned, you can chose any name you prefer for your servant classes.) Note that `NodeI` extends `NodeDisp_`, that is, it derives from its skeleton class.

As far as Ice is concerned, the `NodeI` class must implement only a single method: the abstract `name` method that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other methods and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. (Obviously, the constructor initializes the `_name` member and the `name` method returns its value.)

Normal and `idempotent` Operations

Whether an operation is an ordinary operation or an `idempotent` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

```
interface Example {
   void              normalOp();
   idempotent void   idempotentOp();
};
```

The operations class for this interface looks like this:

```
public interface ExampleOperations_
{
    void normalOp(Ice.Current __current);
    void idempotentOp(Ice.Current __current);
}
```

Note that the signatures of the methods are unaffected by the `idempotent` qualifier.

16.5 Parameter Passing

For each parameter of a Slice operation, the C# mapping generates a corresponding parameter for the corresponding method in the *<interface-name>*`Operations_` interface. In addition, every operation has an additional, trailing parameter of type `Ice.Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` method of the `NodeOperations_` interface has a single parameter of type

`Ice.Current`. We explain the purpose of this parameter in Section 32.6 and will ignore it for now.

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

```
module M {
    interface Example {
        string op(string sin, out string sout);
    };
};
```

The generated method for `op` looks as follows:

```
public interface ExampleOperations_
{
    string op(string sin, out string sout, Ice.Current __current);
}
```

As you can see, there are no surprises here. For example, we could implement `op` as follows:

```
using System;

public class ExampleI : ExampleDisp_
{
    public override string op(string sin, out string sout,
                              Ice.Current current)
    {
        Console.WriteLine(sin);      // In params are initialized
        sout = "Hello World!";       // Assign out param
        return "Done";
    }
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a method; the fact that remote procedure calls are involved does not affect your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal C# rules and do not require special-purpose API calls.

16.6 Raising Exceptions

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

```
// ...
public override void write(string[] text, Ice.Current current)
{
    try
    {
        // Try to write file contents here...
    }
    catch(System.Exception ex)
    {
        GenericError e = new GenericError("cannot write file", ex)
;
        e.reason = "Exception during write operation";
        throw e;
    }
}
```

Note that, for this example, we have supplied the optional second parameter to the `GenericError` constructor (see Section 14.9). This parameter sets the `InnerException` member of `System.Exception` and preserves the original cause of the error for later diagnosis.

```
// ...

public void
write(String[] text, Ice.Current current)
    throws GenericError

{
    // Try to write file contents here...
    // Assume we are out of space...
    if (error) {
        GenericError e = new GenericError();
        e.reason = "file too large";
        throw e;
    }
}
```

If you throw an arbitrary C# run-time exception (such as an `InvalidCastException`), the Ice run time catches the exception and then returns an `UnknownException` to the client. Similarly, if you throw an "impossible" user exception (a user exception that is not listed in the exception specification of the operation), the client receives an `UnknownUserException`.

If you throw an Ice run-time exception, such `MemoryLimitException`, the client receives an `UnknownLocalException`.[1] For that reason, you should never throw system exceptions from operation implementations. If you do, all the client

will see is an `UnknownLocalException`, which does not tell the client anything useful.

16.7 Tie Classes

The mapping to skeleton classes we saw in Section 16.4 requires the servant class to inherit from its skeleton class. Occasionally, this creates a problem: some class libraries require you to inherit from a base class in order to access functionality provided by the library; because C# does not support multiple implementation inheritance, this means that you cannot use such a class library to implement your servants because your servants cannot inherit from both the library class and the skeleton class simultaneously.

To allow you to still use such class libraries, Ice provides a way to write servants that replaces inheritance with delegation. This approach is supported by *tie classes*. The idea is that, instead of inheriting from the skeleton class, you simply create a class (known as an *implementation class* or *delegate class*) that contains methods corresponding to the operations of an interface. You use the **`--tie`** option with the **`slice2cs`** compiler to create a tie class. For example, for the `Node` interface we saw in Section 16.4.1, the **`--tie`** option causes the compiler to create exactly the same code as we saw previously, but to also emit an additional tie class. For an interface *`<interface-name>`*, the generated tie class has the name *`<interface-name>`*`Tie_`:

```
public class NodeTie_ : NodeDisp_, Ice.TieBase
{
    public NodeTie_()
    {
    }

    public NodeTie_(NodeOperations_ del)
    {
        _ice_delegate = del;
    }

    public object ice_delegate()
```

1. There are three run-time exceptions that are not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`. We discuss these exceptions in more detail in Chapter 33.

```
    {
        return _ice_delegate;
    }

    public void ice_delegate(object del)
    {
        _ice_delegate = (NodeOperations_)del;
    }

    public override int GetHashCode()
    {
        return _ice_delegate == null
                                ? 0
                                : _ice_delegate.GetHashCode();
    }

    public override bool Equals(object rhs)
    {
       if (object.ReferenceEquals(this, rhs))
       {
           return true;
       }
       if (!(rhs is NodeTie_))
       {
           return false;
       }
       if (_ice_delegate == null)
       {
           return ((NodeTie_)rhs)._ice_delegate == null;
       }
       return _ice_delegate.Equals(((NodeTie_)rhs)._ice_delegate);
    }

    public override string name(Ice.Current __current)
    {
        return _ice_delegate.name(__current);
    }

    private NodeOperations_ _ice_delegate;
}
```

This looks a lot worse than it is: in essence, the generated tie class is simply a servant class (it extends `NodeDisp_`) that delegates each invocation of a method

that corresponds to a Slice operation to your implementation class (see Figure 16.1).

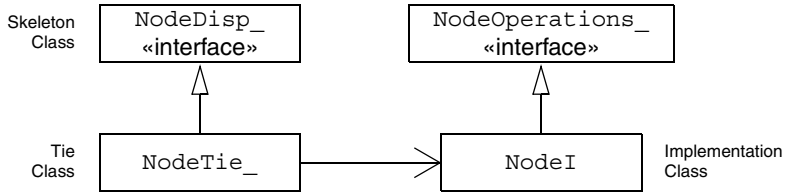


Figure 16.1. A skeleton class, tie class, and implementation class.

The `Ice.TieBase` interface defines the `ice_delegate` methods that allow you to get and set the delegate.

Given this machinery, we can create an implementation class for our `Node` interface as follows:

```
public class NodeI : NodeOperations_
{
    public NodeI(string name)
    {
        _name = name;
    }

    public override string name(Ice.Current current)
    {
        return _name;
    }

    private string _name;
}
```

Note that this class is identical to our previous implementation, except that it implements the `NodeOperations_` interface and does not extend `NodeDisp_` (which means that you are now free to extend any other class to support your implementation).

To create a servant, you instantiate your implementation class and the tie class, passing a reference to the implementation instance to the tie constructor:

```
NodeI fred = new NodeI("Fred");          // Create implementation
NodeTie_ servant = new NodeTie_(fred);   // Create tie
```

Alternatively, you can also default-construct the tie class and later set its delegate instance by calling `ice_delegate`:

```
NodeTie_ servant = new NodeTie_();      // Create tie
// ...
NodeI fred = new NodeI("Fred");         // Create implementation
// ...
servant.ice_delegate(fred);             // Set delegate
```

When using tie classes, it is important to remember that the tie instance is the servant, not your delegate. Furthermore, you must not use a tie instance to incarnate (see Section 16.8) an Ice object until the tie has a delegate. Once you have set the delegate, you must not change it for the lifetime of the tie; otherwise, undefined behavior results.

You should use the tie approach only if you need to, that is, if you need to extend some base class in order to implement your servants: using the tie approach is more costly in terms of memory because each Ice object is incarnated by two C# objects instead of one, the tie and the delegate. In addition, call dispatch for ties is marginally slower than for ordinary servants because the tie forwards each operation to the delegate, that is, each operation invocation requires two function calls instead of one.

Also note that, unless you arrange for it, there is no way to get from the delegate back to the tie. If you need to navigate back to the tie from the delegate, you can store a reference to the tie in a member of the delegate. (The reference can, for example, be initialized by the constructor of the delegate.)

16.8 Object Incarnation

Having created a servant class such as the rudimentary `NodeI` class in Section 16.4.2, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

16.8.1 Instantiating a Servant

Instantiating a servant means to allocate an instance:

```
Node servant = new NodeI("Fred");
```

This code creates a new `NodeI` instance and assigns its address to a reference of type `Node`. This works because `NodeI` is derived from `Node`, so a `Node` reference can refer to an instance of type `NodeI`. However, if we want to invoke a method of the `NodeI` class at this point, we must use a `NodeI` reference:

```
NodeI servant = new NodeI("Fred");
```

Whether you use a `Node` or a `NodeI` reference depends purely on whether you want to invoke a method of the `NodeI` class: if not, a `Node` reference works just as well as a `NodeI` reference.

16.8.2 Creating an Identity

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.[2] An Ice object identity is a structure with the following Slice definition:

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
    // ...
};
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (See Section 32.6 for a discussion of the `category` field.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

```
Ice.Identity id = new Ice.Identity();
id.name = "Fred"; // Not unique, but good enough for now
```

2. The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity. See Chapter 34 for further discussion.

16.8.3 Activating a Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the add operation on the object adapter. Assuming that we have access to the object adapter in the `_adapter` variable, we can write:

```
_adapter.add(servant, id);
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct method on the servant.

Assuming that the object adapter is in the active state (see Section 32.4), client requests are dispatched to the servant as soon as you call `add`.

16.8.4 UUIDs as Identities

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) [14] as identities. The `Ice.Util` package contains a helper function to create such identities:

```
public class Example
{
    public static void Main(string[] args)
    {
        System.Console.WriteLine(Ice.Util.generateUUID());
    }
}
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `genera-`

`teUUID` creates a string that differs from all previous ones.[3] You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

```
_adapter.addWithUUID(new NodeI("Fred"));
```

16.8.5 Creating Proxies

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Chapter 3. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

```
NodePrx proxy = NodePrxHelper.uncheckedCast(
                    _adapter.addWithUUID(new NodeI("Fred")));
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice.ObjectPrx`.

3. Well, almost: eventually, the UUID algorithm wraps around and produces strings that repeat themselves, but this will not happen until approximately the year 3400.

Direct Proxy Creation

The object adapter offers an operation to create a proxy for a given identity:

```
module Ice {
    local interface ObjectAdapter {
        Object* createProxy(Identity id);
        // ...
    };
};
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

```
Ice.Identity id = new Ice.Identity();
id.name = Ice.Util.generateUUID();
Ice.ObjectPrx o = _adapter.createProxy(id);
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Chapter 34.)

16.9 Asynchronous Method Dispatch (AMD)

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool (see Section 32.10). If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (*AMD*), the server-side equivalent of AMI (see Section 14.16), addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server sends a response explicitly using a callback object provided by the Ice run time.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data (i.e., the callback object and operation arguments) for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results while still in the dispatch thread, and then continues using the dispatch thread for additional work.

16.9.1 Enabling AMD with Metadata

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive *replaces* synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

```
["amd"] interface I {
  bool isValid();
  float computeRate();
};

interface J {
  ["amd"] void startProcess();
  int endProcess();
};
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface or class level) minimizes the amount of generated code and, more importantly, minimizes complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

16.9.2 AMD Mapping

The C# mapping emits the following code for each AMD operation:

1. A callback interface used by the implementation to notify the Ice run time about the completion of an operation. The name of this interface is formed using the pattern `AMD_class_op`. For example, an operation named `foo` defined in interface `I` results in an interface named `AMD_I_foo`. The interface is generated in the same scope as the interface or class containing the operation. Two methods are provided:

   ```
   public void ice_response(<params>);
   ```

   The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-`void` return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's `out` parameters follow the return value, in the order of declaration.

   ```
   public void ice_exception(System.Exception ex);
   ```

   The `ice_exception` method allows the server to raise an exception.

   Neither `ice_response` nor `ice_exception` throw any exceptions to the caller.

2. The dispatch method, whose name has the suffix `_async`. This method has a `void` return type. The first parameter is a reference to an instance of the callback interface described above. The remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
  ["amd"] int foo(short s, out long l);
};
```

The callback interface generated for operation `foo` is shown below:

```
public interface AMD_I_foo
{
    void ice_response(int __ret, long l);
    void ice_exception(System.Exception ex);
}
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

```
public abstract void foo_async(AMD_I_foo __cb, short s,
                               Ice.Current __current);
```

16.9.3 Exceptions

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread that receives the invocation), and the response thread (the thread that sends the response).[4] Although we recommend that the callback object be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the callback object. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

Whether raised in a dispatch thread or reported via the callback object, user exceptions are validated as described in Section 4.10.2, and local exceptions may undergo the translation described in Section 4.10.4.

16.9.4 Example

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

```
module Demo {
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {};

    interface Model {
```

4. These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

```
        ["amd"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    };
};
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo._ModelDisp` and supplies a definition for the `interpolate_async` method that creates a `Job` to hold the callback object and arguments, and adds the `Job` to a queue. The method uses a `lock` statement to guard access to the queue:

```
public class ModelI : Demo.ModelDisp_
{
    public override void interpolate_async(
        Demo.AMD_Model_interpolate cb,
        float[][] data,
        float factor,
        Ice.Current current)
    {
        lock(this)
        {
            _jobs.Add(new Job(cb, data, factor));
        }
    }

    private System.Collections.ArrayList _jobs
                = new System.Collections.ArrayList();
}
```

After queuing the information, the operation returns control to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute`, which uses `interpolateGrid` (not shown) to perform the computational work:

```
public class Job {
    public Job(Demo.AMD_Model_interpolate cb,
               float[][] grid, float factor)
    {
        _cb = cb;
        _grid = grid;
        _factor = factor;
    }
```

```
    public void execute()
    {
        if (!interpolateGrid()) {
            _cb.ice_exception(new Demo.RangeError());
            return;
        }
        _cb.ice_response(_grid);
    }

    private boolean interpolateGrid()
    {
        // ...
    }

    private Demo.AMD_Model_interpolate _cb;
    private float[][] _grid;
    private float _factor;
}
```

If `interpolateGrid` returns `false`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

16.10 Summary

This chapter presented the server-side C# mapping. Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanisms to the client side: a small API to initialize and finalize the run time, plus a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Even though the examples in this chapter are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers (which we discuss in Chapter 32), you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any C# mapping rules beyond those we described here.

Chapter 17
Developing a File System Server in C#

17.1 Chapter Overview

In this chapter, we present the source code for a C# server that implements the file system we developed in Chapter 5 (see Chapter 15 for the corresponding client). The code we present here is fully functional, apart from the required interlocking for threads. (We examine threading issues in detail in Section 32.10.)

17.2 Implementing a File System Server

We have now seen enough of the server-side C# mapping to implement a server for the file system we developed in Chapter 5. (You may find it useful to review the Slice definition for our file system in Section 5.4 before studying the source code.)

Our server is composed of three source files:

- `Server.cs`

  This file contains the server main program.

- `DirectoryI.cs`

  This file contains the implementation for the `Directory` servants.

- `FileI.cs`

  This file contains the implementation for the `File` servants.

17.2.1 The Server `Main` Program

Our server main program, in the file `Server.cs`, uses the `Ice.Application` class we discussed in Section 16.3.1. The `run` method installs a signal handler, creates an object adapter, instantiates a few servants for the directories and files in the file system, and then activates the adapter. This leads to a `Main` method as follows:

```
using Filesystem;
using System;

public class Server
{
    class App : Ice.Application
    {
        public override int run(string[] args)
        {
            // Terminate cleanly on receipt of a signal
            //
            shutdownOnInterrupt();

            // Create an object adapter (stored in the _adapter
            // static members)
            //
            Ice.ObjectAdapter adapter
                = communicator().createObjectAdapterWithEndpoints(
                      "SimpleFilesystem", "default -p 10000");
            DirectoryI._adapter = adapter;
            FileI._adapter = adapter;

            // Create the root directory (with name "/" and no
            // parent)
            //
            DirectoryI root = new DirectoryI("/", null);

            // Create a file called "README" in the root directory
            //
            File file = new FileI("README", root);
            string[] text;
            text = new string[]{ "This file system contains "
                                 + "a collection of poetry." };
```

```
            try {
                file.write(text);
            } catch (GenericError e) {
                Console.Error.WriteLine(e.reason);
            }

            // Create a directory called "Coleridge"
            // in the root directory
            //
            DirectoryI coleridge =
                new DirectoryI("Coleridge", root);

            // Create a file called "Kubla_Khan"
            // in the Coleridge directory
            //
            file = new FileI("Kubla_Khan", coleridge);
            text = new string[]{
                "In Xanadu did Kubla Khan",
                "A stately pleasure-dome decree:",
                "Where Alph, the sacred river, ran",
                "Through caverns measureless to man",
                "Down to a sunless sea." };
            try {
                file.write(text);
            } catch (GenericError e) {
                Console.Error.WriteLine(e.reason);
            }

            // All objects are created, allow client requests now
            //
            adapter.activate();

            // Wait until we are done
            //
            communicator().waitForShutdown();

            if (interrupted())
                Console.Error.WriteLine(
                    appName() + ": terminating");

            return 0;
        }
    }

    public static void Main(string[] args)
    {
```

```
        App app = new App();
        Environment.Exit(app.main(args));
    }
}
```

The code uses a using directive for the `Filesystem` namespace. This avoids having to continuously use fully-qualified identifiers with a `Filesystem.` prefix.

The next part of the source code is the definition of the `Server` class, which includes a nested class that derives from `Ice.Application` and contains the main application logic in its `run` method. Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown in Figure 17.1.

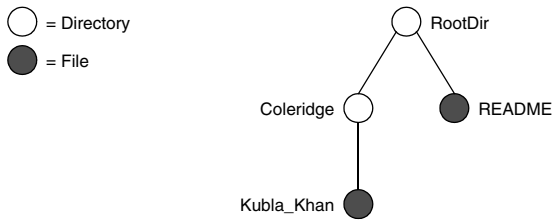


Figure 17.1. A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters, the name of the directory or file to be created and a reference to the servant for the parent directory. (For the root directory, which has no parent, we pass a null parent.) Thus, the statement

```
DirectoryI root = new DirectoryI("/", null);
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in Figure 17.1:

```
// Create the root directory (with name "/" and no parent)
//
DirectoryI root = new DirectoryI("/", null);

// Create a file called "README" in the root directory
//
```

```
File file = new FileI("README", root);
string[] text;
text = new string[]{ "This file system contains "
                     + "a collection of poetry." };
try {
    file.write(text);
} catch (GenericError e) {
    Console.Error.WriteLine(e.reason);
}

// Create a directory called "Coleridge"
// in the root directory
//
DirectoryI coleridge = new DirectoryI("Coleridge", root);

// Create a file called "Kubla_Khan"
// in the Coleridge directory
//
file = new FileI("Kubla_Khan", coleridge);
text = new string[]{ "In Xanadu did Kubla Khan",
                     "A stately pleasure-dome decree:",
                     "Where Alph, the sacred river, ran",
                     "Through caverns measureless to man",
                     "Down to a sunless sea." };
try {
    file.write(text);
} catch (GenericError e) {
    Console.Error.WriteLine(e.reason);
}
```

We first create the root directory and a file `README` within the root directory. (Note that we pass a reference to the root directory as the parent when we create the new node of type `FileI`.)

The next step is to fill the file with text:

```
string[] text;
text = new string[]{ "This file system contains "
                     + "a collection of poetry." };
try {
    file.write(text);
} catch (GenericError e) {
    Console.Error.WriteLine(e.reason);
}
```

Recall from Section 14.7.3 that Slice sequences by default map to C# arrays. The Slice type `Lines` is simply an array of strings; we add a line of text to our `README` file by initializing the `text` array to contain one element.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

```
file.write(text);
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a reference to the servant (of type `FileI`) and *not* via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place—such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary C# method call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in Figure 17.1.

17.2.2 The `FileI` Servant Class

Our `FileI` servant class has the following basic structure:

```
using Filesystem;
using System;

public class FileI : FileDisp_
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private string _name;
    private DirectoryI _parent;
    private string[] _lines;
}
```

The class has a number of data members:

- `_adapter`

  This static member stores a reference to the single object adapter we use in our server.

- `_name`

  This member stores the name of the file incarnated by the servant.

- `_parent`

  This member stores the reference to the servant for the file's parent directory.

- `_lines`

  This member holds the contents of the file.

The `_name` and `_parent` data members are initialized by the constructor:

```
public FileI(string name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    Debug.Assert(_parent != null);

    // Create an identity
    //
    Ice.Identity myID = new Ice.Identity();
    myId.name = Ice.Util.generateUUID();

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);

    // Create a proxy for the new node and
    // add it as a child to the parent
    //
    NodePrx thisNode = NodePrxHelper.uncheckedCast(
                            _adapter.createProxy(myID));
    _parent.addChild(thisNode);
}
```

After initializing the `_name` and `_parent` members, the code verifies that the reference to the parent is not null because every file must have a parent directory. The constructor then generates an identity for the file by calling `Ice.Util.generateUUID` and adds itself to the servant map by calling `ObjectAdapter.add`. Finally, the constructor creates a proxy for this file and calls the `addChild` method on its parent directory. `addChild` is a helper function that a child directory or file calls to add itself to the list of descendant nodes of its parent directory. We will see the implementation of this function on page 557.

The remaining methods of the `FileI` class implement the Slice operations we defined in the `Node` and `File` Slice interfaces:

```
    // Slice Node::name() operation

    public override string name(Ice.Current current)
    {
        return _name;
    }

    // Slice File::read() operation

    public override string[] read(Ice.Current current)
    {
        return _lines;
    }

    // Slice File::write() operation

    public override void write(string[] text, Ice.Current current)
    {
        _lines = text;
    }
```

The `name` method is inherited from the generated `Node` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived). It simply returns the value of the `_name` member.

The `read` and `write` methods are inherited from the generated `File` interface (which is a base interface of the `_FileDisp` class from which `FileI` is derived) and simply return and set the `_lines` member.

17.2.3 The DirectoryI Servant Class

The `DirectoryI` class has the following basic structure:

```
using Filesystem;
using System;
using System.Collections;

public class DirectoryI : DirectoryDisp_
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private string _name;
    private DirectoryI _parent;
    private ArrayList _contents = new ArrayList();
}
```

As for the `FileI` class, we have data members to store the object adapter, the name, and the parent directory. (For the root directory, the `_parent` member holds a null reference.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

```
    public DirectoryI(string name, DirectoryI parent)
    {
        _name = name;
        _parent = parent;

        // Create an identity. The
        // parent has the fixed identity "RootDir"
        //
        Ice.Identity myID = new Ice.Identity();
        myID.name = _parent != null ? Ice.Util.generateUUID() : "R
ootDir";

        // Add the identity to the object adapter
        //
        _adapter.add(this, myID);

        // Create a proxy for the new node and
        // add it as a child to the parent
        //
        NodePrx thisNode = NodePrxHelper.uncheckedCast(
                                _adapter.createProxy(myID));
        if (_parent != null)
            _parent.addChild(thisNode);
    }
```

The constructor creates an identity for the new directory by calling `Ice.Util.generateUUID`. (For the root directory, we use the fixed identity `"RootDir"`.) The servant adds itself to the servant map by calling `ObjectAdapter.add` and then creates a proxy to itself and passes it to the `addChild` helper function.

`addChild` simply adds the passed reference to the `_contents` list:

```
    public void addChild(NodePrx child)
    {
        _contents.Add(child);
    }
```

The remainder of the operations, `name` and `list`, are trivial:

```
public override string name(Ice.Current current)
{
    return _name;
}

public override NodePrx[] list(Ice.Current current)
{
    return (NodePrx[])_contents.ToArray(typeof(NodePrx));
}
```

Note that the `_contents` member is of type `System.Collections.ArrayList`, which is convenient for the implementation of the `addChild` method. However, this requires us to convert the list into a C# array in order to return it from the `list` operation.

17.3 Summary

This chapter showed how to implement a complete server for the file system we defined in Chapter 5. Note that the server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

Note that the server code we presented here is not quite correct as it stands: if two clients access the same file in parallel, each via a different thread, one thread may read the `_lines` data member while another thread updates it. Obviously, if that happens, we may write or return garbage or, worse, crash the server. However, it is trivial to make the `read` and `write` operations thread-safe. We discuss thread safety in Section 32.10.

Part VI

Objective-C Mapping

Chapter 18
Client-Side Slice-to-Objective-C Mapping

18.1 Chapter Overview

In this chapter, we present the client-side Slice-to-Objective-C mapping (see Chapter 20 for the server-side mapping). One part of the client-side Objective-C mapping concerns itself with rules for representing each Slice data type as a corresponding Objective-C type; we cover these rules in Sections 18.3 to 18.11. Another part of the mapping deals with how clients can invoke operations, pass and receive parameters, and handle exceptions. These topics are covered in Sections 18.12 to 18.14. Slice classes have the characteristics of both data types and interfaces and are covered in Sections 18.15 and 18.16. Section 18.17 presents asynchronous invocations, and Section 18.18 discusses how to use the **`slice2objc`** compiler.

18.2 Introduction

The client-side Slice-to-Objective-C mapping defines how Slice data types are translated to Objective-C types, and how clients invoke operations, pass parameters, and handle errors. Much of the Objective-C mapping is intuitive. For example, Slice dictionaries map to Cocoa framework dictionaries, so there is little new you have to learn in order to use Slice dictionaries in Objective-C.

The Objective-C mapping is thread-safe. For example, you can concurrently invoke operations on an object from different threads without the risk of race conditions or corrupting data structures in the Ice run time, but you must still synchronize access to application data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data—the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least Sections 18.11 to 18.13 in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

A word of advice before you start: in order to use the Objective-C mapping, you should need no more than the Slice definition of your application and knowledge of the Objective-C mapping rules. In particular, looking through the generated code in order to discern how to use the Objective-C mapping is likely to be confusing because the generated code is not necessarily meant for human consumption and, occasionally, contains various cryptic constructs to deal with mapping internals. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

18.3 Mapping for Modules

Because Objective-C does not support namespaces, a Slice module maps to a prefix for the identifiers defined inside the modules. By default, the prefix is the same as the name of the module:

```
module example
{
    enum Color { Red, Green, Blue };
};
```

With this definition, the Slice `Color` identifier is mapped to the Objective-C identifier `exampleColor`.

For nested modules, by default, the module identifiers are concatenated. For example, consider the following Slice definition:

```
module outer {
    module inner {
        enum Color { Red, Green, Blue };
    };
};
```

With this definition, the Slice identifier `Color` becomes `outerinnerColor` in Objective-C.

You can use a metadata directive to change the default mapping to a different prefix. For example:

```
["objc:prefix:OUT"]
module outer {
    enum Vehicle { Car, Truck, Bicycle };

    module inner {
        enum Color { Red, Green, Blue };
    };
};
```

With this definition, `Vehicle` maps to `OUTVehicle`. However, `Color` still maps to `outerinnerColor`, that is, the metadata directive applies only to types defined in the `outer` module, but not to types that are defined in nested modules. If you want to assign a prefix for types in the nested module, you must use a separate metadata directive, for example:

```
["objc:prefix:OUT"]
module outer {
    enum Vehicle { Car, Truck, Bicycle };

    ["objc:prefix:IN"]
    module inner {
        enum Color { Red, Green, Blue };
    };
};
```

With this definition, `Vehicle` maps to `OUTVehicle`, and `Color` maps to `INColor`.

For the remainder of the examples in this chapter, we assume that Slice definitions are enclosed by a module `Example` with a `["objc:prefix:EX"]` metadata directive.

18.4 The ICE Prefix

All of the APIs for the Ice run time are prefixed by ICE, to avoid clashes with definitions for other libraries or applications. Parts of the Ice API are generated from Slice definitions; other parts provide special-purpose definitions that do not have a corresponding Slice definition. Regardless of they way they are defined, the ICE prefix universally applies to all entry points in the Ice run time. We will incrementally cover the contents of the Ice API throughout the remainder of the book.

18.5 Mapping for Identifiers

Objective-C identifiers are derived from Slice identifiers. The exact Objective-C identifier that is generated depends on the context. For types that are nested in modules (and hence have global visibility in Objective-C), the generated Objective-C identifiers are prefixed with their module prefix, as discussed in Section 18.3. Slice identifiers that do not have global visibility (such as operation names and structure members) do not use the module prefix and are preserved without change. For example, consider the following Slice definition:

```
["objc:prefix:EX"]
module Example {
    struct Point {
        double x;
        double y;
    };
};
```

This maps to the following Objective-C definition:

```
@interface EXPoint : NSObject <NSCopying>
{
    @private
        ICEDouble x;
        ICEDouble y;
}

@property(nonatomic, assign) ICEDouble x;
```

```
@property(nonatomic, assign) ICEDouble y;

// More definitions here...
@end
```

If a Slice identifier is the same as an Objective-C keyword, the corresponding Objective-C identifier has an underscore suffix. For example, Slice `while` maps to Objective-C `while_`.

In some cases, the Objective-C mapping generates more than one identifier for a given Slice construct. For example, an interface `Intf` generates the identifiers `EXIntf` and `EXIntfPrx`. If a Slice identifier happens to be an Objective-C keyword, the underscore suffix applies only where necessary, so an interface while generates `EXWhile` and `EXWhilePrx`.

Note that Slice operation and member names can clash with the name of an inherited method, property, or instance variable. For example:

```
exception Failed {
    string reason; // Clashes with NSException
};
```

This is a legal Slice definition. However, the generated exception class derives from `NSException`, which defines a `reason` method. To avoid hiding the method in the base class, the generated exception class maps the Slice `reason` member to the Objective-C property `reason_`, just as it would for a keyword.

This escape mechanism applies to all generated classes that directly or indirectly derive from `NSObject` or `NSException`.

18.6 Internal Identifiers

Any methods that contain two or more adjacent underscores (such as `read__` and `op____`) are internal to the Objective-C mapping implementation and are not for use by application code.

18.7 Mapping for Built-In Types

The Slice built-in types are mapped to Objective-C types as shown in Table 18.1.

Table 18.1. Mapping of Slice built-in types to Objective-C.

Slice	Objective-C
`bool`	`BOOL`
`byte`	`ICEByte`
`short`	`ICEShort`
`int`	`ICEInt`
`long`	`ICELong`
`float`	`ICEFloat`
`double`	`ICEDouble`
`string`	`NSString` or `NSMutableString`

Slice `bool` maps to Objective-C `BOOL`. The remaining integral and floating-point types map to Objective-C type definitions instead of native types. This allows the Ice run time to provide a definition as appropriate for each target architecture. (For example, `ICELong` might be defined as `long` on one architecture and as `long long` on another.)

Note that `ICEByte` is a typedef for `unsigned char`. This guarantees that byte values are always in the range 0..255, and it ensures that right-shifting an `ICEByte` does not cause sign-extension.

Whether a Slice string maps to `NSString` or `NSMutableString` depends on the context. `NSMutableString` is used in some cases for operation parameters; otherwise, if a string is a data member of a Slice structure, class, or exception, it maps to `NSString`. (We will discuss these differences in more detail as we cover the mapping of the relevant Slice language features.)

18.8 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

18.8.1 Mapping for Enumerations

Enumerations map to the corresponding enumeration in Objective-C. For example:

```
["objc:prefix:EX"]
module Example {
    enum Fruit { Apple, Pear, Orange };
};
```

The generated Objective-C definition is:

```
typedef enum {
    EXApple, EXPear, EXOrange
} EXFruit;
```

18.8.2 Mapping for Structures

The mapping for structures maps Slice structures to Objective-C classes. For each Slice data member, the Objective-C class has a corresponding property. For example, here is our `Employee` structure from Section 4.9.4 once more:

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The Slice-to-Objective-C compiler generates the following definition for this structure:

```
@interface EXEmployee : NSObject <NSCopying>
{
    @private
        ICELong number;
        NSString *firstName;
        NSString *lastName;
}
```

```
@property(nonatomic, assign) ICELong number;
@property(nonatomic, retain) NSString *firstName;
@property(nonatomic, retain) NSString *lastName;

-(id) init:(ICELong)number firstName:(NSString *)firstName
                           lastName:(NSString *)lastName;
+(id) employee:(ICELong)number firstName:(NSString *)firstName
                               lastName:(NSString *)lastName;
+(id) employee;
// This class also overrides copyWithZone,
// hash, isequal, and dealloc.
@end
```

Mapping for Data Members

For each data member in the Slice definition, the Objective-C class contains a corresponding private instance variable of the same name, as well as a property definition that allows you to set and get the value of the corresponding instance variable. For example, given an instance of `EXEmployee`, you can write the following:

```
ICELong number;
EXemployee *e = ...;
[e setNumber:99];
number = [e number];

// Or, more concisely with dot notation:

e.number = 99;
number = e.number;
```

Properties that represent data members always use the `nonatomic` property attribute. This avoids the overhead of locking each data member during access. The second property attribute is `assign` for integral and floating-point types and `retain` for all other types (such as strings, structures, and so on.)

Creating and Initializing Structures

Structures provide the usual (inherited) `init` method:

```
EXEmployee *e = [[EXEmployee alloc] init];
// ...
[e release];
```

As usual, `init` initializes the instance variables of the structure with zero-filled memory.

In addition, a structure provides a second `init` method that accepts one parameter for each data member of the structure:

```
-(id) init:(ICELong)number firstName:(NSString *)firstName
                          lastName:(NSString *)lastName;
```

Note that the first parameter is always unlabeled; the second and subsequent parameters have a label that is the same as the name of the corresponding Slice data member. The additional `init` method allows you to instantiate a structure and initialize its data members in a single statement:

```
EXEmployee *e = [[EXEmployee alloc] init:99 firstName:@"Brad"
                                            lastName:@"Cox"];
// ...
[e release];
```

`init` applies the memory management policy of the corresponding properties, that is, it calls `retain` on the `firstName` and `lastName` arguments.

Each structure also provides two convenience constructors that mirror the `init` methods: a parameter-less convenience constructor and one that has a parameter for each Slice data member:

```
+(id) employee;
+(id) employee:(ICELong)number firstName:(NSString *)firstName
                               lastName:(NSString *)lastName;
```

The convenience constructors have the same name as the mapped Slice structure (without the module prefix). As usual, they allocate an instance, perform the same initialization actions as the corresponding `init` methods, and call `autorelease` on the return value:

```
EXEmployee *e = [EXEmployee employee:99 firstName:@"Brad"
                                        lastName:@"Cox"];

// No need to call [e release] here.
```

Copying Structures

Structures implement the `NSCopying` protocol. Structures are copied by assigning instance variables of value type and calling `retain` on each instance variable of non-value type. In other words, the copy is shallow:

```
EXEmployee *e = [EXEmployee employee:99 firstName:@"Brad"
                                        lastName:@"Cox"];
EXEmployee *e2 = [e copy];
NSAssert(e.number == e2.number);
NSAssert([e.firstName == e2.firstName]); // Same instance
// ...
[e2 release];
```

Note that, if you assign an `NSMutableString` to a structure member and use the structure as a dictionary key, you must not modify the string inside the structure without copying it because doing so will corrupt the dictionary.

Deallocating Structures

Each structure implements a `dealloc` method that calls `release` on each instance variable with a `retain` property attribute. This means that structures take care of the memory management of their contents: releasing a structure automatically releases all its instance variables.

Structure Comparison and Hashing

Structures implement `isEqual`, so you can compare them for equality. Two structures are equal if all their instance variables are equal. For value types, equality is determined by the `==` operator; for non-value types other than classes, equality is determined by the corresponding instance variable's `isEqual` method. Classes (see Section 18.15) are compared by comparing their identity: two class members are equal if they both point at the same instance.

The `hash` method returns a hash value is that is computed from the hash value of all of the structure's instance variables.

18.8.3 Mapping for Sequences

The Objective-C mapping uses different mappings for sequence of value types (such as `sequence<byte>`) and non-value types (such as `sequence<string>`).

Mapping for Sequences of Value Types

The following Slice types are value types:

- Integral types (`bool`, `byte`, `short`, `int`, `long`)
- Floating point types (`float`, `double`)
- Enumerated types

Sequences of these types map to a type definition. For example:

```
enum Fruit { Apple, Pear, Orange };

sequence<byte> ByteSeq;
sequence<int> IntSeq;
sequence<Fruit> FruitSeq;
```

The three Slice sequences result in the following Objective-C definitions:

```
typedef enum {
    EXApple, EXPear, EXOrange
} EXFruit;

typedef NSData EXByteSeq;
typedef NSMutableData EXMutableByteSeq;

typedef NSData EXIntSeq;
typedef NSMutableData EXMutableIntSeq;

typedef NSData EXFruitSeq;
typedef NSMutableData EXMutableFruitSeq;
```

As you can see, each sequence definition creates a pair of type definitions, an immutable version named *<module-prefix><Slice-name>*, and a mutable version named *<module-prefix>*`Mutable`*<Slice-name>*. This constitutes the entire public API for sequence of value types, that is, sequences of value types simply map to `NSData` or `NSMutableData`. The `NS(Mutable)Data` sequences contain an array of the corresponding element type in their internal byte array.[1]

For example, here is how you could initialize a byte sequence of 1024 elements with values that are the modulo 128 of the element index in reverse order:

```
int limit = 1024;
EXMutableByteSeq *bs = [NSMutableData dataWithLength:limit];
ICEByte *p = (ICEByte *)[bs bytes];
while (--limit > 0) {
    *p++ = limit % 0x80;
}
```

1. We chose to map sequences of value types to `NSData` instead of `NSArray` because of the large overhead of placing each sequence element into an `NSNumber` container instance.

Naturally, you do not need to initialize the sequence using a loop. For example, if the data is available in a buffer, you could use the `dataWithBytes:length` or `dataWithBytesNoCopy:length` methods of `NSData` instead.

Here is one way to retrieve the bytes of the sequence:

```
const ICEByte* p = (const ICEByte *)[bs bytes];
const ICEByte* limitp = p + [bs length];
while (p < limitp) {
    printf("%d\n", *p++);
}
```

For sequences of types other than `byte` or `bool`, you must keep in mind that the length of the `NSData` array is not the same as the number of elements. The following example initializes an integer sequence with the first few primes and prints out the contents of the sequence:

```
const int primes[] = { 1, 2, 3, 5, 7, 9, 11, 13, 17, 19, 23 };
EXMutableIntSeq *is = [NSMutableData dataWithBytes:primes
                                     length:sizeof(primes)];

const ICEInt *p = (const ICEInt *)[is bytes];
int limit = [is length] / sizeof(*p);
int i;
for(i = 0; i < limit; ++i) {
    printf("%d\n", p[i]);
}
```

The code to manipulate a sequence of enumerators is very similar. For portability, you should not assume a particular size for enumerators. That is, instead of relying on all enumerators having the size of, for example, an `int`, it is better to use `sizeof(EXFruit)` to ensure that you are not overstepping the bounds of the sequence.

Mapping of Sequences of Non-Value Types

Sequences of non-value types, such as sequences of `string`, structures, classes, and so on, map to mutable and immutable type definitions of `NSArray`. For example:

```
sequence<string> Page;
sequence<Page> Book;
```

This maps to:

```
typedef NSArray EXPage;
typedef NSMutableArray EXMutablePage;

typedef NSArray EXBook;
typedef NSMutableArray EXMutableBook;
```

You use such sequences as you would use any other `NSArray` in your code. For example:

```
EXMutablePage *page1 = [NSArray arrayWithObjects:
                                @"First line of page one",
                                @"Second line of page one",
                                nil];

EXMutablePage *page2 = [NSArray arrayWithObjects:
                                @"First line of page two",
                                @"Second line of page two",
                                nil];

EXMutableBook *book = [NSMutableArray array];
[book addObject:page1];
[book addObject:page2];
[book addObject:[NSArray array]]; // Empty page
```

This creates a book with three pages; the first two pages contain two lines each, and the third page is empty. You can print the contents of the book as follows:

```
int pageNum = 0;
for (EXPage *page in book) {
    ++pageNum;
    int lineNum = 0;
    if ([page count] == 0) {
        printf("page %d: <empty>\n", pageNum);
    } else {
        for (NSString *line in page) {
            ++lineNum;
            printf("page %d, line %d: %s\n",
                        pageNum, lineNum, [line UTF8String]);
        }
    }
}
```

This prints:

```
page 1, line 1: First line of page one
page 1, line 2: Second line of page one
page 2, line 1: First line of page two
page 2, line 2: Second line of page two
page 3: <empty>
```

If you have a sequence of proxies or a sequence of classes, to transmit a null proxy or class inside a sequence, you must insert an `NSNull` value into the `NSArray`. In addition, the mapping also allows you to use `NSNull` as the element value of an `NSArray` for elements of type string, structure, sequence, or dictionary. For example, instead of inserting an empty `NSArray` into the book sequence in the preceding example, we could also have inserted `NSNull`:

```
EXMutableBook *book = [NSMutableArray array];
[book addObject:page1];
[book addObject:page2];
[book addObject:[NSNull null]]; // Empty page
```

18.8.4 Mapping for Dictionaries

Here is the definition of our `EmployeeMap` from Section 4.9.4 once more:

```
dictionary<long, Employee> EmployeeMap;
```

The following code is generated for this definition:

```
typedef NSDictionary EXEmployeeMap;
typedef NSMutableDictionary EXMutableEmployeeMap;
```

Similar to sequences, Slice dictionaries map to type definitions for `NSDictionary` and `NSMutableDictionary`, with the names *<module-prefix><Slice-name>* and *<module-prefix>*`Mutable`*<Slice-name>*.

As a result, you can use the dictionary like any other `NSDictionary`, for example:

```
EXMutableEmployeeMap *em = [EXMutableEmployeeMap dictionary];
EXEmployee *e = [EXEmployee employee];
e.number = 42;
e.firstName = @"Stan";
e.lastName = @"Lippman";
[em setObject:e forKey:[NSNumber numberWithLong:e.number]];

e = [EXEmployee employee];
```

```
e.number = 77;
e.firstName = @"Herb";
e.lastName = @"Sutter";
[em setObject:e forKey:[NSNumber numberWithLong:e.number]];
```

To put a value type into a dictionary (either as the key or the value), you must use `NSNumber` as the object to hold the value. If you have a dictionary that uses a Slice enumeration as the key or the value, you must insert the enumerator as an `NSNumber` that holds an `int`.

To insert a null proxy or null class instance into a dictionary as a value, you must insert `NSNull`.

As a convenience feature, the Objective-C mapping also allows you to insert `NSNull` as the value of a dictionary if the value type of the dictionary is a string, structure, sequence, or dictionary. If you send such a dictionary to a receiver, the Ice run time marshals an empty string, default-initialized structure, empty sequence, or empty dictionary as the corresponding value to the receiver, respectively.

18.9 Mapping for Constants

Slice constant definitions map to corresponding Objective-C constant definitions. Here are the constant definitions we saw in Section 4.9.5 on page 103 once more:

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

Here are the generated definitions for these constants:

```
static const BOOL EXAppendByDefault = YES;
static const ICEByte EXLowerNibble = 15;
static NSString * const EXAdvice = @"Don't Panic!";
static const ICEShort EXTheAnswer = 42;
static const ICEDouble EXPI = 3.1416;
```

```
typedef enum {
    EXApple, EXPear, EXOrange
} EXFruit;
static const EXFruit EXFavoriteFruit = EXPear;
```

All constants are initialized in the generated header file, so they are compile-time constants and can be used in contexts where a compile-time constant expression is required, such as to dimension an array or as the `case` label of a `switch` statement.

18.10 Mapping for Exceptions

Here is a fragment of the Slice definition for our world time server from Section 4.10.5 on page 120 once more:

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

18.10.1 Inheritance Hierarchy

These exception definitions map as follows:

```
@interface EXGenericError : ICEUserException
{
@private
    NSString *reason_;
}

@property(nonatomic, retain) NSString *reason_;

// ...
@end

@interface EXBadTimeVal : EXGenericError
// ...
@end

@interface EXBadZoneName : EXGenericError
// ...
@end
```

Each Slice exception is mapped to an Objective-C class. For each exception member, the corresponding class contains a private instance variable and a property. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.)

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `EXBadTimeVal` and `EXBadZoneName` inherit from `EXGenericError`.

In turn, `EXGenericError` derives from `ICEUserException`:

```
@interface ICEException : NSException
-(NSString* *)ice_name;
@end

@interface ICEUserException : ICEException
// ...
@end

@interface ICELocalException : ICEException
// ...
@end
```

Note that `ICEUserException` itself derives from `ICEException`, which derives from `NSException`. Similarly, run-time exceptions derive from a

common base class `ICELocalException` that derives from `ICEException`, so we have the inheritance structure shown in Figure 18.1:

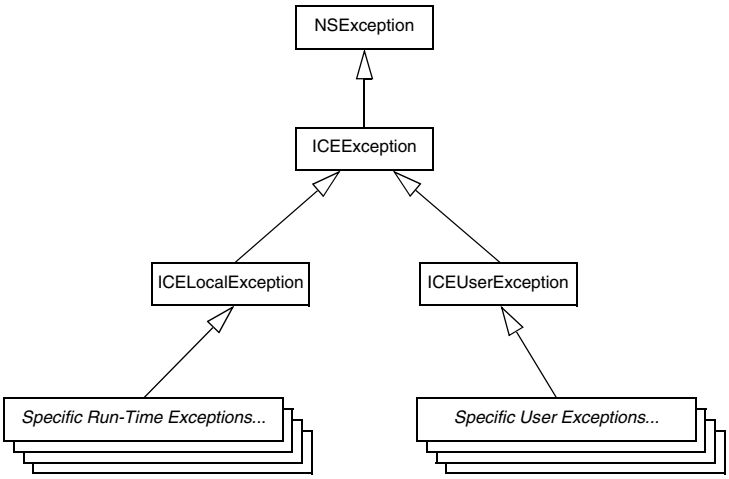


Figure 18.1. Inheritance structure for exceptions.

`ICEException` provides a single method, `ice_name`, that returns the Slice type ID of the exception with the leading `::` omitted. For example, the return value of `ice_name` for our Slice `GenericError` is `Example::GenericError`.

18.10.2 Mapping for Data Members

As we mentioned on page 577, each data member of a Slice exception generates a corresponding Objective-C property. Here is an example that extends our `GenericError` with yet another exception:

```
exception GenericError {
    string reason;
};

exception FileError extends GenericError {
    string name;
    int errorCode;
};
```

The generated properties for these exceptions are as follows:

```
@interface EXGenericError : ICEUserException
{
@private
    NSString *reason_;
}

@property(nonatomic, retain) NSString *reason_;

// ...
@end

@interface EXFileError : EXGenericError
{
@private
    NSString *name_;
    ICEInt errorCode;
}

@property(nonatomic, retain) NSString *name_;
@property(nonatomic, assign) ICEInt errorCode;

// ...
@end
```

This is exactly the same mapping as for structure members (see page 567), with one difference: the `name` and `reason` members map to `name_` and `reason_` properties, whereas—as for structures—`errorCode` maps to `errorCode`. The trailing underscore for `reason_` and `name_` prevents a name collision with the `name` and `reason` methods that are defined by `NSException`: if you call the `name` method, you receive the name that is stored by `NSException`; if you call the `name_` method, you receive the value of the `name_` instance variable of `EXFileError`:

```
@try {
    // Do something that can throw ExFileError...
}
@catch(EXFileError *ex)
{
    // Print the value of the Slice reason, name,
    // and errorCode members.
    printf("reason: %s, name: %s, errorCode: %d\n",
            [ex.reason_ UTF8String],
            [ex.name_ UTF8String],
            ex.errorCode);
```

```
    // Print the NSException name.
    printf("NSException name: %s\n", [[ex name] UTF8String]);
}
```

The same escape mechanism applies if you define exception data members named `callStackReturnAddresses`, `raise`, or `userInfo`.

18.10.3 Mapping for User Exceptions

Creating and Initializing User Exceptions

Initialization of exceptions follows the same pattern as for structures (see page 568): each exception (apart from the inherited no-argument `init` method) provides an `init` method that accepts one argument for each data member of the exception, and two convenience constructors. For example, the generated methods for our `EXGenericError` exception look as follows:

```
@interface EXGenericError : ICEUserException
// ...

-(id) init:(NSString *)reason;
+(id) genericError;
+(id) genericError:(NSString *)reason;
@endif
```

If a user exception has no data members (and its base exceptions do not have data members either), only the inherited `init` method and the no-argument convenience constructor are generated.

If an exception has a base exception with data members, its `init` method and convenience constructor accept one argument for each Slice data member, in base-to-derived order. For example, here are the methods for the `FileError` exception we defined on page 578):

```
@interface EXFileError : EXGenericError
// ...

-(id) init:(NSString *)reason name_:(NSString *)name
                              errorCode:(ICEInt)errorCode;
+(id) fileError;
+(id) fileError:(NSString *)reason name_:(NSString *)name
                                   errorCode:(ICEInt)errorCode;
@end
```

Note that `init` and the second convenience constructor accept three arguments; the first initializes the `EXGenericError` base, and the remaining two initialize the instance variables of `EXFileError`.

18.10.4 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `ICELocalException` which, in turn, derives from `ICEException`. (See Figure 4.4 for an inheritance diagram.)

By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `NSException`

  This is the root of the complete inheritance tree. Catching `NSException` catches all exceptions, whether they relate to Ice or the Cocoa framework.

- `ICEException`

  Catching `ICEException` catches both user and run-time exceptions.

- `ICEUserException`

  This is the root exception for all user exceptions. Catching `ICEUserException` catches all user exceptions (but not run-time exceptions).

- `ICELocalException`

  This is the root exception for all run-time exceptions. Catching `ICELocalException` catches all run-time exceptions (but not user exceptions).

- `ICETimeoutException`

  This is the base exception for both operation-invocation and connection-establishment timeouts.

- `ICEConnectTimeoutException`

  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, an `ICEConnectTimeoutException` can be handled as `ICEConnectTimeoutException`, `ICETimeoutException`, `ICELocalException`, `ICEException`, or `NSException`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `ICELocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in

the implementation of the Ice run time. Exceptions to this rule are `ICEFacetNotExistException` (see Chapter 33) and `ICEObjectNotExistException` (see Chapter 34), which you may want to catch explicitly.

Creating and Initializing Run-Time Exceptions

`ICELocalException` provides two properties that return the file name and line number at which an exception was raised:

```
@interface ICELocalException : ICEException
{
// ...

@property(nonatomic, readonly) NSString* file;
@property(nonatomic, readonly) int line;

-(id)init:(const char*)file line:(int)line;
+(id)localException:(const char*)file line:(int)line;
@end
```

The `init` method and the convenience constructor accept the file name and line number as arguments.

Concrete run-time exceptions that derived from `ICEException` provide a corresponding `init` method and convenience constructor. For example, here is the Slice definition of `ObjectNotExistException`:

```
local exception RequestFailedException {
    Identity id;
    string facet;
    string operation;
};

local exception ObjectNotExistException
    extends RequestFailedException {};
```

The Objective-C mapping for `ObjectNotExistException` is:

```
@interface ICEObjectNotExistException : ICERequestFailedException
// ...
-(id) init:(const char*)file__p line:(int)line__p;
-(id) init:(const char*)file__p
           line:(int)line__p
           id_:(ICEIdentity *)id_
           facet:(NSString *)facet
           operation:(NSString *)operation;
+(id) objectNotExistException:(const char*)file__p
```

```
                                line:(int)line__p;
+(id) objectNotExistException:(const char*)file__p
                                line:(int)line__p
                                id_:(ICEIdentity *)id_
                                facet:(NSString *)facet
                                operation:(NSString *)operation;
@end
```

In other words, as for user exceptions, run-time exceptions provide `init` methods and convenience constructors that accept arguments in base-to-derived order. This means that all run-time exceptions require a file name and line number when they are instantiated. For example, you can throw an `ICEObjectNotExistException` as follows:

```
@throw [ICEObjectNotExistException
          objectNotExistException:__FILE__ line:__LINE__];
```

If you throw this exception in the context of an executing operation on the server side, the `id_`, `facet`, and `operation` instance variables are automatically initialized by the Ice run time.

When you instantiate a run-time exception, the base `NSException` is initialized such that its `name` method returns the same string as `ice_name`; the `reason` and `userInfo` methods return `nil`.

18.10.5 Copying and Deallocation

User exceptions and run-time exceptions implement the `NSCopying` protocol, so you can copy them. The semantics are the same as for structures (see page 569).

Similarly, like structures, exceptions implement a `dealloc` method that takes care of deallocating the instance variables when an exception is released.

18.10.6 Comparison and Hashing

Exceptions do not override `isEqual` or `hash`, so these methods have the behavior inherited from `NSObject`.

18.11 Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that represents the

remote object. This makes the mapping easy and intuitive to use because, for all intents and purposes (apart from error semantics), making a remote procedure call is no different from making a local procedure call.

18.11.1 Proxy Classes and Proxy Protocols

On the client side, interfaces map to a protocol with member functions that correspond to the operations on those interfaces. Consider the following simple interface:

```
["objc:prefix:EX"]
module Example {
    interface Simple {
        void op();
    }
};
```

The Slice compiler generates the following definitions for use by the client:

```
@interface EXSimplePrx : ICEObjectPrx
// Mapping-internal methods here...
@end

@protocol EXSimplePrx <ICEObjectPrx>
-(void) op;
-(void) op:(ICEContext *)context;
@end;
```

As you can see, the compiler generates a proxy protocol `EXSimplePrx` and a proxy class `EXSimplePrx`. In general, the generated name for both protocol and class is *`<module-prefix><interface-name>`*`Prx`.

In the client's address space, an instance of `EXSimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy class instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `EXSimplePrx` derives from `ICEObjectPrx`, and that `EXSimplePrx` adopts the `ICEObjectPrx` protocol. This reflects the fact that all Slice interfaces implicitly derive from `Ice::Object`. For each operation in the interface, the proxy protocol has two methods whose name is derived from the operation. For the preceding example, we find that the operation `op` is mapped to two methods, `op` and `op:`.

The second method has a trailing parameter of type `ICEContext`. This parameter is for use by the Ice run time to store information about how to deliver a request; normally, you do not need to supply a value here and can pretend that the trailing parameter does not exist. (We examine the `ICEContext` parameter in detail in Chapter 32. The parameter is also used by IceStorm—see Chapter 44.)

18.11.2 Instantiation and Casting

Client-side application code never manipulates proxy class instances directly. In fact, you are not allowed to instantiate a proxy class directly. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

Proxies are immutable: once the run time has instantiated a proxy, that proxy continues to denote the same remote object and cannot be changed. This means that, if you want to keep a copy of a proxy, it is sufficient to call `retain` on the proxy. (You can also call `copy` on a proxy because `ICEObjectPrx` implements `NSCopying`. However, calling `copy` has the same effect as calling `retain`.)

Proxies are always passed and returned as type `id<`*`<module-prefix><interface-name>`*`Prx>`. For example, for the preceding `Simple` interface, the proxy type is `id<EXSimplePrx>`.

The `ICEObjectPrx` base class provides two class methods that allow you cast a proxy from one type to another.

- `+(id) checkedCast:(id<ICEObjectPrx>)proxy;`

  A `checkedCast` tests whether the object denoted by a proxy implements the specified interface. If so, the cast returns a proxy to the specified interface; otherwise, if the object denoted by the proxy does not implement the specified interface, the cast returns `nil`. Checked casts are typically used to safely down-cast a proxy to a more derived interface. For example, assuming we have Slice interfaces `Base` and `Derived`, you can write the following:

```
id<EXBasePrx> base = ...;  // Initialize base proxy
id<EXDerivedPrx> derived = [EXDerivedPrx checkedCast:base];
if(derived != nil)
{
    // base implements run-time type Derived
    // use derived...
} else {
    // Base has some other, unrelated type
```

```
}
```

The expression `[EXDerivedPrx checkedCast:base]` tests whether `base` points at an object of type `Derived` (or an object with a type that is derived from `Derived`). If so, the cast succeeds and `derived` is set to point at the same object as `base`. Otherwise, the cast fails and `derived` is set to `nil`. (Proxies that "point nowhere" are represented by `nil`.)

A `checkedCast` typically results in a remote message to the server.[2] The message effectively asks the server "is the object denoted by this proxy of type `Derived`?" The reply from the server is communicated to the application code in form of a successful (non-`nil`) or unsuccessful (`nil`) result. Sending a remote message is necessary because, as a rule, there is no way for the client to find out what the actual run-time type of a proxy is without confirmation from the server. (For example, the server may replace the implementation of the object for an existing proxy with a more derived one.) This means that you have to be prepared for a `checkedCast` to fail. For example, if the server is not running, you will receive an `ICEConnectionRefusedException`; if the server is running, but the object denoted by the proxy no longer exists, you will receive an `ICEObjectNotExistException`.

- `+(id) uncheckedCast:(id<ICEObjectPrx>)proxy;`

  In some cases, it is known that an object supports a more derived interface than the static type of its proxy. For such cases, you can use an unchecked down-cast:

  ```
  id<EXBasePrx> base;
  base = ...;  // Initialize base to point at a Derived
  id<EXDerivedPrx> derived = [EXDerivedPrx uncheckedCast:base];
  // Use derived...
  ```

  An `uncheckedCast` provides a down-cast *without* consulting the server as to the actual run-time type of the object. You should use an `unchecked-Cast` only if you are certain that the proxy indeed supports the more derived type: an `uncheckedCast`, as the name implies, is not checked in any way; it does not contact the object in the server and, if the proxy does not support the specified interface, the cast does not return null. If you use the proxy

2. In some cases, the Ice run time can optimize the cast and avoid sending a message. However, the optimization applies only in narrowly-defined circumstances, so you cannot rely on a `checkedCast` not sending a message.

resulting from an incorrect `uncheckedCast` to invoke an operation, the behavior is undefined. Most likely, you will receive an `ICEOperationNotExistException`, but, depending on the circumstances, the Ice run time may also report an exception indicating that unmarshaling has failed, or even silently return garbage results.

Despite its dangers, `uncheckedCast` is still useful because it avoids the cost of sending a message to the server. And, particularly during initialization (see page 641), it is common to receive a proxy of type `id<ICEObjectPrx>`, but with a known run-time type. In such cases, an `uncheckedCast` saves the overhead of sending a remote message.

Note that an `uncheckedCast` is *not* the same as an ordinary cast. The following is incorrect and has undefined behavior:

```
id<EXDerivedPrx> derived = (id<EXDerivedPrx>)base; // Wrong!
```

Both `checkedCast` and `uncheckedCast` call `autorelease` on the proxy they return so, if you want to prevent the proxy from being deallocated once the enclosing autorelease pool is drained, you must call `retain` on the returned proxy.

18.11.3 Using Proxy Methods

The `ICEObjectPrx` provides a variety of methods for customizing a proxy (see Section 32.11). Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten-second timeout as shown below:

```
id<ICEObjectPrx> proxy = [communicator stringToProxy:...];
proxy = [proxy ice_timeout:10000];
```

A factory method returns a new (autoreleased) proxy object if the requested modification differs from the current proxy, otherwise it returns the original proxy. The returned proxy is always of the same type as the source proxy.

18.11.4 Object Identity and Proxy Comparison

Apart from the methods discussed in Section 18.11.2, proxy handles also support comparison with `isEqual`. Note that `isEqual` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the

protocol and endpoint information, must be the same as well. In other words, comparison with `isEqual` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

```
id<ICEObjectPrx> p1 = ...;      // Get a proxy...
id<ICEObjectPrx> p2 = ...;      // Get another proxy...

if (![p1 isEqual:p2]) {
    // p1 and p2 denote different objects       // WRONG!
} else {
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen if, for example, `p1` and `p2` embed the same object identity, but use a different protocol to contact the target object. Similarly, the protocols might be the same, but could denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `isEqual`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `isEqual`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use additional methods provided by proxies:

```
@protocol ICEObjectPrx <NSObject, NSCopying>
// ...
-(NSComparisonResult) compareIdentity:(id<ICEObjectPrx>)p;
-(NSComparisonResult) compareIdentityAndFacet:(id<ICEObjectPrx>)p;
@end
```

The `compareIdentity` method compares the object identities embedded in two proxies while ignoring other information, such as facet and transport information. To include the facet name (see Chapter 33) in the comparison, use `compareIdentityAndFacet` instead.

`compareIdentity` and `compareIdentityAndFacet` allow you to correctly compare proxies for object identity. The example below demonstrates how to use `compareIdentity`:

```
id<ICEObjectPrx> p1 = ...;      // Get a proxy...
id<ICEObjectPrx> p2 = ...;      // Get another proxy...

if ([p1 compareIdentity:p2] != NSOrderedSame) {
```

```
    // p1 and p2 denote different objects       // Correct
} else {
    // p1 and p2 denote the same object         // Correct
}
```

18.12 Mapping for Operations

As we saw in Section 18.11, for each operation on an interface, the proxy protocol contains two corresponding methods with the same name as the operation. To invoke an operation, you call it via the proxy handle. For example, here is part of the definitions for our file system from Section 5.4:

```
["objc:prefix:FS"]
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The proxy protocol for the `Node` interface looks as follows:

```
@protocol FSNodePrx <ICEObjectPrx>
-(NSMutableString *) name;
-(NSMutableString *) name:(ICEContext *)context;
@end;
```

The `name` method returns a value of type `NSMutableString`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

```
id<EXNodePrx> node = ...;       // Initialize proxy
NSString *name = [node name];   // Get name via RPC
```

The `name` method sends the operation invocation to the server, waits until the operation is complete, and then unmarshals the return value and returns it to the caller.

Because the name method autoreleases the return value, it is safe to ignore the return value. For example, the following code contains no memory leak:

```
id<EXNodePrx> node = ...;       // Initialize proxy
[node name];                    // Useless, but no leak
```

If you ignore the return value, no memory leak occurs because the next time the enclosing autorelease pool is drained, the memory will be reclaimed.

18.12.1 Normal and `idempotent` Operations

You can add an `idempotent` qualifier to a Slice operation. As far as the corresponding proxy protocol methods are concerned, `idempotent` has no effect. For example, consider the following interface:

```
interface Example {
    string op1();
    idempotent  string op2();
    idempotent  void op3(string s);
};
```

The proxy protocol for this interface looks like this:[3]

```
@protocol EXOpsPrx <ICEObjectPrx>
-(NSMutableString *) op1;
-(NSMutableString *) op1:(ICEContext *)context;
-(NSMutableString *) op2;
-(NSMutableString *) op2:(ICEContext *)context;
-(void) op3:(NSString *)s;
-(void) op3:(NSString *)s context:(ICEContext *)context;
@end;
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the mapping to be unaffected by the `idempotent` keyword.

18.12.2 Passing Parameters

In-Parameters

The parameter passing rules for the Objective-C mapping are very simple: value type parameters are passed by value and non-value type parameters are passed by pointer. Semantically, the two ways of passing parameters are identical: the Ice run time guarantees not to change the value of an in-parameter.

Here is an interface with operations that pass parameters of various types from client to server:

3. For brevity, we will not show the methods with the additional trailing `context` parameter for the remainder of this chapter. Of course, the compiler generates the additional methods regardless. See Section 32.12 for more information on contexts.

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following code for this definition:

```
@interface EXNumberAndString : NSObject <NSCopying>
// ...
@property(nonatomic, assign) ICEInt x;
@property(nonatomic, retain) NSString *str;
// ...
@end

typedef NSArray EXStringSeq;
typedef NSMutableArray EXMutableStringSeq;

typedef NSDictionary EXStringTable;
typedef NSMutableDictionary EXMutableStringTable;

@protocol EXClientToServerPrx <ICEObjectPrx>
-(void) op1:(ICEInt)i f:(ICEFloat)f b:(BOOL)b s:(NSString *)s;
-(void) op2:(EXNumberAndString *)ns ss:(EXStringSeq *)ss
            st:(NSDictionary *)st;
-(void) op3:(id<EXClientToServerPrx>)proxy;
@end;
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

```
id<EXClientToServerPrx> p = ...;           // Get proxy...

[p op1:42 f:3.14 b:YES s:@"Hello world!"]; // Pass literals

ICEInt i = 42;
ICEFloat f = 3.14;
BOOL b = YES;
```

```
NSString *s = @"Hello world!";

[p op1:i f:f b:b s:s];                      // Pass simple vars

EXNumberAndString *ns = [EXNumberAndString numberAndString:42
                                           str:@"The Answer"];
EXMutableStringSeq *ss = [ExMutableStringSeq array];
[ss addObject:@"Hello world!"];
EXStringTable *st = [EXMutableStringTable dictionary];
[ss setObject:ss forKey:[NSNumber numberWithInt:0]];

[p op2:ns ss:ss st:st];                     // Pass complex vars

[p op3:p];                                  // Pass proxy
```

You can pass either literals or variables to the various operations. The Ice run time simply marshals the value of the parameters to the server and leaves parameters otherwise untouched, so there are no memory-management issues to consider.

Note that the invocation of `op3` is somewhat unusual: the caller passes the proxy it uses to invoke the operation to the operation as a parameter. While unusual, this is legal (and no memory management issues arise from doing this.)

Passing `nil` and `NSNull`

The Slice language supports the concept of null ("points nowhere") for only two of its types: proxies and classes. For either type, `nil` represents a null proxy or class. For other Slice types, such as strings, the concept of a null string simply does not apply. (There is no such thing as a null string, only the empty string.) However, strings, structures, sequences, and dictionaries are all passed by pointer, which raises the question of how the Objective-C mapping deals with `nil` values.

As a convenience feature, the Objective-C mapping permits passing of `nil` as a parameter for the following types:

- Proxies (`nil` sends a null proxy.)
- Classes (`nil` sends a null class instance.)
- Strings (`nil` sends an empty string.)
- Structures (`nil` sends a default-initialized structure.)
- Sequences (`nil` sends an empty sequence.)
- Dictionaries (`nil` sends an empty dictionary.)

It is impossible to add `nil` to an `NSArray` or `NSDictionary`, so the mapping follows the usual convention that an `NSArray` element or `NSDictionary`

value that is conceptually `nil` is represented by `NSNull`. For example, to send a sequence of proxies, some of which are null proxies, you must insert `NSNull` values into the sequence.

As a convenience feature, if you have a sequence with elements of type string, structure, sequence, or dictionary, you can use `NSNull` as the element value. For elements that are `NSNull`, the Ice run time marshals an empty string, default-initialized structure, empty sequence, or empty dictionary to the receiver.

Similarly, for dictionaries with value type string, structure, sequence, or dictionary, you can use `NSNull` as the value to send the corresponding empty value (or default-initialized value, in the case of structures).

Out-Parameters

The Objective-C mapping passes out-parameters by pointer (for value types) and by pointer-to-pointer (for non-value types). Here is the Slice definition from page 590 once more, modified to pass all parameters in the `out` direction:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ClientToServer* proxy);
};
```

The Slice compiler generates the following code for this definition:

```
@protocol EXServerToClientPrx <ICEObjectPrx>
-(void) op1:(ICEInt *)i f:(ICEFloat *)f b:(BOOL *)b
            s:(NSMutableString **)s;
-(void) op2:(EXNumberAndString **)ns ss:(EXMutableStringSeq **)ss
            st:(EXMutableStringTable **)st;
-(void) op3:(id<EXClientToServerPrx> *)proxy;
@end
```

Note that, for types that come in immutable and mutable variants (strings, sequences, and dictionaries), the corresponding out-parameter uses the mutable variant.

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

```
id<EXServerToClientPrx> p = ...; // Get proxy...

ICEInt i;
ICEFloat f;
BOOL b;
NSMutableString *s;

[p op1:&i f:&f b:&b s:&s];
// i, f, b, and s contain updated values now

EXNumberAndString *ns;
EXStringSeq *ss;
EXStringTable *st;

[p op2:&ns ss:&ss st:&st];
// ns, ss, and st contain updated values now

[p op3:&p];
// p has changed now!
```

Again, there are no surprises in this code: the caller simply passes pointers to pointer variables to a method; once the operation completes, the values of those variables will have been set by the server.

Memory Management for Out-Parameters

When the Ice run time returns an out-parameter to the caller, it does not make any assumptions about the previous value of that parameter (if any). In other words, if you pass an initialized string as an out-parameter, the value you pass is simply discarded and the corresponding variable is assigned a new instance. As an example, consider the following operation:

```
void getString(out string s);
```

You could call this as follows:

```
NSMutableString *s = @"Hello";
[p getString:&s];
// s now points at the returned string.
```

All out-parameters are autoreleased by the Ice run time before they are returned. This is convenient because it does just the right thing with respect to memory management. For example, the following code does not leak memory:

```
NSMutableString *s = @"Hello";
[p getString:&s];
[p getString:&s]; // No leak here.
```

However, because the pointer value of out-parameters is simply assigned by the proxy method, you must be careful not to pass a variable as an out-parameter if that variable was not released or autoreleased:

```
NSMutableString *s = [[NSMutableString alloc] initWithString:
                                                  @"Hello"];
[p getString:&s]; // Bad news!
```

This code leaks the initial string because the proxy method assigns the passed pointer without calling `release` on it first. (In practice, this is rarely a problem because there is no need to initialize out-parameters and, if an out-parameter was initialized by being passed as an out-parameter to an operation earlier, its value will have been autoreleased by the proxy method already.)

It is worth having another look at the final call of the code example on page 594:

```
[p op3:&p];
```

Here, `p` is the proxy that is used to dispatch the call. That same variable `p` is also passed as an out-parameter to the call, meaning that the server will set its value. In general, passing the same parameter as both an input and output parameter is safe (with the caveat we just discussed).

Receiving Return Values

The Objective-C mapping returns return values in much the same way as out-parameters: value types are returned by value, and non-value types are returned by pointer. As an example, consider the following operations:

```
struct NumberAndString {
    int x;
    string str;
};

interface Ops {
```

```
    int getInt();
    string getString();
    NumberAndString getNumberAndString();
};
```

The proxy protocol looks as follows:

```
@protocol EXOpsPrx <ICEObjectPrx>
-(ICEInt) getInt;
-(NSMutableString *) getString;
-(EXNumberAndString *) getNumberAndString;
@end
```

Note that, for types with mutable and immutable variants (strings, sequences, and dictionaries), the formal return type is the mutable variant. As for out-parameters, anything returned by pointer is autoreleased by the Ice run time. This means that the following code works fine and does not contain memory management errors:

```
EXNumberAndString *ns = [NSNumberAndString numberAndString];
ns.x = [p getInt];
ns.str = [p getString]; // Autoreleased by getString,
                        // retained by ns.str.

[p getNumberAndString]; // No leak here.
```

The return value of `getString` is autoreleased by the proxy method but, during the assignment to the property `str`, the generated code calls `retain`, so the structure keeps the returned string alive in memory, as it should. Similarly, ignoring the return value from an invocation is safe because the returned value is autoreleased and will be reclaimed when the enclosing autorelease pool is drained.

Chained Invocations

Consider the following simple interface containing two operations, one to set a value and one to get it:

```
interface Name {
    string getName();
    void setName(string name);
};
```

Suppose we have two proxies to interfaces of type `Name`, `p1` and `p2`, and chain invocations as follows:

```
[p2 setName:[p1 getName]]; // No leak here.
```

This works exactly as intended: the value returned by `p1` is transferred to `p2`. There are no memory-management or exception safety issues with this code.

`nil` Out-Parameters and Return Values

If an out-parameter or return value is a proxy or class, and the operation returns a null proxy or class, the proxy method returns `nil`. If a proxy or class is returned as part of a sequence or dictionary, the corresponding entry is `NSNull`.

For strings, structures, sequences, and dictionaries, the Ice run-time *never* returns `nil` or `NSNull` (even if the server passed `nil` or `NSNull` as the value). Instead, the unmarshaling code always instantiates an empty string, empty sequence, or empty dictionary, and it always initializes structure members during unmarshaling, so structures that are returned from an operation invocation never contain a `nil` instance variable (except for proxy and class instance variables).

18.13 Exception Handling

Any operation invocation may throw a run-time exception (see Section 18.10.4 on page 581) and, if the operation has an exception specification, may also throw user exceptions (see Section 18.10 on page 576). Suppose we have the following simple interface:

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as Objective-C exceptions, so you can simply enclose one or more operation invocations in a `try–catch` block:

```
id<EXChildPrx> child = ...;    // Get proxy...
@try {
    [child askToCleanUp];      // Give it a try...
} @catch (EXTantrum *t) {
    printf("The child says: %s\n", t.reason_);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be dealt with by exception handlers higher in the hierarchy. For example:

```
void run()
{
    id<EXChildPrx> child = ...;       // Get proxy...
    @try {
        [child askToCleanUp];         // Give it a try...
    } @catch (EXTantrum *t) {
        printf("The child says: %s\n", t.reason);
        [child scold];                // Recover from error...
    }
    [child praise];                   // Give positive feedback...
}

int
main(int argc, char* argv[])
{
    int status = 1;
    @try {
        // ...
        run();
        // ...
        status = 0;
    } @catch (ICEException *e) {
        printf("Unexpected run-time error: %s\n", [e ice_name]);
    }
    // ...
    return status;
}
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Chapter 3.)

Exceptions and Out-Parameters

If an operation throws an exception, the Ice run time makes no guarantee for the value of out-parameters. Individual out-parameters may have the old value, the new value, or a value that is indeterminate, such that parts of the out-parameter have been assigned and others have not. However, no matter what their state, the values will be “safe” for memory-management purposes, that is, any out-parameters that were successfully unmarshaled are autoreleased.

Exceptions and Return Values

For return values, the Objective-C mapping provides the guarantee that a variable receiving the return value of an operation will not be overwritten if an exception is thrown.

18.14 Mapping for Local Interfaces

The Ice run time defines APIs using Slice (see Section 4.7). These APIs are provided as part of the Ice run time library and cannot be invoked remotely. (Doing so would not make any sense.) Therefore, the Slice interfaces for the Ice run time are defined as local interfaces (see Section 4.15). The Objective-C mapping for local interfaces differs from the default mapping in two ways:

- Local interfaces do not adopt an `<interface-name>Prx` protocol. (Doing so would be misleading because proxies imply that the target object can be remote.) Instead, the protocol for local interfaces has the same name as the interface. For example, the `Ice::Communicator` interfaces is defined as:

  ```
  ["objc:prefix:ICE"]
  module Ice {
      local interface Communicator {
          // ...
      }
  };
  ```

  Because `Communicator` is a local interface, objects of type `ICECommunicator` are passed as `id<ICECommunicator>` (*not* `ICECommunicator *` or `id<ICEComunicatorPrx>`).

- Types that come in mutable and immutable variants (strings, sequences, and dictionaries) are always passed as the immutable variant. For example, the `getName` operation on the `ObjectAdapter` interface is defined as:

  ```
  ["objc:prefix:ICE"]
  module Ice {
      local interface ObjectAdapter {
          string getName();
      };
  };
  ```

  Because `ObjectAdapter` is a local interface, the `getName` operation maps to:

```
-(NSString *) getName;
```

Note that the returned string is of type `NSString` instead of `NSMutableString` (as would be the case for an operation on a non-local interface).

For local interfaces, parameters are passed as the immutable version because their values are not meant to be modified by application code. In addition, passing the immutable version avoids an unnecessary data copy.

18.15 Mapping for Classes

Slice classes are mapped similar to structures and exceptions. The generated class contains an instance variable and a property for each Slice data member. Consider the following class definition:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler generates the following code for this definition:

```
@interface EXTimeOfDay : ICEObject
{
    ICEShort hour;
    ICEShort minute;
    ICEShort second;
}

@property(nonatomic, assign) ICEShort hour;
@property(nonatomic, assign) ICEShort minute;
@property(nonatomic, assign) ICEShort second;

-(id) init:(ICEShort)hour minute:(ICEShort)minute
           second:(ICEShort)second;
+(id) timeOfDay;
+(id) timeOfDay:(ICEShort)hour minute:(ICEShort)minute
                second:(ICEShort)second;
@end
```

There are a number of things to note about the generated code:

1. The generated class `EXTimeOfDay` derives from `ICEObject`, which is the parent of all classes. Note that `ICEObject` is *not* the same as `ICEObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The generated class contains a property for each Slice data member.
3. The generated class provides an `init` method that accepts one argument for each data member, and it provides the same two convenience constructors as structures and exceptions.

18.15.1 Derivation from `ICEObject`

All classes ultimately derive from a common base class, `ICEObject`. Note that this is not the same as implementing the `ICEObjectPrx` protocol (which is implemented by proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`ICEObject` defines a number of methods:

```
@protocol ICEObject <NSObject>
-(BOOL) ice_isA:(NSString*)typeId current:(ICECurrent*)current;
-(void) ice_ping:(ICECurrent*)current;
-(NSString*) ice_id:(ICECurrent*)current;
-(NSArray*) ice_ids:(ICECurrent*)current;
@end

@interface ICEObject NSObject <ICEObject, NSCopying>
-(BOOL) ice_isA:(NSString*)typeId;
-(void) ice_ping;
-(NSString*) ice_id;
-(NSArray*) ice_ids;
+(NSString*) ice_staticId;
-(void) ice_preMarshal;
-(void) ice_postUnmarshal;
-(BOOL) ice_dispatch:(id<ICERequest>)request;
-(id) initWithDelegate:(id)delegate;
+(id) objectWithDelegate:(id)delegate;
@end
```

The methods of `ICEObject` behave as follows:[4]

- `ice_isA`

  This function returns `YES` if the object supports the given type ID, and `NO` otherwise.

- `ice_ping`

  `ice_ping` provides a basic reachability test for the class. If it completes without raising an exception, the class exists and is reachable.[5]

- `ice_ids`

  This function returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.

- `ice_id`

  This function returns the actual run-time type ID for a class. If you call `ice_id` via a pointer to a base instance, the returned type ID is the actual (possibly more derived) type ID of the instance.

- `ice_staticId`

  This function returns the static type ID of a class.

- `ice_preMarshal`

  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.

- `ice_postUnmarshal`

  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.

- `ice_dispatch`

  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors (see Section 32.23).

- `initWithDelegate`
  `objectWithDelegate`

  These constructors enable the implementation of servants with a delegate (see page 634).

---

4. The methods are split between the `ICEObject` protocol and class because classes can be servants.
5. `ice_ping` is normally only invoked on the proxy for a class that might be remote because a class instance that is local (in the caller's address space) can always be reached.

18.15.2 Data Members of Classes

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding property.

18.15.3 Class Constructors

Classes provide the usual `init` method and a parameter-less convenience constructor that perform default initialization of the class's instance variables. In addition, if a class has data members, it provides an `init` method and a convenience constructor that accept one argument for each data member. This allows you to allocate and initialize a class instance in a single statement (instead of first having to allocate and default-initialize the instance and then assigning to its properties).

For derived classes, the `init` method and the convenience constructor have one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order. For example:

```
class Base {
    int i;
};

class Derived extends Base {
    string s;
};
```

This generates:

```
@interface EXBase : ICEObject
// ...

@property(nonatomic, assign) ICEInt i;

-(id) init:(ICEInt)i;
+(id) base;
+(id) base:(ICEInt)i;
@end

@interface EXDerived : EXBase
// ...

@property(nonatomic, retain) NSString *s;
```

```
-(id) init:(ICEInt)i s:(NSString *)s;
+(id) derived;
+(id) derived:(ICEInt)i s:(NSString *)s;
@end
```

18.15.4 Derived Classes

Note that, in the preceding example, the derivation of the Slice definitions is preserved for the generated classes: `EXBase` derives from `ICEObject`, and `EXDerived` derives from `EXBase`. This allows you to treat and pass classes polymorphically: you can always pass an `EXDerived` instance where an `EXBase` instance is expected.

18.15.5 Passing Classes as Parameters

Classes are passed by pointer, like any other Objective-C object. For example, here is an operation that accepts a `Base` as an in-parameter and returns a `Derived`:

```
Derived getDerived(Base d);
```

The corresponding proxy method looks as follows:

```
-(EXDerived *) getDerived:(EXBase *)d;
```

To pass a null instance, you simply pass `nil`.

18.15.6 Operations of Classes

If you look back at the code that is generated for the `EXTimeOfDay` class (see page 600), you will notice that there is no indication at all that the class has a `format` operation. As opposed to proxies, classes do not implement any protocol that would define which operations are available. This means that you can partially implement the operations of a class. For example, you might have a Slice class with five operations that is returned from a server to a client. If the client uses only one of the five operations, the client-side code needs to implement only that one operation and can leave the remaining four operations without implementation. (If the class were to implement a mandatory protocol, the client-side code would have to implement all operations in order to avoid a compiler warning.)

Of course, you must implement those operations that you actually intend to call. The mapping of operations for classes follows the *server-side* mapping for operations on interfaces: parameter types and labels are exactly the same. (See

Section 20.5 for details.) In a nutshell, the server-side mapping is the same as the client-side mapping except that, for types that have mutable and immutable variants, they map to the immutable variant where the client-side mapping uses the mutable variant, and vice versa.

For example, here is how we could implement the `format` operation of our `TimeOfDay` class:

```
@interface TimeOfDayI : EXTimeOfDay
@end

@implementation TimeOfDayI
-(NSString *) format
{
    return [NSString stringWithFormat:@"%.2d:%.2d:%.2d",
                        self.hour, self.minute, self.second];
}
@end
```

By convention, the implementation of classes with operations has the same name as the Slice class with an `I`-suffix. Doing this is not mandatory—you can call your implementation class anything you like. However, if you do not want to use the `I`-suffix naming, we recommend that you adopt another naming convention and follow it consistently.

Note that `TimeOfDayI` derives from `EXTimeOfDay`. This is because, as we will see in a moment, the Ice run time will instantiate a `TimeOfDayI` instance whenever it receives a `TimeOfDay` instance over the wire and expects that instance to provide the properties of `EXTimeOfDay`.

18.15.7 Class Factories

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDayI` class. However, unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements a `format` method. To allow the Ice run time to instantiate the correct

object, we must provide a factory that knows that the Slice `TimeOfDay` class is implemented by our `TimeOfDayI` class. The `Ice::Communicator` interface provides us with the necessary operations:

```
["objc:prefix:ICE"]
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };
};
```

The object factory's `create` operation is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` operation is called by the Ice run time when its communicator is destroyed. A possible implementation of our object factory is:

```
@interface ObjectFactory<ICEObjectFactory>
@end

@implementation ObjectFactory
-(ICEObject*) create:(NSString *)type
{
    NSAssert([type isEqualToString:@"::Example::TimeOfDay"]);
    return [[TimeOfDayI alloc] init];
}
@end
```

The `create` method is passed the type ID (see Section 4.13) of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::Example::TimeOfDay"`.

Our implementation of `create` checks the type ID: if it is `"::Example::TimeOfDay"`, it instantiates and returns a `TimeOfDayI` object. For other type IDs, it asserts because it does not know how to instantiate other types of objects.

Note that your factory *must not* autorelease the returned instance. The Ice run time takes care of the necessary memory management activities on your behalf.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

```
id<ICECommunicator> ice = ...;
ObjectFactory *factory =
    [[[ObjectFactory alloc] init] autorelease];
[ic addObjectFactory:factory sliceId:@"::Example::TimeOfDay"];
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::Example::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator—if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called. However, the Ice run time may make concurrent calls to `create`.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not (but can) create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

Using a Category to Implement Operations

An alternative to registering a class factory is to use an Objective-C category to implement operations. For example, we could have implemented our `format` method using a category instead:

```
@interface EXTimeOfDay (TimeOfDayI)
@end

@implementation EXTimeOfDay (TimeOfDayI)
-(NSString *) format
{
    return [NSString stringWithFormat:@"%.2d:%.2d:%.2d",
                        self.hour, self.minute, self.second];
}
@end
```

In this case, there is no need to derive from the generated `EXTimeOfDay` class because we provide the format implementation as a category. There is also no need to register a class factory: the Ice run time instantiates an `EXTimeOfDay` instance when a `TimeOfDay` instance arrives over the wire, and the `format` method is found at run time when it is actually called.

This is a viable alternative approach to implement class operations. However, keep in mind that, if the operation implementation requires use of instance variables that are not defined as part of the Slice definitions of a class, you cannot use this approach because Objective-C categories do not permit you to add instance variables to a class.

18.15.8 Copying of Classes

Classes implement `NSCopying`. The behavior is the same as for structures: instance variables of value type are copied by assignment, instance variables of pointer type are copied by calling `retain`, that is, the copy is shallow. To illustrate this, consider the following class definition:

```
class Node {
    int i;
    string s;
    Node next;
};
```

We can initialize two instances of type `EXNode` as follows:

```
NSString lastString = [NSString stringWithString:@"last"];
EXNode *last = [EXNode node:99 s:lastString next:nil];

NSString firstString = [NSString stringWithString:@"first"];
EXNode *first = [EXNode node:1 s:firstString next:last];
```

This creates the situation shown in Figure 18.2.

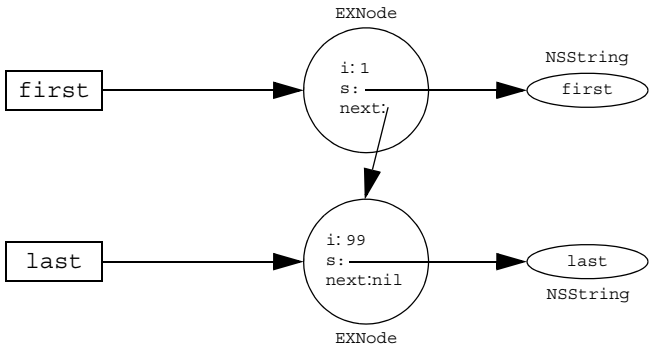


Figure 18.2. Two instances of type `EXNode`.

Now we create a copy of the first node by calling `copy`:

```
EXNode *copy = [[first copy] autorelease];
```

This creates the situation shown in Figure 18.3.

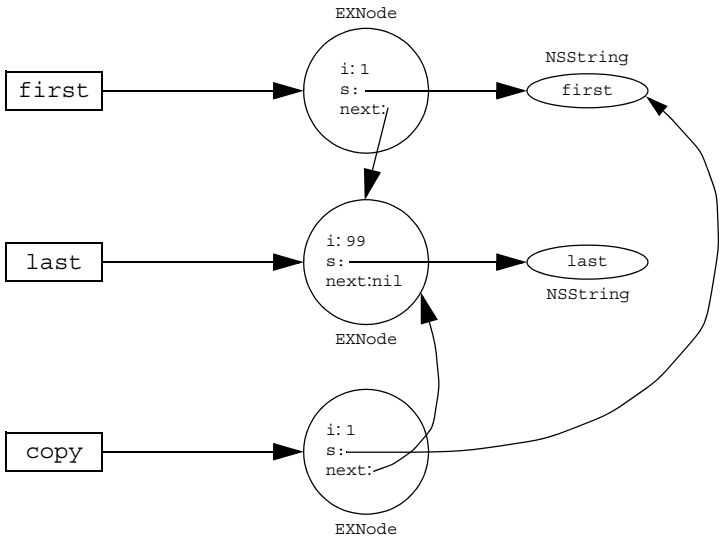


Figure 18.3. `EXNode` instances after calling `copy` on `first`.

As you can see, the first node is copied, but the last node (pointed at by the `next` instance variable of the first node) is not copied; instead, `first` and `copy` now both have their `next` instance variable point at the same last node, and both point at the same string.

Cyclic References

One thing to be aware about are cyclic references among classes. As an example, we can easily create a cycle by executing the following statements:

```
EXNode *first = [EXNode node];
ExNode *last = [EXNode node];
first.next = last;
last.next = first;
```

This makes the `next` instance variable of the two classes point at each other, creating the cycle shown in Figure 18.4.

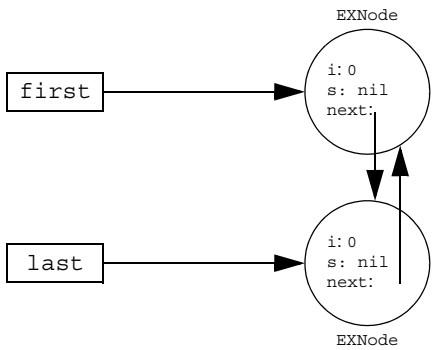


Figure 18.4. Two nodes with cyclic references.

There is no problem with sending this class graph as a parameter. For example, you could pass either `first` or `last` as a parameter to an operation and, in the server, the Ice run time will faithfully rebuild the corresponding graph, preserving the cycle. However, if a server returns such a graph from an operation invocation as the return value or as an out-parameter, all class instances that are part of a cycle are leaked. The same is true on the client side: if you receive such a graph from an operation invocation and do not explicitly break the cycle, you will leak all instances that form part of the cycle.

Because it is difficult to break cycles manually (and, on the server side, for return values and out-parameters, it is impossible to break them), we recommend that you avoid cyclic references among classes.[6]

18.16 Interfaces by Value

Slice permits you to pass an interface by value:

```
interface ClassBase {
    void someOp();
    // ...
};

interface Processor {
    ClassBase process(ClassBase b);
};

class SomeClass implements ClassBase {
    // ...
};
```

Note that `process` accepts and returns a value of type `ClassBase`. This is *not* the same as passing a `ClassBase*`, which is a *proxy* to an object of type `ClassBase` that is possibly remote. Instead, what is passed here is an *interface*, and the interface is passed *by value*.

The immediate question is "what does this mean?" After all, interfaces are abstract and, therefore, it is impossible to pass an interface by value. The answer is that, while an interface cannot be passed, what *can* be passed is a class that implements the interface. That class is type compatible with the formal parameter type and, therefore, can be passed by value. In the preceding example, `SomeClass` implements ClassBase and, hence, can be passed to and returned from the `process` operation.

The Objective-C mapping maps interface-by-value parameters to `ICEObject*`, regardless of the type of the interface. For example, the proxy protocol for the `process` operation is:

```
-(ICEObject *) process:(ICEObject *)b;
```

6. A future version of the Objective-C run time may provide a garbage collector similar to the one used by Ice for C++.

This means that you can pass a class of any type to the operation, even if it is not type-compatible with the formal parameter type, because all classes derive from `ICEObject`. However, an invocation of `process` is still type-safe at run time: the Ice run time verifies that the class instance that is passed implements the specified interface; if not, the invocation throws an `ICEMarshalException`.

Passing interfaces by value as `ICEObject*` is a consequence of the decision to not generate a formal protocol for classes. (If such a protocol would exist, the formal parameter type could be `id<`*`ProtocolName`*`>`. However, as we mentioned on page 604, a protocol would require the implementation of a class to implement all of its operations, which can be inconvenient. Because it is rare to pass interfaces by value (more often, the formal parameter type will be a base *class* instead of a base *interface*), the minor loss of static type safety is an acceptable trade-off.

18.17 Asynchronous Method Invocation (AMI)

Asynchronous Method Invocation (*AMI*) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

To use AMI with Objective-C, you must annotate your Slice definitions with an `["objc:ami"]` metadata directive. This directive instructs **`slice2objc`** to generate AMI support in addition to synchronous API (which is always generated).

The metadata directive applies interfaces or operations, for example:

```
["objc:ami"] interface I {
  bool isValid();
  float computeRate();
};
```

```
interface J {
  void startProcess();
  ["objc:ami"] int endProcess();
};
```

In this example, all proxy methods of interface `I` are generated with support for synchronous and asynchronous invocations. In interface `J`, the `startProcess` operation uses asynchronous dispatch, and the `endProcess` operation supports asynchronous invocation and dispatch.

Specifying metadata at the operation level, rather than at the interface or class level, not only minimizes the amount of generated code, but more importantly, it minimizes complexity. Although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations for which it provides a particular advantage, while using the simpler synchronous model for the rest.

Proxy Methods

Besides the synchronous proxy methods, **`slice2objc`** generates an additional proxy method with the name *`<operation>`*`_async`. For example, consider the following definition:

```
["ami"] interface Intf {
    string op(int i, string s, out double d, out bool b);
};
```

The corresponding asynchronous proxy method is:[7]

```
-(BOOL) op_async:(id)target_
                 response:(SEL)response_
                 exception:(SEL)exception_
                 i:(ICEInt)i
                 s:(NSString *)s;
```

The return value and the parameters work as follows:

- An asynchronous invocation returns `YES` if it was written to the local transport and `NO` if it was queued for later delivery.
- The first parameter (`target`) of an asynchronous operation is the callback object that is notified once the invocation completes.

7. Each asynchronous operation actually results in four methods. We will return to these once we have covered the basics.

- The second parameter (`response`) is the selector of a method of the `target` callback object. This method is called by the Ice run time once the operation invocation completes successfully. In other words, this selector is used if the operation did not raise an exception.
- The third parameter (`exception`) is the selector of a method of the `target` callback object. This method is called by the Ice run time once the operation invocation completes unsuccessfully. In other words, this selector is used if the operation did raise an exception.
- The remaining parameters are the in-parameters for the operation, in the order in which they are defined in Slice. (Out-parameters are not visible on the invoking end of the API.)

Given a proxy to an object of type `Intf`, you can invoke `op` assynchronously as follows:

```
id<EXIntfPrx> proxy = ...; // Get proxy...

@try {
    [proxy op_async:cb response:@selector(opResponse:d:b:)
                    exception:@selector(opException:)
                    i:99 s:@"Hello"]) {
} @catch (ICECommunicatorDestroyedException *ex)
        // Communicator no longer exists.
}
```

Note that the in-parameters passed to the operation are `99` and `"Hello"`. (We will return to the callback parameters shortly.)

In your code, you are unlikely to catch `ICECommunicatorDestroyedException` for every asynchronous invocation. Instead, it usually is easier to catch this exception higher up in the call hierarchy and deal with it there. (After all, this indication most likely indicates that you have initiated program termination.) We have included the catch handler here for illustration purposes only.

Once the Ice run time has successfully initiated an asynchronous invocation, control returns to the caller. The actual invocation is processed in the background (if it could not be written to the network immediately). Eventually, the operation will complete, either successfully or with an exception.

If the operation raised an exception, the Ice run time delivers the exception to the selector that you passed to the invocation. The exception callback method accepts a single argument of type `ICEException` that informs it of the cause of the failure. The `exception` callback must have `void` return type.

If the operation completed successfully, the Ice run time calls the `response` callback method whose selector you passed to the invocation (`opResponse:d:b:` for the preceding example). The `response` callback must have `void` return type. The rules for how the parameter list of the `response` callback is formed are as follows:

- If an operation has `void` return type and does not use out-parameters, the `response` callback has no arguments.
- If an operation has non-`void` return type, the first parameter of the `response` callback is the return value of the operation.
- If an operation has out-parameters, each out-parameters becomes an argument for the `response` callback, in the same order as the order of out-parameters in the corresponding Slice definition. The arguments for out-parameters follow the argument for the return value (if any).

For our `op_async` invocation, here is how we could write our callback object to process the results:

```
@interface Callback
// ...
@end

@implementation Callback
-(void) opResponse:(NSString *)ret d:(ICEDouble) b:(BOOL)b
{
    // Process results...
}

-(void) opException:(ICEException *)ex
{
    // Handle exception...
}
@end
```

Of course, you are free to add constructors and other methods to your callback object. For example, it is common for the callback object to store a reference to an object that can further process the results (such as display them to the user). Typically, the constructor stores that reference in an instance variable.

On page 613, we mentioned that each Slice operation generates four `_async` methods. Here is one of these for our example operation `op`:

```
-(BOOL) op_async:(id)target_
                 response:(SEL)response_
                 exception:(SEL)exception_
                 i:(ICEInt)i
                 s:(NSString *)s
                 context:(ICEContext *)context;
```

This is exactly the same as the version we have already seen, except for the trailing `context` parameter. This parameter allows you to pass a context with asynchronous operations. (See Section 32.12 for details on contexts.)

The remaining two variants also come in "without context" and "with context" versions, but also have a `sent` parameter:

```
-(BOOL) op_async:(id)target_
                 response:(SEL)response_
                 exception:(SEL)exception_
                 sent:(SEL)sent_
                 i:(ICEInt)i
                 s:(NSString *)s;
-(BOOL) op_async:(id)target_
                 response:(SEL)response_
                 exception:(SEL)exception_
                 sent:(SEL)sent_
                 i:(ICEInt)i
                 s:(NSString *)s
                 context:(ICEContext *)context;
```

When you invoke an operation asynchronously, the Ice run time attempts to write the invocation to the local network buffers immediately. However, if doing so would block, the invocation is instead queued for later processing in the background. In other words, once control returns to you after making an asynchronous invocation, you do not know whether the invocation was written to the local transport or whether it will be sent some time later.

The purpose of the `sent` parameter (which always follows the `exception` parameter and precedes the in-parameters) is to notify you if an asynchronous invocation could not be written immediately. If the invocation was queued because it could not be written to the local transport, the Ice run time calls the `sent` callback. The `sent` callback has void return type and accepts no parameters:

```
@implementation Callback
// ...

-(void) sent
```

```
{
    // Invocation was queued for later delivery.
}
@end
```

The reason for providing this callback is that, without it, a client could flood the Ice run time with asynchronous requests and run out of memory. For example, the client might asynchronously invoke operations in a loop. If the network is temporarily congested, or the client loses connectivity, all the client's invocations will end up being queued in the Ice run time (at least until they time out, if the client has configured a timeout). Of course, there is only a limited amount of buffer space available and, unless the client realizes that all its invocations are piling up in memory, it will die an untimely death.

The `sent` callback allows you to implement flow-control for asynchronous invocations. If an asynchronous invocation returns `NO`, you know that the invocation has not been written to the local transport yet. In that case, you can increment a counter to keep track of the number of queued invocations. In the `sent` callback, you can decrement that counter again. This mechanism allows you to limit the number of queued invocations and avoid running out of memory.

18.17.1 Concurrency Issues

Support for asynchronous invocations in Ice is enabled by the client thread pool (see Section 32.10), whose threads are primarily responsible for processing reply messages. It is important to understand the concurrency issues associated with asynchronous invocations:

- A callback object must not be used for multiple simultaneous invocations. An application that needs to aggregate information from multiple replies can create a separate object to which the callback objects delegate.
- Calls to the callback object are always made by threads from an Ice thread pool, therefore synchronization may be necessary if the application might interact with the callback object at the same time as the reply arrives. Furthermore, since the Ice run time never invokes callback methods from the client's calling thread, the client can safely make AMI invocations while holding a lock without risk of a deadlock.
- The number of threads in the client thread pool determines the maximum number of simultaneous callbacks possible for asynchronous invocations. The default size of the client thread pool is one, meaning invocations on callback objects are serialized. If the size of the thread pool is increased, the application

may require synchronization, and replies can be dispatched out of order. The client thread pool can also be configured to serialize messages received over a connection so that AMI replies from a connection are dispatched in the order they are received (see Section 32.10.4).

- AMI invocations do not use collocation optimization (see Section 32.20). As a result, AMI invocations are always sent "over the wire" and thus are dispatched by the server thread pool.

18.17.2 Flushing Batch Requests

Applications that send batched requests (see Section 32.16) can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides an asynchronous version of this method for applications that wish to flush batch requests without the risk of blocking.

The proxy method `ice_flushBatchRequests_async` initiates an asynchronous flush. Its only argument is a callback object; this object must define an `ice_exception` method for receiving a notification if an error occurs before the message is sent.

If the application is interested in flow control (see page 616), the return value of `ice_flushBatchRequests_async` is a boolean indicating whether the message was sent synchronously. Furthermore, the callback object can define an `ice_sent` method that is invoked when an asynchronous flush completes.

18.17.3 Limitations

AMI invocations cannot be sent using collocated optimization. If you attempt to invoke an AMI operation using a proxy that is configured to use collocation optimization, the Ice run time raises `CollocationOptimizationException` if the servant happens to be collocated; the request is sent normally if the servant is not collocated. Section 32.21 provides more information about this optimization and describes how to disable it when necessary.

18.18 slice2objc Command-Line Options

The Slice-to-Objective-C compiler, **slice2objc**, offers the following command-line options in addition to the standard options described in Section 4.20:

- **--include-dir *DIR***

  Modifies #import directives in source files to prepend the path name of each header file with the directory ***DIR***.

- **--output-dir *DIR***

  Places the generated source files into the specified output directory ***DIR***.

- **--depend**

  Prints makefile dependency information to standard output. No code is generated when this option is specified. The output generally needs to be filtered before it can be included in a makefile; the Ice build system uses the script config/makedepend.py for this purpose.

- **--depend-xml**

  Prints dependency information to standard output in XML format. No code is generated when this option is specified. This option is intended for use with Apple's Xcode development environment.

Chapter 19
Developing a File System Client in Objective-C

19.1 Chapter Overview

In this chapter, we present the source code for an Objective-C client that accesses the file system we developed in Chapter 5 (see Chapter 21 for the corresponding server).

19.2 The Objective-C Client

We now have seen enough of the client-side Objective-C mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

```
["objc:prefix:FS"]
module Filesystem {
    exception GenericError {
        string reason;
    };

    interface Node {
        idempotent string name();
    };

    sequence<string> Lines;
```

```
    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

```
#import <Ice/Ice.h>
#import <Filesystem.h>

#import <Foundation/NSAutoreleasePool.h>
#import <stdio.h>

static void
listRecursive(id<FSDirectoryPrx> dir, int depth)
{
    // ...
}

int
main(int argc, char* argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int status = 1;
    id<ICECommunicator> communicator;
    @try
    {
        communicator = [ICEUtil createCommunicator:&argc
                                                argv:argv];

        // Create a proxy for the root directory
        //
        id<FSDirectoryPrx> rootDir = [FSDirectoryPrx checkedCast:
```

```
                [communicator stringToProxy:
                    @"RootDir:default -p 10000"]];
        if (!rootDir)
            [NSException raise:@"invalid proxy" format:@"nil"];

        // Recursively list the contents of the root directory
        //
        printf("Contents of root directory:\n");
        listRecursive(rootDir, 0);

        status = 0;
    } @catch (NSException *ex) {
        NSLog(@"%@\n", [ex name]);
    }

    @try {
        [communicator destroy];
    } @catch (NSException* ex) {
        NSLog(@"%@\n", [ex name]);
    }

    [pool release];
    return status;
}
```

1. The code imports a few header files:

   1. `Ice/Ice.h`

      This file is always included in both client and server source files. It provides definitions that are necessary for accessing the Ice run time.

   2. `Filesystem.h`

      This is the header that is generated by the Slice compiler from the Slice definitions in `Filesystem.ice`.

   3. `NSAutoreleasePool.h`

      The client uses an autorelease pool to reclaim memory before it exits.

   4. `stdio.h`

      The implementation of `listRecursive` prints to `stdout`.

2. The structure of the code in `main` follows what we saw in Chapter 3. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.

3. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`:

```
// Print the specified number of tabs.

static void
printIndent(int depth)
{
    while (depth-- > 0)
        putchar('\t');
}

// Recursively print the contents of directory "dir" in tree
// fashion.  For files, show the contents of each file.
// The "depth" parameter is the current nesting level
// (for indentation).

static void
listRecursive(id<FSDirectoryPrx> dir, int depth)
{
    ++depth;
    FSNodeSeq *contents = [dir list];

    for (id<FSNodePrx> node in contents) {
        id<FSDirectoryPrx> dir =
            [FSDirectoryPrx checkedCast:node];
        id<FSFilePrx> file = [FSFilePrx uncheckedCast:node];
        printIndent(depth);
        printf("%s%s\n", [[node name] UTF8String],
                         (dir ? " (directory):" : " (file):"));
        if (dir) {
            listRecursive(dir, depth);
        } else {
            FSLines *text = [file read];
            for (NSString *line in text) {
                printIndent(depth);
                printf("\t%s\n", [line UTF8String]);
            }
        }
    }
}
```

The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the

name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `id<FSDirectoryPrx>` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the `Node` *is-a* `File` and, therefore, an `uncheckedCast` is sufficient to get an `id<FSFilePrx>`.

   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.
2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.
3. The code checks the type of the node:
   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of two files and a directory as follows:

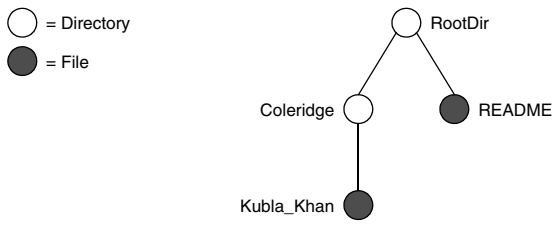


Figure 19.1. A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
```

```
Kubla_Khan (file):
        In Xanadu did Kubla Khan
        A stately pleasure-dome decree:
        Where Alph, the sacred river, ran
        Through caverns measureless to man
        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in Chapter 38 and Chapter 34.

19.3 Summary

This chapter presented a very simple client to access a server that implements the file system we developed in Chapter 5. As you can see, the Objective-C code hardly differs from the code you would write for an ordinary Objective-C program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Objective-C object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. As we will see in Chapter 20, this is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

Chapter 20
Server-Side Slice-to-Objective-C Mapping

20.1 Chapter Overview

In this chapter, we present the server-side Slice-to-Objective-C mapping (see Chapter 18 for the client-side mapping). Section 20.3 discusses how to initialize and finalize the server-side run time, sections 20.4 to 20.6 show how to implement interfaces and operations, and Section 20.7 discusses how to register objects with the server-side Ice run time.

20.2 Introduction

The mapping for Slice data types to Objective-C is identical on the client side and server side, except for operation parameters, which map slightly differently for types that have mutable and immutable variants (strings, sequence, and dictionaries). This means that the mappings in Chapter 18 also apply to the server side. However, for the server side, there are a few additional things you need to know, specifically:

- how to initialize and finalize the server-side run time
- how to implement servants
- how to pass parameters and throw exceptions

- how to create servants and register them with the Ice run time.

We discuss these topics in the remainder of this chapter.

20.3 The Server-Side `main` Function

The main entry point to the Ice run time is represented by the local interface `ICECommunicator`. As for the client side, you must initialize the Ice run time by calling `createCommunicator` (a class method of the `ICEUtil` class) before you can do anything else in your server. `createCommunicator` returns an instance of type `id<ICECommunicator>`:

```
int
main(int argc, char* argv[])
{
    // ...
    id<ICECommunicator> communicator =
        [ICEUtil createCommunicator:&argc argv:argv];
    // ...
}
```

`createCommunicator` accepts a *pointer* to `argc` as well as `argv`. The class method scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `createCommunicator` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `createCommunicator` throws an exception.

Before leaving your `main` function, you *must* call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, `destroy` waits for any operation implementations that are still executing in the server to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your `main` function to terminate without calling `destroy` first; doing so has undefined behavior.

The general shape of our server-side `main` function is therefore as follows:

```
#import <Ice/Ice.h>

int
main(int argc, char* argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

```
    int status = 1;
    id<ICECommunicator> communicator = nil;
    @try {
        communicator =
            [ICEUtil createCommunicator:&argc argv:argv];

        // Server code here...

        status = 0;
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    @try {
        [communicator destroy];
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    [pool release];
    return status;
}
```

Note that the code places the call to `createCommunicator` into a `try` block and takes care to return the correct exit status to the operating system. Also note that the code creates and releases an autorelease pool. This ensures that memory will be released before the program terminates.

The `catch` handler for `NSException` ensures that the communicator is destroyed regardless of whether the program terminates normally or due to an exception.

You must *not* release the communicator that is returned by `createCommunicator`. As for any operation that returns a pointer, the Ice run time calls `autorelease` on the returned instance, so you do not have to release it yourself.[1]

1. This is also the reason why `createCommunicator` is not called `initialize` (as it is for other language mappings)—`initialize` would suggest that the return value must be released because the method name begins with `init`.

Alternative Ways to Create a Communicator

`createCommunicator` is provided in several versions that accept different arguments. Here is the complete list:

- `(id<ICECommunicator>) createCommunicator:`
  `(int*)argc argv:(char*[])argv`
  `initData:(ICEInitializationData*)initData;`

  This is the designated initializer—the remaining versions of `createCommunicator` are implemented in terms of this initializer.

  As for the version we saw in the preceding section, this version accepts a pointer to `argc` as well as `argv` and removes Ice-related command-line options from the argument vector.

  The `initData` argument allows you to pass additional initialization information to the Ice run time (see below).

- `+(id<ICECommunicator>) createCommunicator;`

  This is equivalent to calling

  ```
  [ICEUtil createCommunicator:nil argv:nil initData:nil];
  ```

- `+(id<ICECommunicator>) createCommunicator:`
  `(int*)argc argv:(char*[])argv;`

  This is equivalent to calling

  ```
  [ICEUtil createCommunicator:&argc argv:argv initData:nil];
  ```

- `+(id<ICECommunicator>) createCommunicator:`
  `(ICEInitializationData*)initData`

  This is equivalent to calling

  ```
  [ICEUtil createCommunicator:nil argv:nil initData:initData];
  ```

The `initData` argument is of type `ICEInitializationData`. Even though it has no Slice definition, this class behaves as if it were a Slice structure with the following definition:

```
#include <Properties.ice>
#include <Logger.ice>

["objc:prefix:ICE"]
module Ice {
    dictionary<string, string> PrefixDict;

    local struct InitializationData {
```

```
        Ice::Properties properties;
        Ice::Logger logger;
    };
};
```

The `properties` member allows you to explicitly set property values for the communicator to be created. This is useful, for example, if you want to ensure that a particular property setting is always used by the communicator. (See Chapter 30 for more information.)

The `logger` member sets the logger that the Ice run time uses to log messages. If you do not set a logger (leaving the `logger` member as `nil`), the run time installs a default logger that calls `NSLog` to log messages. (See Section 32.19 for more information on loggers.)

20.4 Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing methods in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

20.4.1 Skeleton Classes

On the client side, interfaces map to proxy protocols and classes (see Section 18.11). On the server side, interfaces map to *skeleton* protocols and classes. A skeleton is a class that has a method for each operation on the corresponding interface. For example, consider the Slice definition for the `Node` interface we defined in Chapter 5 once more:

```
["objc:prefix:FS"]
module Filesystem {
    interface Node {
        idempotent string name();
    };
// ...
};
```

The Slice compiler generates the following definition for this interface:

```
@protocol FSNode <ICEObject>
-(NSString *) name:(ICECurrent *)current;
@end

@interface FSNode : ICEObject
// ...
@end
```

As you can see, the server-side API consists of a protocol and a class, known as the *skeleton protocol* and *skeleton class*. The methods of the skeleton class are internal to the mapping, so they do not concern us here. The skeleton protocol defines one method for each Slice operation. As for the client-side mapping, the method name is the same as the name of the corresponding Slice operation. If the Slice operation has parameters or a return value, these are reflected in the generated method, just as they are for the client-side mapping. In addition, each method has an additional trailing parameter of type `ICECurrent`. This parameter provides additional information about an invocation to your server-side code (see Section 32.6).

As for the client-side mapping, the generated code reflects the fact that all Slice interfaces and classes ultimately derive from `Ice::Object`. As you can see, the generated protocol incorporates the `ICEObject` protocol, and the generated class derives from the `ICEObject` class.

20.4.2 Servant Classes

The Objective-C mapping supports two different ways to implement servants. You can implement a servant by deriving from the skeleton class and implementing the methods for the Slice operations in your derived class. Alternatively, you can use a delegate servant, which need not derive from the skeleton class.

Derived Servants

To provide an implementation for an Ice object, you can create a servant class that derives from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

```
@interface NodeI : FSNode <FSNode>
{
    @private
        NSString *myName;
```

```
}

+(id) nodei:(NSString *)name;
@end
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can chose any name you prefer for your servant classes.)

Note that `NodeI` derives from `FSNode`, that is, it derives from its skeleton class. In addition, it adopts the `FSNode` protocol. Adopting the protocol is not strictly necessary; however, if you do write your servants this way, the compiler emits a warning if you forget to implement one or more Slice operations for the corresponding interface, so we suggest that you make it a habit to always have your servant class adopt its skeleton protocol.

As far as Ice is concerned, the `NodeI` class must implement the single `name` method that is defined by its skeleton protocol. That way, the Ice run time gets a servant that can respond to the operation that is defined by its Slice interface. You can add other methods and instance variables as you see fit to support your implementation. For example, in the preceding definition, we added a `myName` instance variable and property, a convenience constructor, and `dealloc`. Not surprisingly, the convenience constructor initializes the `myName` instance variable, the `name` method returns the value of that variable, and `dealloc` releases it:

```
@implementation NodeI

+(id) nodei:(NSString *)name
{
    NodeI *instance = [[[NodeI alloc] init] autorelease];
    instance.myName = [[name copy] retain];
    return instance;
}

-(NSString *) name:(ICECurrent *)current
{
    return myName;
}

-(void) dealloc
{
```

```
    [myName release];
    [super dealloc];
}
@end
```

Delegate Servants

An alternate way to implement a servant is to use a delegate. `ICEObject` provides two constructors to do this:

```
@interface ICEObject NSObject <ICEObject, NSCopying>
// ...
-(id) initWithDelegate:(id)delegate;
+(id) objectWithDelegate:(id)delegate;
@end
```

The `delegate` parameter specifies an object to which the servant will delegate operation invocations. That object need not derive from the skeleton class; the only requirement on the delegate is that it must have an implementation of the methods corresponding to the Slice operations that are called by clients. As for derived servants, we suggest that the delegate adopt the skeleton protocol, so the compiler will emit a warning if you forget to implement one or more Slice operations in the delegate.

The implementation of the Slice operations in a delegate servant is exactly the same as for a derived servant.

Delegate servants are useful if you need to derive your servant implementation from a base class in order to access some functionality. In that case, you cannot also derive the servant from the generated skeleton class. A delegate servant gets around Objective-C's single inheritance limitation and saves you having to write a servant class that forwards each operation invocation to the delegate.

Another use case are different interfaces that share their implementation. As an example, consider the following Slice definitions:

```
interface Intf1 {
    void op1();
};

interface Intf2 {
    void op2();
};
```

If `op1` and `op2` are substantially similar in their implementation and share common state, it can be convenient to implement the servants for `Intf1` and `Intf2` using a common delegate class:

```
@interface Intf1AndIntf2 : NSObject<EXIntf1, EXIntf2>
    +(id) intf1AndIntf2;
@end

@implementation Intf1AndIntf2
    +(id) intf1AndIntf2 { /*...*/ }
    -(void) op1:(ICECurrent*)current { /*...*/ }
    -(void) op2:(ICECurrent*)current { /*...*/ }
@end
```

See page 638 for an example of how to instantiate delegate servants.

Delegate servants do not permit you to override operations that are inherited from `ICEObject` (such as `ice_ping`). Therefore, if you want to override `ice_ping`, for example, to implement a default servant (see page 968), you must use a derived servant.

20.5 Parameter Passing

For each parameter of a Slice operation, the Objective-C mapping generates a corresponding parameter for the method in the skeleton. In addition, every method has an additional, trailing parameter of type `ICECurrent`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` method of the `Node` skeleton protocol has a single parameter of type `ICECurrent`. We explain the purpose of this parameter in Section 32.6 and will ignore it for now.

Parameter passing on the server side follows the rules for the client side (with one exception):

- In-parameters and the return value are passed by value or by pointer, depending on the parameter type.
- Out-parameters are passed by pointer-to-pointer.

The exception to the client-side rules concerns types that come in mutable and immutable variants (strings, sequences, and dictionaries). For these, the server-side mapping passes the mutable variant where the client-side passes the immutable variant, and vice versa.

To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

```
interface Intf {
    string op(string sin, out string sout);
};
```

The generated skeleton protocol for this interface looks as follows:

```
@protocol EXIntf <ICEObject>
-(NSString *) op:(NSMutableString *)sin
                 sout:(NSString **)sout
                 current:(ICECurrent *)current;
@end
```

As you can see, the in-parameter `sin` is of type `NSMutableString`, and out parameter and return value are passed as `NSString` (the opposite of the client-side mapping). This means that in-parameters are passed to the servant as their mutable variant, and it is safe for you to modify such in-parameters. This is useful, for example, if a client passes a sequence to the operation, and the operation returns the sequence with a few minor changes. In that case, there is no need for the operation implementation to copy the sequence. Instead, you can simply modify the passed sequence as necessary and return the modified sequence to the client.

Here is an example implementation of the operation:

```
-(NSString *) op:(NSMutableString *)sin
                 sout:(NSString **)sout
                 current:(ICECurrent *)current
{
    printf("%s\n", [sin UTF8String]); // In-params are initialized
    *sout = [sin appendString:@"appended"]; // Assign out-param
    return @"Done";                         // Return a string
}
```

Memory Management for Operations

To avoid leaking memory, you must be aware of how the Ice run time manages memory for operation implementations:

- In-parameters are passed to the servant already autoreleased.
- Out-parameters and return values must be returned by the servant as autoreleased values.

This follows the usual Objective-C convention: the allocator of a value is responsible for releasing it. This is what the Ice run time does for in-parameters, and what you are expected to do for out-parameters and return values. These rules also mean that it is OK to return an in-parameter as an out-parameter or return value. For example:

```
-(NSString *) op:(NSMutableString *)sin
                 sout:(NSString **)sout
                 current:(ICECurrent *)current
{
    *sout = sin; // Works fine.
    return sin;  // Works fine.
}
```

The Ice run time creates and releases a separate autorelease pool for each invocation. This means that the memory for parameters is reclaimed as soon as the run time has marshaled the operation results back to the client.

20.6 Raising Exceptions

To throw an exception from an operation implementation, you simply allocate the exception, initialize it, and throw it. For example:

```
-(void) write:(NSMutableArray *)text current:(ICECurrent *)current
{
    // Try to write the file contents here...
    // Assume we are out of space...
    if (error)
        @throw [FSGenericError genericError:@"file too large"];
}
```

As for out-parameters and return values, you must take care to throw an autoreleased exception.

If you throw an "impossible" user exception (a user exception that is not listed in the exception specification of the operation), the client receives an `UnknownUserException`.

If you throw a run-time exception, such as `MemoryLimitException`, the client receives an `UnknownLocalException`.[2] For that reason, you should never throw system exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

2. There are three system exceptions that are not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`. We discuss these exceptions in more detail in Section 4.10.4 and Chapter 33.

If you throw an arbitrary Objective-C exception that does not derive from `ICEException`, the client receives an `UnknownException`.

20.7 Object Incarnation

Having created a servant class such as the rudimentary `NodeI` class in Section 20.4.2, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must follow these steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

20.7.1 Instantiating a Servant

Instantiating a servant means to allocate an instance on the heap:

```
NodeI *servant = [NodeI nodei:@"Fred"];
```

This code creates a new `NodeI` instance. For this example, we used the convenience constructor we saw on page 633. Of course, you are not obliged to define such a constructor but, if you do not, you must explicitly call `release` or `autorelease` on the servant.

For the delegate servants we saw on page 634, instantiation would look as follows:

```
Intf1AndIntf2 *delegate = [Intf1AndIntf2 intf1AndIntf2];
ICEObject *servant = [ICEObject objectWithDelegate:delegate];
```

20.7.2 Creating an Identity

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.[3] An Ice object identity is a structure with the following Slice definition:

```
["objc:prefix:ICE"]
module Ice {
    struct Identity {
        string name;
        string category;
    };
    // ...
};
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (See Section 32.7 for a discussion of the `category` field.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

```
ICEIdentity ident = [ICEIdentity identity:"Fred" category:nil];
```

20.7.3 Activating a Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `adapter` variable, we can write:

```
[adapter add:servant identity:ident];
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.

3. The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity. See Chapter 34 for further discussion.

3. If a servant with that identity is active, the object adapter retrieves the servant pointer from the servant map and dispatches the incoming request into the correct method on the servant.

Assuming that the object adapter is in the active state (see Section 32.4.5), client requests are dispatched to the servant as soon as you call `add`.

Putting the preceding points together, we can write a simple method that instantiates and activates one of our `NodeI` servants. For this example, we use a simple method on our servant called `activate` that activates a servant in an object adapter with the passed identity:

```
-(void) activate:(id<ICEObjectAdapter>)a
                 name:(NSString *)name
{
    ICEIdentity ident = [ICEIdentity identity:name category:nil];
    [a add:self identity:ident];
}
```

20.7.4 UUIDs as Identities

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) [14] as identities. The `ICEUtil` class contains a helper function to create such identities:

```
@interface ICEUtil : NSObject
+(id) generateUUID;
// ...
@end
```

When executed, this method returns a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` creates a string that differs from all previous ones. You can use a UUID such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step:

```
-(id<ICEObjectPrx>) addWithUUID:(ICEObject*)servant
```

Note that the operation returns the proxy for the servant just activated.

20.7.5 Creating Proxies

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in page 68. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object, as we saw earlier. This means we can write:

```
NodeI *servant = [NodeI nodei:name];
id<FSNodePrx> proxy = [FSNodePrx uncheckedCast:
                        [adapter addWithUUID:servant]];

// Pass proxy to client...
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `id<ICEObjectPrx>`.

Direct Proxy Creation

The object adapter offers an operation to create a proxy for a given identity:

```
["objc:prefix:ICE"]
module Ice {
    local interface ObjectAdapter {
        Object* createProxy(Identity id);
        // ...
    };
};
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

```
ICEIdentity *ident = [ICEIdentity identity];
ident.name = [ICEUtil generateUUID];
id<ICEObjectPrx> o = [adapter createProxy:ident];
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Chapter 34.)

20.8 Summary

This chapter presented the server-side Objective-C mapping. Because the mapping for Slice data types is almost identical for clients and servers, the server-side mapping only adds a few additional mechanisms to the client side: a small API to initialize and finalize the run time, plus a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Even though the examples in this chapter are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers (which we discuss in Chapter 32), you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any Objective-C mapping rules beyond those we described here.

Chapter 21
Developing a File System Server in Objective-C

21.1 Chapter Overview

In this chapter, we present the source code for an Objective-C server that implements the file system we developed in Chapter 5 (see Chapter 19 for the corresponding client). The code we present here is fully functional, apart from the required interlocking for threads. (We examine threading issues in detail in Chapter 31.)

21.2 Implementing a File System Server

We have now seen enough of the server-side Objective-C mapping to implement a server for the file system we developed in Chapter 5. (You may find it useful to review the Slice definition for our file system in Section 5.4 before studying the source code.)

Our server is composed of three source files:

- `Server.m`

  This file contains the server main program.

- `FileI.m`

  This file contains the implementation for the `File` servants.

- `DirectoryI.m`

  This file contains the implementation for the `Directory` servants.

21.2.1 The Server `main` Program

Our server main program, in the file `Server.m`, uses the structure we saw in Section 20.3:

```
#import <Ice/Ice.h>
#import <FileI.h>
#import <DirectoryI.h>

#import <Foundation/NSAutoreleasePool.h>

int
main(int argc, char* argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int status = 1;
    id<ICECommunicator> communicator = nil;
    @try {
        communicator =
            [ICEUtil createCommunicator:&argc argv:argv];

        id<ICEObjectAdapter> adapter =
            [communicator createObjectAdapterWithEndpoints:
                                @"SimpleFilesystem"
                                endpoints:@"default -p 10000"];

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI *root =
            [DirectoryI directoryi:@"/" parent:nil];
        [root activate:adapter];

        // Create a file called "README" in the root directory
        //
        FileI *file = [FileI filei:@"README" parent:root];
        NSMutableArray *text = [NSMutableArray arrayWithObject:
            @"This file system contains a collection of poetry."];
        [file write:text current:nil];
        [file activate:adapter];

        // Create a directory called "Coleridge" in the root dir
```

```
        //
        DirectoryI *coleridge =
            [DirectoryI directoryi:@"Coleridge" parent:root];
        [coleridge activate:adapter];

        // Create a file called "Kubla_Khan"
        // in the Coleridge directory
        //
        file = [FileI filei:@"Kubla_Khan" parent:coleridge];
        text = [NSMutableArray arrayWithObjects:
                            @"In Xanadu did Kubla Khan",
                            @"A stately pleasure-dome decree:",
                            @"Where Alph, the sacred river, ran",
                            @"Through caverns measureless to man",
                            @"Down to a sunless sea.",
                            nil];
        [file write:text current:nil];
        [file activate:adapter];

        // All objects are created, allow client requests now
        //
        [adapter activate];

        // Wait until we are done
        //
        [communicator waitForShutdown];

        status = 0;
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    @try {
        [communicator destroy];
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    [pool release];
    return status;
}
```

There is quite a bit of code here, so let us examine each section in detail:

```
#import <Ice/Ice.h>
#import <FileI.h>
#import <DirectoryI.h>

#import <Foundation/NSAutoreleasePool.h>
```

The code includes the header `Ice/Ice.h`, which contains the definitions for the Ice run time, and the files `FileI.h` and `DirectoryI.h`, which contain the definitions of our servant implementations. Because we use an autorelease pool, we need to include `Foundation/NSAutoreleasePool.h` as well.

The next part of the source code is mostly boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`, which blocks the calling thread until you call `shutdown` or `destroy` on the communicator. (Ice does not make any demands on the main thread, so `waitForShutdown` simply blocks the calling thread; if you want to use the main thread for other purposes, you are free to do so.)

```
int
main(int argc, char* argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    int status = 1;
    id<ICECommunicator> communicator = nil;
    @try {
        communicator =
            [ICEUtil createCommunicator:&argc argv:argv];

        id<ICEObjectAdapter> adapter =
            [communicator createObjectAdapterWithEndpoints:
                                @"SimpleFilesystem"
                                endpoints:@"default -p 10000"];

        // ...

        // All objects are created, allow client requests now
        //
        [adapter activate];

        // Wait until we are done
        //
        [communicator waitForShutdown];

        status = 0;
    } @catch (NSException* ex) {
```

```
        NSLog(@"%@", ex);
    }

    @try {
        [communicator destroy];
    } @catch (NSException* ex) {
        NSLog(@"%@", ex);
    }

    [pool release];
    return status;
}
```

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown in Figure 21.1.

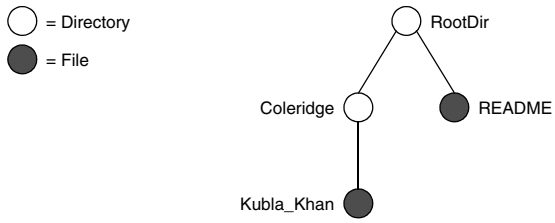


Figure 21.1. A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters: the name of the directory or file to be created and the servant for the parent directory. (For the root directory, which has no parent, we pass a `nil` parent.) Thus, the statement

```
DirectoryI *root = [DirectoryI directoryi:@"/" parent:nil];
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in Figure 21.1:

```
    // Create the root directory (with name "/" and no parent)
    //
    DirectoryI *root =
        [DirectoryI directoryi:@"/" parent:nil];
    [root activate:adapter];

    // Create a file called "README" in the root directory
```

```
//
FileI *file = [FileI filei:@"README" parent:root];
NSMutableArray *text = [NSMutableArray arrayWithObject:
    @"This file system contains a collection of poetry."];
[file write:text current:nil];
[file activate:adapter];

// Create a directory called "Coleridge" in the root dir
//
DirectoryI *coleridge =
    [DirectoryI directoryi:@"Coleridge" parent:root];
[coleridge activate:adapter];

// Create a file called "Kubla_Khan"
// in the Coleridge directory
//
file = [FileI filei:@"Kubla_Khan" parent:coleridge];
text = [NSMutableArray arrayWithObjects:
                    @"In Xanadu did Kubla Khan",
                    @"A stately pleasure-dome decree:",
                    @"Where Alph, the sacred river, ran",
                    @"Through caverns measureless to man",
                    @"Down to a sunless sea.",
                    nil];
[file write:text current:nil];
[file activate:adapter];
```

We first create the root directory and a file `README` within the root directory. (Note that we pass the servant for the root directory as the parent pointer when we create the new node of type `FileI`.)

After creating each servant, the code calls `activate` on the servant. (We will see the definition of this member function shortly.) The `activate` member function adds the servant to the ASM.

The next step is to fill the file with text:

```
// Create the root directory (with name "/" and no parent)
//
DirectoryI *root =
    [DirectoryI directoryi:@"/" parent:nil];
[root activate:adapter];

// Create a file called "README" in the root directory
//
FileI *file = [FileI filei:@"README" parent:root];
NSMutableArray *text = [NSMutableArray arrayWithObject:
```

```
    @"This file system contains a collection of poetry."];
[file write:text current:nil];
[file activate:adapter];

// Create a directory called "Coleridge" in the root dir
//
DirectoryI *coleridge =
    [DirectoryI directoryi:@"Coleridge" parent:root];
[coleridge activate:adapter];

// Create a file called "Kubla_Khan"
// in the Coleridge directory
//
file = [FileI filei:@"Kubla_Khan" parent:coleridge];
text = [NSMutableArray arrayWithObjects:
                    @"In Xanadu did Kubla Khan",
                    @"A stately pleasure-dome decree:",
                    @"Where Alph, the sacred river, ran",
                    @"Through caverns measureless to man",
                    @"Down to a sunless sea.",
                    nil];
[file write:text current:nil];
[file activate:adapter];
```

Recall from Section 18.8.3 that Slice string sequences map to `NSArray` or `NSMutableArray`, depending on the parameter direction. Here, we instantiate that array and add a line of text to it.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

```
[file write:text current:nil];
```

This code is interesting: the server code invokes an operation on one of its own servants. Because the call happens via the pointer to the servant (of type `FileI`) and *not* via a proxy (of type `id<FilePrx>`), the Ice run time does not know that this call is even taking place—such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary Objective-C function call. The operation implementation in the servant expects a `current` object. In this case, we pass nil (which is fine because the operation implementation does not use it anyway).

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in Figure 21.1.

21.2.2 The Servant Class Definitions

We must provide servants for the concrete interfaces in our Slice specification, that is, we must provide servants for the `File` and `Directory` interfaces in the Objective-C classes `FileI` and `DirectoryI`. This means that our servant classes look as follows:

```
#import <Filesystem.h>

@interface FileI : FSFile <FSFile>
// ...
@end

@interface DirectoryI : FSDirectory <FSDirectory>
// ...
@end
```

Each servant class derives from its skeleton class and adopts its skeleton protocol.

We now can think about how to implement our servants. One thing that is common to all nodes is that they have a name and a parent directory. As we saw earlier, we pass these details to a convenience constructor, which also takes care of calling `autorelease` on the new servant.

In addition, we will use UUIDs as the object identities for files and directories. This relieves us of the need to otherwise come up with a unique identity for each servant (such as path names, which would only complicate our implementation). Because the `list` operation returns proxies to nodes, and because each proxy carries the identity of the servant it denotes, this means that our servants must store their own identity, so we can create proxies to them when clients ask for them.

For `File` servants, we also need to store the contents of the file, leading to the following definition for the `FileI` class:

```
#import <Filesystem.h>

@class DirectoryI;

@interface FileI : FSFile <FSFile>
{
    @private
        NSString *myName;
        DirectoryI *parent;
        ICEIdentity *ident;
        NSArray *lines;
}
```

```
@property(nonatomic, retain) NSString *myName;
@property(nonatomic, retain) DirectoryI *parent;
@property(nonatomic, retain) ICEIdentity *ident;
@property(nonatomic, retain) NSArray *lines;

+(id) filei:(NSString *)name parent:(DirectoryI *)parent;
-(void) write:(NSMutableArray *)text
              current:(ICECurrent *)current;
-(void) activate:(id<ICEObjectAdapter>)a;
@end
```

The instance variables store the name, parent node, identity, and the contents of the file. The `filei` convenience constructor instantiates the servant, remembers the name and parent directory, assigns a new identity, and calls `autorelease`.

Note that the only Slice operation we have defined here is the `write` method. This is necessary because, as we saw previously, the code in `Server.m` calls this method to initialize the files it creates.

For directories, the requirements are similar. They also need to store a name, parent directory, and object identity. Directories are also responsible for keeping track of the child nodes. We can store these nodes in an array of proxies. This leads to the following definition:

```
#import <Filesystem.h>

@interface DirectoryI : FSDirectory <FSDirectory>
{
    @private
        NSString *myName;
        DirectoryI *parent;
        ICEIdentity *ident;
        NSMutableArray *contents;
}
@property(nonatomic, retain) NSString *myName;
@property(nonatomic, retain) DirectoryI *parent;
@property(nonatomic, retain) ICEIdentity *ident;
@property(nonatomic, retain) NSMutableArray *contents;

+(id) directoryi:(NSString *)name parent:(DirectoryI *)parent;
-(void) addChild:(id<FSNodePrx>)child;
-(void) activate:(id<ICEObjectAdapter>)a;
@end
```

Because the code in `Server.m` does not call any Slice operations on directory servants, we have not declared any of the corresponding methods. (We will see the

purpose of the `addChild` method shortly.) As for files, the convenience constructor creates the servant, remembers the name and parent, and assigns an object identity, as well as calling `autorelease`.

21.2.3 The Servant Implementation

Let us now turn to how to implement each of the methods for our servants.

Implementing `FileI`

The implementation of the `name`, `read`, and `write` operations for files is trivial, returning or updating the corresponding instance variable:

```
-(NSString *) name:(ICECurrent *)current
{
    return myName;
}

-(NSArray *) read:(ICECurrent *)current
{
    return lines;
}

-(void) write:(NSMutableArray *)text current:(ICECurrent *)current
{
    self.lines = text;
}
```

Note that this constitutes the complete implementation of the Slice operations for files.

Here is the convenience constructor:

```
+(id) filei:(NSString *)name parent:(DirectoryI *)parent
{
    FileI *instance = [[[FileI alloc] init] autorelease];
    if(instance == nil)
    {
        return nil;
    }
    instance.myName = name;
    instance.parent = parent;
    instance.ident = [ICEIdentity
        identity:[ICEUtil generateUUID] category:nil];
    return instance;
}
```

After allocating and autoreleasing the instance, the constructor initializes the instance variables. The only interesting part of this code is how we create the identity for the servant. `generateUUID` is a class method of the `ICEUtil` class that returns a UUID. We assign this UUID to the `name` member of the identity. (The `category` is unused—see Section 32.5 for more information.)

We saw earlier that the server calls `activate` after it creates each servant. Here is the implementation of this method:

```
-(void) activate:(id<ICEObjectAdapter>)a
{
    id<FSNodePrx> thisNode =
        [FSNodePrx uncheckedCast:[a add:self identity:ident]];
    [parent addChild:thisNode];
}
```

This is how our code informs the Ice run time of the existence of a new servant. The call to `add` on the object adapter adds the servant and object identity to the adapter's servant map. In other words, this step creates the link between the object identity (which is embedded in proxies), and the actual Objective-C class instance that provides the behavior for the Slice operations.

`add` returns a proxy to the servant, of type `id<ICEObjectPrx>`. Because the `contents` instance variable of directory servants stores proxies of type `id<FSNodePrx>` (and `addChild` expects a proxy of that type), we down-cast the returned proxy to `id<FSNodePrx>`. In this case, because we know that the servant we just added to the adapter is indeed a servant that implements the operations on the Slice `Node` interface, we can use an `uncheckedCast`.

The call to `addChild` connects the new file to its parent directory.

Finally, we need a `dealloc` function so we do not leak the memory for the servant's instance variables:

```
-(void) dealloc
{
    [myName release];
    [parent release];
    [ident release];
    [lines release];
    [super dealloc];
}
```

Implementing `DirectoryI`

The implementation of the Slice operations for directories is just as simple as for files:

```
-(NSString *) name:(ICECurrent *)current
{
    return myName;
}

-(NSArray *) list:(ICECurrent *)current
{
    return contents;
}
```

Because the `contents` instance variable stores the proxies for child nodes of the directory, the list operation simply returns that variable.

The convenience constructor looks much like the one for file servants:

```
+(id) directoryi:(NSString *)name parent:(DirectoryI *)parent
{
    DirectoryI *instance =
        [[[DirectoryI alloc] init] autorelease];
    if(instance == nil)
    {
        return nil;
    }
    instance.myName = name;
    instance.parent = parent;
    instance.ident = [ICEIdentity
        identity:(parent ? [ICEUtil generateUUID] : @"RootDir")
        category:nil];
    instance.contents = [[NSMutableArray alloc] init];
    return instance;
}
```

The only noteworthy differences are that, for the root directory (which has no parent), the code uses `"RootDir"` as the identity. (As we saw on page 623, the client knows that this is the identity of the root directory and uses it to create its proxy.)

The `addChild` method connects our nodes into a hierarchy by updating the `contents` instance variable. That way, each directory knows which nodes are contained in it:

```
-(void) addChild:(id<FSNodePrx>)child
{
     [contents addObject:child];
}
```

Finally, the `activate` and `dealloc` methods are very much like the corresponding methods for files:

```
-(void) activate:(id<ICEObjectAdapter>)a
{
    id<FSNodePrx> thisNode
        = [FSNodePrx uncheckedCast:[a add:self identity:ident]];
    [parent addChild:thisNode];
}

-(void) dealloc
{
    [myName release];
    [parent release];
    [ident release];
    [contents release];
    [super dealloc];
}
```

21.3 Summary

This chapter showed how to implement a complete server for the file system we defined in Chapter 5. Note that the server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same for a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

Note that the server code we presented here is not quite correct as it stands: if two clients access the same file in parallel, each via a different thread, one thread may read the `lines` instance variable while another thread updates it. Obviously, if that happens, we may write or return garbage or, worse, crash the server. However, it is trivial to make the `read` and `write` operations thread-safe with a few lines of code. We discuss how to write thread-safe servant implementations in Chapter 31.

Part VII

Python Mapping

Chapter 22
Client-Side Slice-to-Python Mapping

22.1 Chapter Overview

In this chapter, we present the client-side Slice-to-Python mapping (see Chapter 24 for the server-side mapping). One part of the client-side Python mapping concerns itself with rules for representing each Slice data type as a corresponding Python type; we cover these rules in Section 22.3 to Section 22.10. Another part of the mapping deals with how clients can invoke operations, pass and receive parameters, and handle exceptions. These topics are covered in Section 22.11 to Section 22.13. Slice classes have the characteristics of both data types and interfaces and are covered in Section 22.14. Section 22.15 discusses asynchronous invocations. Code generation issues are discussed in Section 22.16, while Section 22.17 addresses the use of Slice checksums.

22.2 Introduction

The client-side Slice-to-Python mapping defines how Slice data types are translated to Python types, and how clients invoke operations, pass parameters, and handle errors. Much of the Python mapping is intuitive. For example, Slice sequences map to Python lists, so there is essentially nothing new you have to learn in order to use Slice sequences in Python.

The Python API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data—the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least Section 22.11 to Section 22.13 in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

A word of advice before you start: in order to use the Python mapping, you should need no more than the Slice definition of your application and knowledge of the Python mapping rules. In particular, looking through the generated code in order to discern how to use the Python mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

22.3 Mapping for Identifiers

Slice identifiers map to an identical Python identifier. For example, the Slice identifier `Clock` becomes the Python identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a Python keyword or is an identifier reserved by the Ice run time (such as `checkedCast`), the corresponding Python identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.[1]

The mapping does not modify a Slice identifier that matches the name of a Python built-in function because it can always be accessed by its fully-qualified name. For example, the built-in function `hash` can also be accessed as `__builtin__.hash`.

1. As suggested in Section 4.5.3 on page 92, you should try to avoid such identifiers as much as possible.

22.4 Mapping for Modules

Slice modules map to Python modules with the same name as the Slice module. The mapping preserves the nesting of the Slice definitions. See Section 22.16.2 for information about the mapping's use of Python packages.

22.5 The `Ice` Module

All of the APIs for the Ice run time are nested in the `Ice` module, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` module are generated from Slice definitions; other parts of the `Ice` module provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` module throughout the remainder of the book.

A Python application can load the Ice run time using the `import` statement:

```
import Ice
```

If the statement executes without error, the Ice run time is loaded and available for use. You can determine the version of the Ice run time you have just loaded by calling the `stringVersion` function:

```
icever = Ice.stringVersion()
```

22.6 Mapping for Simple Built-In Types

The Slice built-in types are mapped to Python types as shown in Table 22.1.

Table 22.1. Mapping of Slice built-in types to Python.

Slice	Python
`bool`	`bool`
`byte`	`int`
`short`	`int`

Table 22.1. Mapping of Slice built-in types to Python.

Slice	Python
`int`	`int`
`long`	`long`
`float`	`double`
`double`	`double`
`string`	`string`

Although Python supports arbitrary precision in its integer types, the Ice run time validates integer values to ensure they have valid ranges for their declared Slice types.

22.6.1 String Mapping

String values returned as the result of a Slice operation (including return values, out parameters, and data members) are always represented as instances of Python's 8-bit `string` type. These string values contain UTF-8 encoded strings unless the program has installed a string converter, in which case string values use the converter's native encoding instead. See Section 32.24 for more information on string converters.

Legal string input values for a remote Slice operation are shown below:

- `None`

  Ice marshals an empty string whenever `None` is encountered.

- 8-bit string objects

  Ice assumes that all 8-bit string objects contain valid UTF-8 encoded strings unless the program has installed a string converter, in which case Ice assumes that 8-bit string objects use the native encoding expected by the converter.

- Unicode objects

  Ice converts a Unicode object into UTF-8 and marshals it directly. If a string converter is installed, it is not invoked for Unicode objects.

22.7 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

22.7.1 Mapping for Enumerations

Python does not have an enumerated type, so the Slice enumerations are emulated using a Python class: the name of the Slice enumeration becomes the name of the Python class; for each enumerator, the class contains an attribute with the same name as the enumerator. For example:

```
enum Fruit { Apple, Pear, Orange };
```

The generated Python class looks as follows:

```
class Fruit(object):
    def __init__(self, val):
        assert(val >= 0 and val < 3)
        self.value = val

    # ...

Fruit.Apple = Fruit(0)
Fruit.Pear = Fruit(1)
Fruit.Orange = Fruit(2)
```

Each instance of the class has a `value` attribute providing the integer value of the enumerator. Note that the generated class also defines a number of Python special methods, such as `__str__` and `__cmp__`, which we have not shown.

Given the above definitions, we can use enumerated values as follows:

```
f1 = Fruit.Apple
f2 = Fruit.Orange

if f1 == Fruit.Apple:                # Compare with constant
    # ...

if f1 == f2:                         # Compare two enums
    # ...

if f2.value == Fruit.Apple.value:    # Use integer values
    # ...
```

```
elif f2.value == Fruit.Pear.value:
    # ...
elif f2.value == Fruit.Orange.value:
    # ...
```

As you can see, the generated class enables natural use of enumerated values. The `Fruit` class attributes are preinitialized enumerators that you can use for initialization and comparison. You may also instantiate an enumerator explicitly by passing its integer value to the constructor, but you must make sure that the passed value is within the range of the enumeration; failure to do so will result in an assertion failure:

```
favoriteFruit = Fruit(4) # Assertion failure!
```

22.7.2 Mapping for Structures

Slice structures map to Python classes with the same name. For each Slice data member, the Python class contains a corresponding attribute. For example, here is our `Employee` structure from Section 4.9.4 once more:

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The Python mapping generates the following definition for this structure:

```
class Employee(object):
    def __init__(self, number=0, firstName='', lastName=''):
        self.number = number
        self.firstName = firstName
        self.lastName = lastName

    def __hash__(self):
        # ...

    def __eq__(self, other):
        # ...

    def __str__(self):
        # ...
```

The constructor initializes each of the attributes to a default value appropriate for its type. You can also declare different default values for members of primitive and enumerated types, as discussed in Section 4.9.2.

The `__hash__` method returns a hash value for the structure based on the value of all its data members.

The `__eq__` method returns true if all members of two structures are (recursively) equal.

The `__str__` method returns a string representation of the structure.

22.7.3 Mapping for Sequences

Slice sequences map by default to Python lists; the only exception is a sequence of bytes, which maps by default to a string in order to lower memory utilization and improve throughput. This use of native types means that the Python mapping does not generate a separate named type for a Slice sequence. It also means that you can take advantage of all the inherent functionality offered by Python's native types. For example:

```
sequence<Fruit> FruitPlatter;
```

We can use the `FruitPlatter` sequence as shown below:

```
platter = [ Fruit.Apple, Fruit.Pear ]
assert(len(platter) == 2)
platter.append(Fruit.Orange)
```

The Ice run time validates the elements of a tuple or list to ensure that they are compatible with the declared type; a `ValueError` exception is raised if an incompatible type is encountered.

Allowable Sequence Values

Although each sequence type has a default mapping, the Ice run time allows a sender to use other types as well. Specifically, a tuple is also accepted for a sequence type that maps to a list, and in the case of a byte sequence, the sender is allowed to supply a tuple or list of integers as an alternative to a string[2].

2. Using a string for a byte sequence bypasses the validation step and avoids an extra copy, resulting in much greater throughput than a tuple or list. For larger byte sequences, the use of a string is strongly recommended.

Furthermore, the Ice run time accepts objects that implement Python's buffer protocol as legal values for sequences of all primitive types except strings. For example, you can use the `array` module to create a buffer that is transferred much more efficiently than a tuple or list. Consider the two sequence values in the sample code below:

```
import array
...
seq1 = array.array("i", [1, 2, 3, 4, 5])
seq2 = [1, 2, 3, 4, 5]
```

The values have the same on-the-wire representation, but they differ greatly in marshaling overhead because the buffer can be traversed more quickly and requires no validation.

Note that the Ice run time has no way of knowing what type of elements a buffer contains, therefore it is the application's responsibility to ensure that a buffer is compatible with the declared sequence type.

Customizing the Sequence Mapping

The previous section described the allowable types that an application may use when sending a sequence. That kind of flexibility is not possible when receiving a sequence, because in this case it is the Ice run time's responsibility to create the container that holds the sequence.

As stated earlier, the default mapping for most sequence types is a list, and for byte sequences the default mapping is a string. Unless otherwise indicated, an application always receives sequences as the container type specified by the default mapping. If it would be more convenient to receive a sequence as a different type, you can customize the mapping by annotating your Slice definitions with metadata. Table 22.2 describes the metadata directives supported by the Python mapping.

Table 22.2. Custom metadata directives for the sequence mapping.

Directive	Description
`python:seq:default`	Use the default mapping.
`python:seq:list`	Map to a Python list.

Table 22.2. Custom metadata directives for the sequence mapping.

Directive	Description
`python:seq:tuple`	Map to a Python tuple.

A metadata directive may be specified when defining a sequence, or when a sequence is used as a parameter, return value or data member. If specified at the point of definition, the directive affects all occurrences of that sequence type unless overridden by another directive at a point of use. The following Slice definitions illustrate these points:

```
sequence<int> IntList; // Uses list by default
["python:seq:tuple"] sequence<int> IntTuple; // Defaults to tuple

sequence<byte> ByteString; // Uses string by default
["python:seq:list"] sequence<byte> ByteList; // Defaults to list

struct S {
    IntList i1; // list
    IntTuple i2; // tuple
    ["python:seq:tuple"] IntList i3; // tuple
    ["python:seq:list"] IntTuple i4; // list
    ["python:seq:default"] IntTuple i5; // list

    ByteString b1; // string
    ByteList b2; // list
    ["python:seq:list"] ByteString b3; // list
    ["python:seq:tuple"] ByteString b4; // tuple
    ["python:seq:default"] ByteList b5; // string
};

interface I {
    IntList op1(ByteString s1, out ByteList s2);

    ["python:seq:tuple"]
    IntList op2(["python:seq:list"] ByteString s1,
                ["python:seq:tuple"] out ByteList s2);
};
```

The operation `op2` and the data members of structure `S` demonstrate how to override the mapping for a sequence at the point of use.

It is important to remember that these metadata directives only affect the receiver of the sequence. For example, the data members of structure `S` are populated with the specified sequence types only when the Ice run time unmarshals an instance of `S`. In the case of an operation, custom metadata affects the client when specified for the operation's return type and output parameters, whereas metadata affects the server for input parameters.

22.7.4 Mapping for Dictionaries

Here is the definition of our `EmployeeMap` from Section 4.9.4 once more:

```
dictionary<long, Employee> EmployeeMap;
```

As for sequences, the Python mapping does not create a separate named type for this definition. Instead, *all* dictionaries are simply instances of Python's dictionary type. For example:

```
em = {}

e = Employee()
e.number = 31
e.firstName = "James"
e.lastName = "Gosling"

em[e.number] = e
```

The Ice run time validates the elements of a dictionary to ensure that they are compatible with the declared type; a `ValueError` exception is raised if an incompatible type is encountered.

22.8 Mapping for Constants

Here are the constant definitions we saw in Section 4.9.5 on page 103 once more:

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

The generated definitions for these constants are shown below:

```
AppendByDefault = True
LowerNibble = 15
Advice = "Don't Panic!"
TheAnswer = 42
PI = 3.1416
FavoriteFruit = Fruit.Pear
```

As you can see, each Slice constant is mapped to a Python attribute with the same name as the constant.

22.9 Mapping for Exceptions

The mapping for exceptions is based on the inheritance hierarchy shown in Figure 22.1

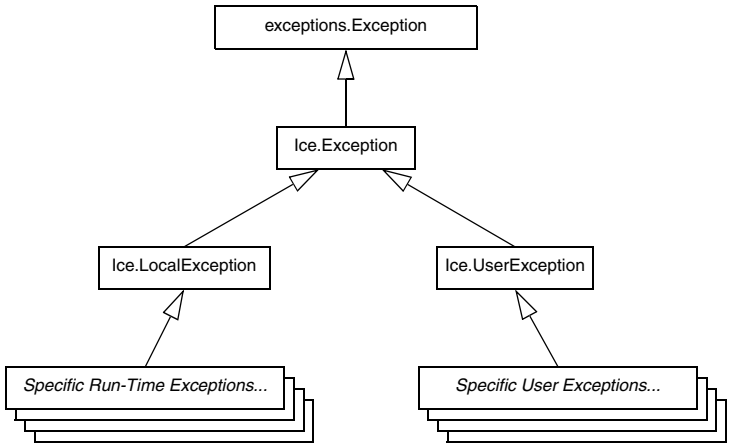


Figure 22.1. Inheritance structure for Ice exceptions.

The ancestor of all exceptions is `exceptions.Exception`, from which `Ice.Exception` is derived. `Ice.LocalException` and `Ice.UserException` are derived from `Ice.Exception` and form the base for all run-time and user exceptions.

Here is a fragment of the Slice definition for our world time server from Section 4.10.5 on page 120 once more:

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

These exception definitions map as follows:

```
class GenericError(Ice.UserException):
    def __init__(self, reason=''):
        self.reason = reason

    def ice_name(self):
        # ...

    def __str__(self):
        # ...

class BadTimeVal(GenericError):
    def __init__(self, reason=''):
        GenericError.__init__(self, reason)

    def ice_name(self):
        # ...

    def __str__(self):
        # ...

class BadZoneName(GenericError):
    def __init__(self, reason=''):
        GenericError.__init__(self, reason)

    def ice_name(self):
        # ...

    def __str__(self):
        # ...
```

Each Slice exception is mapped to a Python class with the same name. The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception member corresponds to an attribute of the instance, which the constructor initializes to a default value appropriate for its type. You can also declare different default values for members of primitive and enumerated types, as discussed in Section 4.10.2. Although `BadTimeVal` and `BadZoneName` do not

declare data members, their constructors still accept a value for the inherited data member `reason` in order to pass it to the constructor of the base exception `GenericError`.

Each exception also defines the `ice_name` method to return the name of the exception, and the special method `__str__` to return a stringified representation of the exception and its members.

All user exceptions are derived from the base class `Ice.UserException`. This allows you to catch all user exceptions generically by installing a handler for `Ice.UserException`. Similarly, you can catch all Ice run-time exceptions with a handler for `Ice.LocalException`, and you can catch all Ice exceptions with a handler for `Ice.Exception`.

22.10 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice.LocalException` (which, in turn, derives from `Ice.Exception`).

An inheritance diagram for user and run-time exceptions appears in Figure 4.4 on page 117. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice.LocalException`

  This is the root of the inheritance tree for run-time exceptions.

- `Ice.UserException`

  This is the root of the inheritance tree for user exceptions.

- `Ice.TimeoutException`

  This is the base exception for both operation-invocation and connection-establishment timeouts.

- `Ice.ConnectTimeoutException`

  This exception is raised when the initial attempt to establish a connection to a server times out.

You will probably have little need to catch the remaining run-time exceptions; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. However, there is one exception you will probably be interested in specifically: `Ice.ObjectNotExistException`. This exception is raised if a client invokes an operation on an Ice

object that no longer exists. In other words, the client holds a dangling reference to an object that probably existed some time in the past but has since been permanently destroyed.

22.11 Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a method on a local class instance that represents the remote object. This makes the mapping easy and intuitive to use because, for all intents and purposes (apart from error semantics), making a remote procedure call is no different from making a local procedure call.

22.11.1 Proxy Classes

On the client side, Slice interfaces map to Python classes with methods that correspond to the operations on those interfaces. Consider the following simple interface:

```
interface Simple {
    void op();
};
```

The Python mapping generates the following definition for use by the client:

```
class SimplePrx(Ice.ObjectPrx):
    def op(self, _ctx=None):
        # ...

    # ...
```

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice.ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a method of the same name. In the preceding example, we find that the operation `op` has been mapped to the method `op`. Note that `op` accepts an optional trailing parameter `_ctx` representing the operation context. This parameter is a Python dictionary for use by the

Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the context parameter in detail in Chapter 32. The parameter is also used by IceStorm—see Chapter 44.)

Proxy instances are always created on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

A value of `None` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

22.11.2 The `Ice.ObjectPrx` Class

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice.ObjectPrx`. `ObjectPrx` provides a number of methods:

```
class ObjectPrx(object):
    def equals(self, other):
    def ice_getIdentity(self):
    def ice_isA(self, id):
    def ice_id(self):
    def ice_ping(self):
    # ...
```

The methods behave as follows:

- `equals`

  This operation compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

- `ice_getIdentity`

  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  ```
  module Ice {
      struct Identity {
          string name;
          string category;
      };
  };
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

```
proxy1 = ...
proxy2 = ...
id1 = proxy1.ice_getIdentity()
id2 = proxy2.ice_getIdentity()

if id1 == id2:
    # proxy1 and proxy2 denote the same object
else:
    # proxy1 and proxy2 denote different objects
```

- `ice_isA`

  This method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID (see Section 4.13). For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

  ```
  proxy = ...
  if proxy != None and proxy.ice_isA("::Printer"):
      # proxy denotes a Printer object
  else:
      # proxy denotes some other type of object
  ```

  Note that we are testing whether the proxy is `None` before attempting to invoke the `ice_isA` method. This avoids getting a run-time error if the proxy is `None`.

- `ice_id`

  This method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type `BasePrx`, with a static type ID of `::Base`, the return value of `ice_id` might be `"::Base"`, or it might be something more derived, such as `"::Derived"`.

- `ice_ping`

  This method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as `ObjectNotExistException` or `ConnectTimeoutException`.

Note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call. (We discuss these methods in Chapter 32.)

22.11.3 Casting Proxies

The Python mapping for a proxy also generates two static methods:

```
class SimplePrx(Ice.ObjectPrx):
    # ...

    def checkedCast(proxy, facet=''):
        # ...
    checkedCast = staticmethod(checkedCast)

    def uncheckedCast(proxy, facet=''):
        # ...
    uncheckedCast = staticmethod(uncheckedCast)
```

Both the `checkedCast` and `uncheckedCast` methods implement a downcast: if the passed proxy is a proxy for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is `None`), the cast returns `None`.

The method names `checkedCast` and `uncheckedCast` are reserved for use in proxies. If a Slice interface defines an operation with either of those names, the mapping escapes the name in the generated proxy by prepending an underscore. For example, an interface that defines an operation named `checkedCast` is mapped to a proxy with a method named `_checkedCast`.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

```
obj = ...       # Get a proxy from somewhere...

simple = SimplePrx.checkedCast(obj)
if simple != None:
    # Object supports the Simple interface...
else:
    # Object is not of type Simple...
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of a proxy in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`.

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a

run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather non-sensical things. To illustrate this, consider the following two interfaces:

```
interface Process {
    void launch(int stackSize, int dataSize);
};

// ...

interface Rocket {
    void launch(float xCoord, float yCoord);
};
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

```
obj = ...                                # Get proxy...
process = ProcessPrx.uncheckedCast(obj) # No worries...
process.launch(40, 60)                  # Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

22.11.4 Using Proxy Methods

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy (see Section 32.11). Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modifica-

tion. For example, you can obtain a proxy configured with a ten second timeout as shown below:

```
proxy = communicator.stringToProxy(...)
proxy = proxy.ice_timeout(10000)
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method. The example below demonstrates these semantics:

```
base = communicator.stringToProxy(...)
hello = Demo.HelloPrx.checkedCast(base)
hello = hello.ice_timeout(10000) # Type is preserved
hello.sayHello()
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

22.11.5 Object Identity and Proxy Comparison

Proxy objects support comparison using the built-in relational operators as well as the `cmp` function. Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if p1 != p2:
    # p1 and p2 denote different objects       # WRONG!
else:
    # p1 and p2 denote the same object         # Correct
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each uses a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can

be contacted via several different transport endpoints). In other words, if two proxies compare equal, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function in the `Ice` module:

```
def proxyIdentityCompare(lhs, rhs)
def proxyIdentityAndFacetCompare(lhs, rhs)
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if Ice.proxyIdentityCompare(p1, p2) != 0:
    # p1 and p2 denote different objects       # Correct
else:
    # p1 and p2 denote the same object         # Correct
```

The function returns 0 if the identities are equal, –1 if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major sort key and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name (see Chapter 33).

22.12 Mapping for Operations

As we saw in Section 22.11, for each operation on an interface, the proxy class contains a corresponding method with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system from Section 5.4:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

```
node = ...          # Initialize proxy
name = node.name()  # Get name via RPC
```

22.12.1 Normal and `idempotent` Operations

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

```
interface Example {
                string op1();
    idempotent  string op2();
};
```

The proxy class for this is:

```
class ExamplePrx(Ice.ObjectPrx):
    def op1(self, _ctx=None):
        # ...

    def op2(self, _ctx=None):
        # ...
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to look the same.

22.12.2 Passing Parameters

In Parameters

All parameters are passed by reference in the Python mapping; it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;
```

```
dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following proxy for this definition:

```
class ClientToServerPrx(Ice.ObjectPrx):
    def op1(self, i, f, b, s, _ctx=None):
        # ...

    def op2(self, ns, ss, st, _ctx=None):
        # ...

    def op3(self, proxy, _ctx=None):
        # ...
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

```
p = ...                                  # Get proxy...

p.op1(42, 3.14f, True, "Hello world!")  # Pass simple literals

i = 42
f = 3.14f
b = True
s = "Hello world!"
p.op1(i, f, b, s)                        # Pass simple variables

ns = NumberAndString()
ns.x = 42
ns.str = "The Answer"
ss = [ "Hello world!" ]
st = {}
st[0] = ns
p.op2(ns, ss, st)                        # Pass complex variables

p.op3(p)                                 # Pass proxy
```

Out Parameters

As in Java, Python functions do not support reference arguments. That is, it is not possible to pass an uninitialized variable to a Python function in order to have its

value initialized by the function. The Java mapping (see Section 10.12.2) overcomes this limitation with the use of "holder classes" that represent each `out` parameter. The Python mapping takes a different approach, one that is more natural for Python users.

The semantics of `out` parameters in the Python mapping depend on whether the operation returns one value or multiple values. An operation returns multiple values when it has declared multiple `out` parameters, or when it has declared a non-`void` return type and at least one `out` parameter.

If an operation returns multiple values, the client receives them in the form of a *result tuple*. A non-`void` return value, if any, is always the first element in the result tuple, followed by the `out` parameters in the order of declaration.

If an operation returns only one value, the client receives the value itself.

Here is the same Slice definition we saw on page 679 once more, but this time with all parameters being passed in the `out` direction:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    int op1(out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Python mapping generates the following code for this definition:

```
class ServerToClientPrx(Ice.ObjectPrx):
    def op1(self, _ctx=None):
        # ...

    def op2(self, _ctx=None):
        # ...

    def op3(self, _ctx=None):
        # ...
```

Given a proxy to a `ServerToClient` interface, the client code can receive the results as in the following example:

```
p = ...                # Get proxy...
i, f, b, s = p.op1()
ns, ss, st = p.op2()
stcp = p.op3()
```

The operations have no `in` parameters, therefore no arguments are passed to the proxy methods. Since `op1` and `op2` return multiple values, their result tuples are unpacked into separate values, whereas the return value of `op3` requires no unpacking.

Parameter Type Mismatches

Although the Python compiler cannot check the types of arguments passed to a function, the Ice run time does perform validation on the arguments to a proxy invocation and reports any type mismatches as a `ValueError` exception.

Null Parameters

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `None`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `None` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid a run-time error. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, it makes no difference to the receiver whether you send a string as `None` or as an empty string: either way, the receiver sees an empty string.

22.13 Exception Handling

Any operation invocation may throw a run-time exception (see Section 22.10) and, if the operation has an exception specification, may also throw user exceptions (see Section 22.9). Suppose we have the following simple interface:

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as Python exceptions, so you can simply enclose one or more operation invocations in a `try–except` block:

```
child = ...        # Get child proxy...

try:
    child.askToCleanUp()
except Tantrum, t:
    print "The child says:", t.reason
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will usually be handled by exception handlers higher in the hierarchy. For example:

```
import traceback, Ice

def run():
    child = ...        # Get child proxy...
    try:
        child.askToCleanUp()
    except Tantrum, t:
        print "The child says:", t.reason
        child.scold()  # Recover from error...
    child.praise()     # Give positive feedback...

try:
    # ...
    run()
    # ...
except Ice.Exception:
    traceback.print_exc()
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Chapter 3.)

22.14 Mapping for Classes

Slice classes are mapped to Python classes with the same name. The generated class contains an attribute for each Slice data member (just as for structures and exceptions). Consider the following class definition:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Python mapping generates the following code for this definition:

```
class TimeOfDay(Ice.Object):
    def __init__(self, hour=0, minute=0, second=0):
        # ...
        self.hour = hour
        self.minute = minute
        self.second = second

    def ice_staticId():
        return '::M::TimeOfDay'
    ice_staticId = staticmethod(ice_staticId)

    # ...

    #
    # Operation signatures.
    #
    # def format(self, current=None):
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice.Object`. This means that all classes implicitly inherit from `Ice.Object`, which is the ultimate ancestor of all classes. Note that `Ice.Object` is *not* the same as `Ice.ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.

2. The constructor defines an attribute for each Slice data member.
3. The class defines the static method `ice_staticId`.
4. A comment summarizes the method signatures for each Slice operation.

We will discuss these items in the subsections below.

22.14.1 Inheritance from `Ice.Object`

Like interfaces, classes implicitly inherit from a common base class, `Ice.Object`. However, as shown in Figure 22.2, classes inherit from `Ice.Object` instead of `Ice.ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

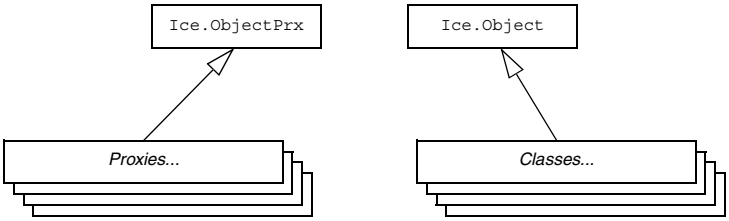


Figure 22.2. Inheritance from `Ice.ObjectPrx` and `Ice.Object`.

`Ice.Object` contains a number of member functions:

```
class Object(object):
    def ice_isA(self, id, current=None):
        # ...

    def ice_ping(self, current=None):
        # ...

    def ice_ids(self, current=None):
        # ...

    def ice_id(self, current=None):
        # ...

    def ice_staticId():
        # ...
    ice_staticId = staticmethod(ice_staticId)
```

```
def ice_preMarshal(self):
    # ...

def ice_postUnmarshal(self):
    # ...
```

The member functions of `Ice.Object` behave as follows:

- `ice_isA`

  This method returns `true` if the object supports the given type ID, and `false` otherwise.

- `ice_ping`

  As for interfaces, `ice_ping` provides a basic reachability test for the class.

- `ice_ids`

  This method returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.

- `ice_id`

  This method returns the actual run-time type ID of the object. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.

- `ice_staticId`

  This method is generated in each class and returns the static type ID of the class.

- `ice_preMarshal`

  The Ice run time invokes this method prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.

- `ice_postUnmarshal`

  The Ice run time invokes this method after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.

Note that neither `Ice.Object` nor the generated class override `__hash__` and `__eq__`, so the default implementations apply.

22.14.2 Data Members of Classes

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding attribute.

Although Python provides no standard mechanism for restricting access to an object's attributes, by convention an attribute whose name begins with an underscore signals the author's intent that the attribute should only be accessed by the class itself or by one of its subclasses. You can employ this convention in your Slice classes using the `protected` metadata directive. The presence of this directive causes the Slice compiler to prepend an underscore to the mapped name of the data member. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

```
class TimeOfDay {
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler produces the following generated code for this definition:

```
class TimeOfDay(Ice.Object):
    def __init__(self, hour=0, minute=0, second=0):
        # ...
        self._hour = hour
        self._minute = minute
        self._second = second

    # ...

    #
    # Operation signatures.
    #
    # def format(self, current=None):
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

```
["protected"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

22.14.3 Class Constructors

Classes have a constructor that assigns to each data member a default value appropriate for its type. You can also declare different default values for data members of primitive and enumerated types, as discussed in Section 4.11.1.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

22.14.4 Operations of Classes

Operations of classes are mapped to methods in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

```
class TimeOfDayI(TimeOfDay):
    def __init__(self, hour=0, minute=0, second=0):
        TimeOfDay.__init__(self, hour, minute, second)

    def format(self, current=None):
        return "%02d:%02d:%02d" %\
            (self.hour, self.minute, self.second)
```

A Slice class such as `TimeOfDay` that declares or inherits an operation is inherently abstract. Python does not support the notion of abstract classes or abstract methods, therefore the mapping merely summarizes the required method signatures in a comment for your convenience. Furthermore, the mapping generates code in the constructor of an abstract class to prevent it from being instantiated directly; any attempt to do so raises a `RuntimeError` exception.

You may notice that the mapping for an operation adds an optional trailing parameter named `current`. For now, you can ignore this parameter and pretend it does not exist. (We look at it in more detail in Section 32.6.)

22.14.5 Class Factories

Having created a class such as this, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

```
class ObjectFactory(Ice.ObjectFactory):
    def create(self, type):
        if type == M.TimeOfDay.ice_staticId():
            return TimeOfDayI()
        assert(False)
        return None
```

```
    def destroy(self):
        # Nothing to do
        pass
```

The object factory's `create` method is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` method is called by the Ice run time when its communicator is destroyed.

The `create` method is passed the type ID (see Section 4.13) of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::M::TimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `TimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoObjectFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

```
ic = ...   # Get Communicator...
ic.addObjectFactory(ObjectFactory(),
                    M.TimeOfDay.ice_staticId())
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::M::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator—if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not

be called once `destroy` has been called. However, calls to `create` can be made concurrently.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

22.15 Asynchronous Method Invocation (AMI)

NOTE: As of version 3.4, Ice provides a new API for asynchronous method invocation. This section describes this API. You can find documentation for the previous API in Appendix K. Note that the old API is deprecated and will be removed in a future release.

Asynchronous Method Invocation (*AMI*) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and poll or wait for completion of the invocation, or receive a callback when the invocation completes.

AMI is transparent to the server: there is no way for the server to tell whether a client sent a request synchronously or asynchronously.

22.15.1 Basic Asynchronous API

Consider the following simple Slice definition:

```
module Demo {
    interface Employees {
        string getName(int number);
    };
};
```

Proxy Methods

Besides the synchronous proxy methods, the Python mapping generates the following asynchronous proxy methods:

```
def begin_getName(self, number, _response=None, _ex=None,
                  _sent=None, _ctx=None)
def end_getName(self, result)
```

As you can see, the single `getName` operation results in `begin_getName` and `end_getName` methods. The `begin_` method optionally accepts a per-invocation context (see Section 32.12) and callbacks (see Section 22.15.4).

- The `begin_getName` method sends (or queues) an invocation of `getName`. This method does not block the calling thread.
- The `end_getName` method collects the result of the asynchronous invocation. If, at the time the calling thread calls `end_getName`, the result is not yet available, the calling thread blocks until the invocation completes. Otherwise, if the invocation completed some time before the call to `end_getName`, the method returns immediately with the result.

A client could call these methods as follows:

```
e = EmployeePrx.checkedCast(...)
r = e.begin_getName(99)

# Continue to do other things here...

name = e.end_getName(r)
```

Because `begin_getName` does not block, the calling thread can do other things while the operation is in progress.

Note that `begin_getName` returns a value of type `AsyncResult`. This value contains the state that the Ice run time requires to keep track of the asynchronous invocation. You must pass the `AsyncResult` that is returned by the `begin_` method to the corresponding `end_` method.

The `begin_` method has one parameter for each in-parameter of the corresponding Slice operation. The `end_` method accepts the `AsyncResult` object as its only argument and returns the out-parameters using the same semantics as for regular synchronous invocations (see page 680). For example, consider the following operation:

```
double op(int inp1, string inp2, out bool outp1, out long outp2);
```

The `begin_op` and `end_op` methods have the following signature:

```
def begin_op(self, inp1, inp2, ...)
def end_op(self, result)
```

The call to `end_op` returns the following tuple:

```
doubleValue, outp1, outp2 = p.end_op(result)
```

Exception Handling

If an invocation raises an exception, the exception is thrown by the `end_` method, even if the actual error condition for the exception was encountered during the `begin_` method ("on the way out"). The advantage of this behavior is that all exception handling is located with the code that calls the `end_` method (instead of being present twice, once where the `begin_` method is called, and again where the `end_` method is called).

There is one exception to the above rule: if you destroy the communicator and then make an asynchronous invocation, the `begin_` method throws `CommunicatorDestroyedException`. This is necessary because, once the run time is finalized, it can no longer throw an exception from the `end_` method.

The only other exception that is thrown by the `begin_` and `end_` methods is `RuntimeError`. This exception indicates that you have used the API incorrectly. For example, the `begin_` method throws this exception if you call an operation that has a return value or out-parameters on a oneway proxy. Similarly, the `end_` method throws this exception if you use a different proxy to call the `end_` method than the proxy you used to call the `begin_` method, or if the `AsyncResult` you pass to the `end_` method was obtained by calling the `begin_` method for a different operation.

22.15.2 The `AsyncResult` Class

The `AsyncResult` that is returned by the `begin_` method encapsulates the state of the asynchronous invocation:

```
class AsyncResult:
    def getCommunicator()
    def getConnection()
    def getProxy()
    def getOperation()

    def isCompleted()
    def waitForCompleted()
```

```
def isSent()
def waitForSent()

def sentSynchronously()
```

The methods have the following semantics:

- `getCommunicator()`

  This method returns the communicator that sent the invocation.

- `getConnection()`

  This method returns the connection that was used for the invocation.

- `getProxy()`

  This method returns the proxy that was used to call the `begin_` method.

- `getOperation()`

  This method returns the name of the operation.

- `isCompleted()`

  This method returns true if, at the time it is called, the result of an invocation is available, indicating that a call to the `end_` method will not block the caller. Otherwise, if the result is not yet available, the method returns false.

- `waitForCompleted()`

  This method blocks the caller until the result of an invocation becomes available.

- `isSent()`

  When you call the `begin_` method, the Ice run time attempts to write the corresponding request to the client-side transport. If the transport cannot accept the request, the Ice run time queues the request for later transmission. `isSent` returns true if, at the time it is called, the request has been written to the local transport (whether it was initially queued or not). Otherwise, if the request is still queued, `isSent` returns false.

- `waitForSent()`

  This method blocks the calling thread until a request has been written to the client-side transport.

- `sentSynchronously()`

  This method returns true if a request was written to the client-side transport without first being queued. If the request was initially queued, `sentSyn-`

chronously returns false (independent of whether the request is still in the queue or has since been written to the client-side transport).

22.15.3 Polling for Completion

The AsyncResult methods allow you to poll for call completion. Polling is useful in a variety of cases. As an example, consider the following simple interface to transfer files from client to server:

```
interface FileTransfer
{
    void send(int offset, ByteSeq bytes);
};
```

The client repeatedly calls send to send a chunk of the file, indicating at which offset in the file the chunk belongs. A naïve way to transmit a file would be along the following lines:

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0
while not file.eof():
    bytes = file.read(chunkSize)  # Read a chunk
    ft.send(offset, bytes)        # Send the chunk
    offset += len(bytes.length)
```

This works, but not very well: because the client makes synchronous calls, it writes each chunk on the wire and then waits for the server to receive the data, process it, and return a reply before writing the next chunk. This means that both client and server spend much of their time doing nothing—the client does nothing while the server processes the data, and the server does nothing while it waits for the client to send the next chunk.

Using asynchronous calls, we can improve on this considerably:

```
file = open(...)
ft = FileTransferPrx.checkedCast(...)
chunkSize = ...
offset = 0

results = []
numRequests = 5

while not file.eof():
    bytes = file.read(chunkSize) # Read a chunk
```

```
    # Send up to numRequests + 1 chunks asynchronously.
    r = ft.begin_send(offset, bytes)
    offset += len(bytes)

    # Wait until this request has been passed to the transport.
    r.waitForSent()
    results.append(r)

    # Once there are more than numRequests, wait for the least
    # recent one to complete.
    while len(results) > numRequests:
        r = results[0]
        del results[0]
        r.waitForCompleted()

# Wait for any remaining requests to complete.
while len(results) > 0:
    r = results[0]
    del results[0]
    r.waitForCompleted()
```

With this code, the client sends up to `numRequests + 1` chunks before it waits for the least recent one of these requests to complete. In other words, the client sends the next request without waiting for the preceding request to complete, up to the limit set by `numRequests`. In effect, this allows the client to "keep the pipe to the server full of data": the client keeps sending data, so both client and server continuously do work.

Obviously, the correct chunk size and value of `numRequests` depend on the bandwidth of the network as well as the amount of time taken by the server to process each request. However, with a little testing, you can quickly zoom in on the point where making the requests larger or queuing more requests no longer improves performance. With this technique, you can realize the full bandwidth of the link to within a percent or two of the theoretical bandwidth limit of a native socket connection.

22.15.4 Completion Callbacks

The `begin_` method accepts three optional callback arguments that allow you to be notified asynchronously when a request completes. Here is the signature of the `begin_getName` method from Section 22.15.1 once more:

```
def begin_getName(self, number, _response=None, _ex=None,
                  _sent=None, _ctx=None)
```

The value you pass for the response callback (`_response`), the exception callback (`_ex`), or the sent callback (`_sent`) argument must be a *callable object* such as a function or method. The response callback is invoked when the request completes successfully, and the exception callback is invoked when the operation raises an exception. (We discuss the sent callback in Section 22.15.7.)

For example, consider the following callbacks for an invocation of the `getName` operation:

```
def getNameCB(name):
    print "Name is: " + name

def failureCB(ex):
    print "Exception is: " + str(ex)
```

The response callback parameters depend on the operation signature. If the operation has a non-`void` return type, the first parameter of the response callback is the return value. The return value (if any) is followed by a parameter for each out-parameter of the corresponding Slice operation, in the order of declaration.

The exception callback is invoked if the invocation fails because of an Ice run time exception, or if the operation raises a user exception.

To inform the Ice run time that you want to receive callbacks for the completion of the asynchronous call, you pass the callbacks to the `begin_` method:

```
e = EmployeesPrx.checkedCast(...)

e.begin_getName(99, getNameCB, failureCB)
```

Although the signature of an asynchronous proxy method implies that all of the callbacks are optional and therefore can be supplied in any combination, Ice enforces the following semantics at run time:

- If you omit all callbacks, you must call the `end_` method explicitly as described in Section 22.15.1.
- If you supply either a response callback or a sent callback (or both), you must also supply an exception callback.
- You may omit the response callback for an operation that returns no data (that is, an operation with a `void` return type and no out-parameters).

22.15.5 Sharing State Between the `begin_` and `end_` Method

It is common for the `end_` method to require access to some state that is established by the code that calls the `begin_` method. As an example, consider an application that asynchronously starts a number of operations and, as each operation completes, needs to update different user interface elements with the results. In this case, the `begin_` method knows which user interface element should receive the update, and the `end_` method needs access to that element.

Assuming that we have a `Widget` class that designates a particular user interface element, you could pass different widgets by storing the widget to be used as a member of a callback class:

```
class MyCallback(object):
    def __init__(self, w):
        self._w = w

    def getNameCB(self, name):
        self._w.writeString(name)

    def failureCB(self, ex):
        print "Exception is: " + str(ex)
```

For this example, we assume that widgets have a `writeString` method that updates the relevant UI element.

When you call the `begin_` method, you pass the appropriate callback instance to inform the `end_` method how to update the display:

```
e = EmployeesPrx.checkedCast(...)
widget1 = ...
widget2 = ...

# Invoke the getName operation with different widget callbacks.
cb1 = MyCallback(widget1)
e.begin_getName(99, cb1.getNameCB, cb1.failureCB)
cb2 = MyCallback(widget2)
e.begin_getName(24, cb2.getNameCB, cb2.failureCB)
```

The callback class provides a simple and effective way for you to pass state between the point where an operation is invoked and the point where its results are processed. Moreover, if you have a number of operations that share common state, you can pass the same callback instance to multiple invocations. (If you do this, your callback methods may need to use synchronization.)

For those situations in which a stateless callback is preferred, you can use a lambda function to pass state to a callback. Consider the following example:

```
def getNameCB(name, w):
    w.writeString(name)

def failureCB(ex):
    print "Exception is: " + str(ex)

e = EmployeesPrx.checkedCast(...)
widget1 = ...
widget2 = ...

# Use lambda functions to pass state.
e.begin_getName(99, lambda name: getNameCB(name, widget1),
                failureCB)
e.begin_getName(24, lambda name: getNameCB(name, widget2),
                failureCB)
```

This strategy eliminates the need to encapsulate shared state in a callback class. Since lambda functions can refer to variables in the enclosing scope, they provide a convenient way to pass state directly to your callback.

22.15.6 Oneway Invocations

You can invoke operations via oneway proxies asynchronously, provided the operation has `void` return type, does not have any out-parameters, and does not raise user exceptions. If you call the `begin_` method on a oneway proxy for an operation that returns values or raises a user exception, the `begin_` method throws a `RuntimeError`.

The callback signatures look exactly as for a twoway invocation, but the response method is never called and may be omitted.

22.15.7 Flow Control

Asynchronous method invocations never block the thread that calls the `begin_` method: the Ice run time checks to see whether it can write the request to the local transport. If it can, it does so immediately in the caller's thread. (In that case, `AsyncResult.sentSynchronously` returns true.) Alternatively, if the local transport does not have sufficient buffer space to accept the request, the Ice run time queues the request internally for later transmission in the background. (In that case, `AsyncResult.sentSynchronously` returns false.)

This creates a potential problem: if a client sends many asynchronous requests at the time the server is too busy to keep up with them, the requests pile up in the client-side run time until, eventually, the client runs out of memory.

The API provides a way for you to implement flow control by counting the number of requests that are queued so, if that number exceeds some threshold, the client stops invoking more operations until some of the queued operations have drained out of the local transport.

You can supply a sent callback to be notified when the request was successfully sent:

```
def response(name):
    # ...

def exception(ex):
    # ...

def sent(sentSynchronously):
    # ...
```

You inform the Ice run time that you want to be informed when a call has been passed to the local transport as usual:

```
e.begin_getName(99, response, exception, sent)
```

If the Ice run time can immediately pass the request to the local transport, it does so and invokes the sent callback from the thread that calls the `begin_` method. On the other hand, if the run time has to queue the request, it calls the sent callback from a different thread once it has written the request to the local transport. The boolean `sentSynchronously` parameter indicates whether the request was sent synchronously or was queued.

The sent callback allows you to limit the number of queued requests by counting the number of requests that are queued and decrementing the count when the Ice run time passes a request to the local transport.

22.15.8 Batch Requests

Applications that send batched requests (see Section 32.16) can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides asynchronous versions of this method so you can flush batch requests asynchronously.

`begin_ice_flushBatchRequests` and `end_ice_flushBatchRequests` are proxy methods that flush any batch requests queued by that proxy.

In addition, similar methods are available on the communicator and the `Connection` object that is returned by `AsyncResult.getConnection`. These methods flush batch requests sent via the same communicator and via the same connection, respectively.

22.15.9 Concurrency

The Ice run time always invokes your callback methods from a separate thread, with one exception: it calls the `sent` callback from the thread calling the `begin_` method if the request could be sent synchronously. In the `sent` callback, you know which thread is calling the callback by looking at the `sentSynchronously` parameter.

22.16 Code Generation

The Python mapping supports two forms of code generation: dynamic and static.

22.16.1 Dynamic Code Generation

Using dynamic code generation, Slice files are "loaded" at run time and dynamically translated into Python code, which is immediately compiled and available for use by the application. This is accomplished using the `Ice.loadSlice` function, as shown in the following example:

```
Ice.loadSlice("Color.ice")
import M

print "My favorite color is", M.Color.blue
```

For this example, we assume that `Color.ice` contains the following definitions:

```
module M {
    enum Color { red, green, blue };
};
```

The code imports module `M` after the Slice file is loaded because module `M` is not defined until the Slice definitions have been translated into Python.

`Ice.loadSlice` Options

The `Ice.loadSlice` function behaves like a Slice compiler in that it accepts command-line arguments for specifying preprocessor options and controlling code generation. The arguments must include at least one Slice file.

The function has the following Python definition:

```
def Ice.loadSlice(cmd, args=[])
```

The command-line arguments can be specified entirely in the first argument, `cmd`, which must be a string. The optional second argument can be used to pass additional command-line arguments as a list; this is useful when the caller already has the arguments in list form. The function always returns `None`.

For example, the following calls to `Ice.loadSlice` are functionally equivalent:

```
Ice.loadSlice("-I/opt/IcePy/slice Color.ice")
Ice.loadSlice("-I/opt/IcePy/slice", ["Color.ice"])
Ice.loadSlice("", ["-I/opt/IcePy/slice", "Color.ice"])
```

In addition to the standard compiler options described in Section 4.20, `Ice.loadSlice` also supports the following command-line options:

- **`--all`**

  Generate code for all Slice definitions, including those from included files.

- **`--checksum`**

  Generate checksums for Slice definitions. See Section 22.17 for more information.

Locating Slice Files

If your Slice files depend on Ice types, you can avoid hard-coding the path name of your Ice installation directory by calling the `Ice.getSliceDir` function:

```
Ice.loadSlice("-I" + Ice.getSliceDir() + " Color.ice")
```

This function attempts to locate the `slice` subdirectory of your Ice installation using an algorithm that succeeds for the following scenarios:

- Installation of a binary Ice archive
- Installation of an Ice source distribution using make install
- Installation via a Windows installer
- RPM installation on Linux
- Execution inside a compiled Ice source distribution

If the `slice` subdirectory can be found, `getSliceDir` returns its absolute path name, otherwise the function returns `None`.

Loading Multiple Files

You can specify as many Slice files as necessary in a single invocation of `Ice.loadSlice`, as shown below:

```
Ice.loadSlice("Syscall.ice Process.ice")
```

Alternatively, you can call `Ice.loadSlice` several times:

```
Ice.loadSlice("Syscall.ice")
Ice.loadSlice("Process.ice")
```

If a Slice file includes another file, the default behavior of `Ice.loadSlice` generates Python code only for the named file. For example, suppose `Syscall.ice` includes `Process.ice` as follows:

```
// Syscall.ice
#include <Process.ice>
...
```

If you call `Ice.loadSlice("-I. Syscall.ice")`, Python code is not generated for the Slice definitions in `Process.ice` or for any definitions that may be included by `Process.ice`. If you also need code to be generated for included files, one solution is to load them individually in subsequent calls to `Ice.loadSlice`. However, it is much simpler, not to mention more efficient, to use the `--all` option instead:

```
Ice.loadSlice("--all -I. Syscall.ice")
```

When you specify `--all`, `Ice.loadSlice` generates Python code for all Slice definitions included directly or indirectly from the named Slice files.

There is no harm in loading a Slice file multiple times, aside from the additional overhead associated with code generation. For example, this situation could arise when you need to load multiple top-level Slice files that happen to include a common subset of nested files. Suppose that we need to load both `Syscall.ice` and `Kernel.ice`, both of which include `Process.ice`. The simplest way to load both files is with a single call to `Ice.loadSlice`:

```
Ice.loadSlice("--all -I. Syscall.ice Kernel.ice")
```

Although this invocation causes the Ice extension to generate code twice for `Process.ice`, the generated code is structured so that the interpreter ignores

duplicate definitions. We could have avoided generating unnecessary code with the following sequence of steps:

```
Ice.loadSlice("--all -I. Syscall.ice")
Ice.loadSlice("-I. Kernel.ice")
```

In more complex cases, however, it can be difficult or impossible to completely avoid this situation, and the overhead of code generation is usually not significant enough to justify such an effort.

22.16.2 Static Code Generation

You should be familiar with static code generation if you have used other Slice language mappings, such as C++ or Java. Using static code generation, the Slice compiler **slice2py** (see Section 22.16.4) generates Python code from your Slice definitions.

Compiler Output

For each Slice file `X.ice`, **slice2py** generates Python code into a file named `X_ice.py`[3] in the output directory. The default output directory is the current working directory, but a different directory can be specified using the **--output-dir** option.

In addition to the generated file, **slice2py** creates a Python package for each Slice module it encounters. A Python package is nothing more than a subdirectory that contains a file with a special name (`__init__.py`). This file is executed automatically by Python when a program first imports the package. It is created by **slice2py** and must not be edited manually. Inside the file is Python code to import the generated files that contain definitions in the Slice module of interest.

For example, the Slice files `Process.ice` and `Syscall.ice` both define types in the Slice module `OS`. First we present `Process.ice`:

```
module OS {
    interface Process {
        void kill();
    };
};
```

3. Using the file name `X.py` would create problems if `X.ice` defined a module named X, therefore the suffix `_ice` is appended to the name of the generated file.

And here is `Syscall.ice`:

```
#include <Process.ice>
module OS {
    interface Syscall {
        Process getProcess(int pid);
    };
};
```

Next, we translate these files using the Slice compiler:

```
> slice2py -I. Process.ice Syscall.ice
```

If we list the contents of the output directory, we see the following entries:

```
OS/
Process_ice.py
Syscall_ice.py
```

The subdirectory `OS` is the Python package that **`slice2py`** created for the Slice module `OS`. Inside this directory is the special file `__init__.py` that contains the following statements:

```
import Process_ice
import Syscall_ice
```

Now when a Python program executes `import OS`, the two files `Process_ice.py` and `Syscall_ice.py` are implicitly imported.

Subsequent invocations of **`slice2py`** for Slice files that also contain definitions in the `OS` module result in additional `import` statements being added to `OS/__init__.py`. Be aware, however, that `import` statements may persist in `__init__.py` files after a Slice file is renamed or becomes obsolete. This situation may manifest itself as a run-time error if the interpreter can no longer locate the generated file while attempting to import the package. It may also cause more subtle problems, if an obsolete generated file is still present and being loaded unintentionally. In general, it is advisable to remove the package directory and regenerate it whenever the set of Slice files changes.

A Python program may also import a generated file explicitly, using a statement such as `import Process_ice`. Typically, however, it is more convenient to import the Python module once, rather than importing potentially several individual files that comprise the module, especially when you consider that the program must still import the module explicitly in order to make its definitions available. For example, it is much simpler to state

```
import OS
```

rather than the following alternative:

```
import Process_ice
import Syscall_ice
import OS
```

In situations where a Python package is unnecessary or undesirable, the **--no-package** option can be specified to prevent the creation of a package. In this case, the application must import the generated file(s) explicitly, as shown above.

Include Files

It is important to understand how **slice2py** handles include files. In the absence of the **--all** option, the compiler does not generate Python code for Slice definitions in included files. Rather, the compiler translates Slice `#include` statements into Python `import` statements in the following manner:

1. Determine the full pathname of the include file.
2. Create the shortest possible relative pathname for the include file by iterating over each of the include directories (specified using the `-I` option) and removing the leading directory from the include file if possible.

   For example, if the full pathname of an include file is `/opt/App/slice/OS/Process.ice`, and we specified the options `-I/opt/App` and `-I/opt/App/slice`, then the shortest relative pathname is `OS/Process.ice` after removing `/opt/App/slice`.
3. Replace any slashes with underscores, remove the `.ice` extension, and append `_ice`. Continuing our example from the previous step, the translated `import` statement becomes

   ```
   import OS_Process_ice
   ```

There is a potential problem here that must be addressed. The generated `import` statement shown above expects to find the file `OS_Process_ice.py` somewhere in Python's search path. However, **slice2py** uses a different default name, `Process_ice.py`, when it compiles `Process.ice`. To resolve this issue, we must use the **--prefix** option when compiling `Process.ice`:

```
> slice2py --prefix OS_ Process.ice
```

The **--prefix** option causes the compiler to prepend the specified prefix to the name of each generated file. When executed, the above command creates the desired file name: `OS_Process_ice.py`.

It should be apparent by now that generating Python code for a complex Ice application requires a bit of planning. In particular, it is imperative that you be consistent in your use of `#include` statements, include directories, and **`--prefix`** options to ensure that the correct file names are used at all times.

Of course, these precautionary steps are only necessary when you are compiling Slice files individually. An alternative is to use the **`--all`** option and generate Python code for all of your Slice definitions into one Python source file. If you do not have a suitable Slice file that includes all necessary Slice definitions, you could write a "master" Slice file specifically for this purpose.

22.16.3 Static Versus Dynamic Code Generation

There are several issues to consider when evaluating your requirements for code generation.

Application Considerations

The requirements of your application generally dictate whether you should use dynamic or static code generation. Dynamic code generation is convenient for a number of reasons:

- it avoids the intermediate compilation step required by static code generation
- it makes the application more compact because the application requires only the Slice files, not the assortment of files and directories produced by static code generation
- it reduces complexity, which is especially helpful during testing, or when writing short or transient programs.

Static code generation, on the other hand, is appropriate in many situations:

- when an application uses a large number of Slice definitions and the startup delay must be minimized
- when it is not feasible to deploy Slice files with the application
- when a number of applications share the same Slice files
- when Python code is required in order to utilize third-party Python tools.

Mixing Static and Dynamic Generation

Using a combination of static and dynamic translation in an application can produce unexpected results. For example, consider a situation where a dynamically-translated Slice file includes another Slice file that was statically translated:

```
// Slice
#include <Glacier2/Session.ice>

module App {
    interface SessionFactory {
        Glacier2::Session* createSession();
    };
};
```

The Slice file `Session.ice` is statically translated, as are all of the Slice files included with the Ice run time.

Assuming the above definitions are saved in `App.ice`, let's execute a simple Python script:

```
# Python
import Ice
Ice.loadSlice("-I/opt/Ice/slice App.ice")

import Glacier2
class MyVerifier(Glacier2.PermissionsVerifier): # Error
    def checkPermissions(self, userId, password):
        return (True, "")
```

The code looks reasonable, but running it produces the following error:

```
'module' object has no attribute 'PermissionsVerifier'
```

Normally, importing the Glacier2 module as we have done here would load all of the Python code generated for the Glacier2 Slice files. However, since `App.ice` has already included a subset of the Glacier2 definitions, the Python interpreter ignores any subsequent requests to import the entire module, and therefore the `PermissionsVerifier` type is not present.

One way to address this problem is to import the statically-translated modules first, prior to loading Slice files dynamically:

```
# Python
import Ice, Glacier2 # Import Glacier2 before App.ice is loaded
Ice.loadSlice("-I/opt/Ice/slice App.ice")

class MyVerifier(Glacier2.PermissionsVerifier): # OK
    def checkPermissions(self, userId, password):
        return (True, "")
```

The disadvantage of this approach in a non-trivial application is that it breaks encapsulation, forcing one Python module to know what other modules are doing.

For example, suppose we place our `PermissionsVerifier` implementation in a module named `verifier.py`:

```
# Python
import Glacier2
class MyVerifier(Glacier2.PermissionsVerifier):
    def checkPermissions(self, userId, password):
        return (True, "")
```

Now that the use of Glacier2 definitions is encapsulated in `verifier.py`, we would like to remove references to Glacier2 from the main script:

```
# Python
import Ice
Ice.loadSlice("-I/opt/Ice/slice App.ice")
...
import verifier # Error
v = verifier.MyVerifier()
```

Unfortunately, executing this script produces the same error as before. To fix it, we have to break the `verifier` module's encapsulation and import the `Glacier2` module in the main script because we know that the `verifier` module requires it:

```
# Python
import Ice, Glacier2
Ice.loadSlice("-I/opt/Ice/slice App.ice")
...
import verifier # OK
v = verifier.MyVerifier()
```

Although breaking encapsulation in this way might offend our sense of good design, it is a relatively minor issue.

Another solution is to import the necessary submodules explicitly. We can safely remove the Glacier2 reference from our main script after rewriting `verifier.py` as shown below:

```
# Python
import Glacier2_PermissionsVerifier_ice
import Glacier2
class MyVerifier(Glacier2.PermissionsVerifier):
    def checkPermissions(self, userId, password):
        return (True, "")
```

Using the rules defined in Section 22.16.2, we can derive the name of the module containing the code generated for `PermissionsVerifier.ice` and import it

directly. We need a second `import` statement to make the Glacier2 definitions accessible in this module.

22.16.4 `slice2py` Command-Line Options

The Slice-to-Python compiler, **`slice2py`**, offers the following command-line options in addition to the standard options described in Section 4.20:

- **`--all`**

  Generate code for all Slice definitions, including those from included files.

- **`--checksum`**

  Generate checksums for Slice definitions.

- **`--prefix` *`PREFIX`***

  Use ***`PREFIX`*** as the prefix for generated file names. See Section 22.16.2 for more information.

22.16.5 Packages

By default, the scope of a Slice definition determines the module of its mapped Python construct (see Section 22.4 for more information on the module mapping). There are times, however, when applications require greater control over the packaging of generated Python code. For example, consider the following Slice definitions:

```
module sys {
    interface Process {
        // ...
    };
};
```

Other language mappings can use these Slice definitions as shown, but they present a problem for the Python mapping: the top-level Slice module `sys` conflicts with Python's predefined module `sys`. A Python application executing the statement `import sys` would import whichever module the interpreter happens to locate first in its search path.

A workaround for this problem is to modify the Slice definitions so that the top-level module no longer conflicts with a predefined Python module, but that may not be feasible in certain situations. For example, the application may already be deployed using other language mappings, in which case the impact of modifying the Slice definitions could represent an unacceptable expense.

The Python mapping could have addressed this issue by considering the names of predefined modules to be reserved, in which case the Slice module `sys` would be mapped to the Python module `_sys`. However, the likelihood of a name conflict is relatively low to justify such a solution, therefore the mapping supports a different approach: global metadata (see Section 4.17) can be used to enclose generated code in a Python package. Our modified Slice definitions demonstrate this feature:

```
[["python:package:zeroc"]]
module sys {
    interface Process {
        // ...
    };
};
```

The global metadata directive `python:package:zeroc` causes the mapping to generate all of the code resulting from definitions in this Slice file into the Python package `zeroc`. The net effect is the same as if we had enclosed our Slice definitions in the module `zeroc`: the Slice module `sys` is mapped to the Python module `zeroc.sys`. However, by using metadata we have not affected the semantics of the Slice definitions, nor have we affected other language mappings.

22.17 Using Slice Checksums

As described in Section 4.21, the Slice compilers can optionally generate checksums of Slice definitions. For **`slice2py`**, the **`--checksum`** option causes the compiler to generate code that adds checksums to the dictionary `Ice.sliceChecksums`. The checksums are installed automatically when the Python code is first imported; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the dictionaries using the comparison operator. However, this is not feasible if it is possible that the server might return a superset of the client's checksums. A more general solution is to iterate over the local checksums as demonstrated below:

```
serverChecksums = ...
for i in Ice.sliceChecksums:
    if not serverChecksums.has_key(i):
        # No match found for type id!
    elif Ice.sliceChecksums[i] != serverChecksums[i]:
        # Checksum mismatch!
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

Chapter 23
Developing a File System Client in Python

23.1 Chapter Overview

In this chapter, we present the source code for a Python client that accesses the file system we developed in Chapter 5 (see Chapter 25 for the corresponding server).

23.2 The Python Client

We now have seen enough of the client-side Python mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
```

```
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

```
import sys, traceback, Ice, Filesystem

# Recursively print the contents of directory "dir"
# in tree fashion. For files, show the contents of
# each file. The "depth" parameter is the current
# nesting level (for indentation).

def listRecursive(dir, depth):
    indent = ''
    depth = depth + 1
    for i in range(depth):
        indent = indent + '\t'

    contents = dir.list()

    for node in contents:
        subdir = Filesystem.DirectoryPrx.checkedCast(node)
        file = Filesystem.FilePrx.uncheckedCast(node)
        print indent + node.name(),
        if subdir:
            print "(directory):"
            listRecursive(subdir, depth)
        else:
            print "(file):"
            text = file.read()
            for line in text:
                print indent + "\t" + line
```

```
status = 0
ic = None
try:
    # Create a communicator
    #
    ic = Ice.initialize(sys.argv)

    # Create a proxy for the root directory
    #
    obj = ic.stringToProxy("RootDir:default -p 10000")

    # Down-cast the proxy to a Directory proxy
    #
    rootDir = Filesystem.DirectoryPrx.checkedCast(obj)

    # Recursively list the contents of the root directory
    #
    print "Contents of root directory:"
    listRecursive(rootDir, 0)
except:
    traceback.print_exc()
    status = 1

if ic:
    # Clean up
    #
    try:
        ic.destroy()
    except:
        traceback.print_exc()
        status = 1

sys.exit(status)
```

The program first defines the `listRecursive` function, which is a helper function to print the contents of the file system, and the main program follows. Let us look at the main program first:

1. The structure of the code follows what we saw in Chapter 34. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.
2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.

   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.

2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.

3. The code checks the type of the node:

   - If it is a directory, the code recurses, incrementing the indent level.
   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of a two files and a a directory as follows:

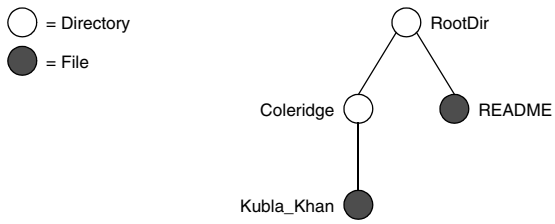


Figure 23.1. A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client (and server) are not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in Chapter 38 and Chapter 34.

23.3 Summary

This chapter presented a very simple client to access a server that implements the file system we developed in Chapter 5. As you can see, the Python code hardly differs from the code you would write for an ordinary Python program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Python object. This allows you to put your effort where you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs. As we will see in Chapter 25, this is true for the server side as well, meaning that you can develop distributed applications easily and efficiently.

Chapter 24
Server-Side Slice-to-Python Mapping

24.1 Chapter Overview

In this chapter, we present the server-side Slice-to-Python mapping (see Chapter 22 for the client-side mapping). Section 24.3 discusses how to initialize and finalize the server-side run time, sections 24.4 to 24.6 show how to implement interfaces and operations, and Section 24.7 discusses how to register objects with the server-side Ice run time. Finally, Section 24.8 shows how to implement operations asynchronously.

24.2 Introduction

The mapping for Slice data types to Python is identical on the client side and server side. This means that everything in Chapter 22 also applies to the server side. However, for the server side, there are a few additional things you need to know, specifically:

- how to initialize and finalize the server-side run time
- how to implement servants
- how to pass parameters and throw exceptions
- how to create servants and register them with the Ice run time.

We discuss these topics in the remainder of this chapter.

24.3 The Server-Side Main Program

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. As for the client side, you must initialize the Ice run time by calling `Ice.initialize` before you can do anything else in your server. `Ice.initialize` returns a reference to an instance of an `Ice.Communicator`:

```
import sys, traceback, Ice

status = 0
ic = None
try:
    ic = Ice.initialize(sys.argv)
    # ...
except:
    traceback.print_exc()
    status = 1

# ...
```

`Ice.initialize` accepts the argument list that is passed to the program by the operating system. The function scans the argument list for any command-line options that are relevant to the Ice run time; any such options are removed from the argument list so, when `Ice.initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

You can pass a second argument of type `InitializationData` to `Ice.initialize`. `InitializationData` is defined as follows:

```
class InitializationData(object):
    def __init__(self):
        self.properties = None
        self.logger = None
        self.threadHook = None
```

You can pass in an instance of this class to set properties for the communicator (see Chapter 30), establish a logger (see Section 32.19), and to establish a thread notification hook (see page 1016).

Before leaving your program, you *must* call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular,

`destroy` waits for any operation implementations that are still executing in the server to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your program to terminate without calling `destroy` first; doing so has undefined behavior.

The general shape of our server-side program is therefore as follows:

```
import sys, traceback, Ice

status = 0
ic = None
try:
    ic = Ice.initialize(sys.argv)
    # ...
except:
    traceback.print_exc()
    status = 1

if ic:
    try:
        ic.destroy()
    except:
        traceback.print_exc()
        status = 1

sys.exit(status)
```

Note that the code places the call to `Ice.initialize` into a `try` block and takes care to return the correct exit status to the operating system. Also note that an attempt to destroy the communicator is made only if the initialization succeeded.

24.3.1 The `Ice.Application` Class

The preceding program structure is so common that Ice offers a class, `Ice.Application`, that encapsulates all the correct initialization and finalization activities. The synopsis of the class is as follows (with some detail omitted for now):

```
class Application(object):

    def __init__(self, signalPolicy=0):

    def main(self, args, configFile=None, initData=None):
```

```
    def run(self, args):

    def appName():
        # ...
    appName = staticmethod(appName)

    def communicator():
        # ...
    communicator = staticmethod(communicator)
```

The intent of this class is that you specialize `Ice.Application` and implement the abstract `run` method in your derived class. Whatever code you would normally place in your main program goes into `run` instead. Using `Ice.Application`, our program looks as follows:

```
import sys, Ice

class Server(Ice.Application):
    def run(self, args):
        # Server code here...
        return 0

app = Server()
status = app.main(sys.argv)
sys.exit(status)
```

You also can call `main` with an optional file name or an `InitializationData` structure (see Section 32.3 and Section 30.9). If you pass a configuration file name to `main`, the settings in this file are overridden by settings in a file identified by the `ICE_CONFIG` environment variable (if defined). Property settings supplied on the command line take precedence over all other settings.

The `Application.main` function does the following:

1. It installs an exception handler. If your code fails to handle an exception, `Application.main` prints the exception information before returning with a non-zero return value.
2. It initializes (by calling `Ice.initialize`) and finalizes (by calling `Communicator.destroy`) a communicator. You can get access to the communicator for your server by calling the static `communicator` accessor.
3. It scans the argument list for options that are relevant to the Ice run time and removes any such options. The argument list that is passed to your `run` method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.

4. It provides the name of your application via the static `appName` member function. The return value from this call is the first element of the argument vector passed to `Application.main`, so you can get at this name from anywhere in your code by calling `Ice.Application.appName` (which is usually required for error messages).
5. It installs a signal handler that properly shuts down the communicator.
6. It installs a per-process logger (see Section 32.19.5) if the application has not already configured one. The per-process logger uses the value of the `Ice.ProgramName` property (see Section 30.8) as a prefix for its messages and sends its output to the standard error channel. An application can specify an alternate logger as described in Section 32.19.

Using `Ice.Application` ensures that your program properly finalizes the Ice run time, whether your server terminates normally or in response to an exception or signal. We recommend that all your programs use this class; doing so makes your life easier. In addition `Ice.Application` also provides features for signal handling and configuration that you do not have to implement yourself when you use this class.

Using `Ice.Application` on the Client Side

You can use `Ice.Application` for your clients as well: simply implement a class that derives from `Ice.Application` and place the client code into its `run` method. The advantage of this approach is the same as for the server side: `Ice.Application` ensures that the communicator is destroyed correctly even in the presence of exceptions or signals.

Catching Signals

The simple server we developed in Chapter 3 had no way to shut down cleanly: we simply interrupted the server from the command line to force it to exit. Terminating a server in this fashion is unacceptable for many real-life server applications: typically, the server has to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is particularly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

To make it easier to deal with signals, `Ice.Application` encapsulates Python's signal handling capabilities, allowing you to cleanly shut down on receipt of a signal:

```
class Application(object):
    # ...
    def destroyOnInterrupt():
        # ...
    destroyOnInterrupt = classmethod(destroyOnInterrupt)

    def shutdownOnInterrupt():
        # ...
    shutdownOnInterrupt = classmethod(shutdownOnInterrupt)

    def ignoreInterrupt():
        # ...
    ignoreInterrupt = classmethod(ignoreInterrupt)

    def callbackOnInterrupt():
        # ...
    callbackOnInterrupt = classmethod(callbackOnInterrupt)

    def holdInterrupt():
        # ...
    holdInterrupt = classmethod(holdInterrupt)

    def releaseInterrupt():
        # ...
    releaseInterrupt = classmethod(releaseInterrupt)

    def interrupted():
        # ...
    interrupted = classmethod(interrupted)

    def interruptCallback(self, sig):
        # Default implementation does nothing.
        pass
```

The methods behave as follows:

- `destroyOnInterrupt`

  This method installs a signal handler that destroys the communicator if it is interrupted. This is the default behavior.

- `shutdownOnInterrupt`

  This method installs a signal handler that shuts down the communicator if it is interrupted.

- `ignoreInterrupt`

  This method causes signals to be ignored.

- `callbackOnInterrupt`

  This function configures `Ice.Application` to invoke `interrupt-Callback` when a signal occurs, thereby giving the subclass responsibility for handling the signal.

- `holdInterrupt`

  This method temporarily blocks signal delivery.

- `releaseInterrupt`

  This method restores signal delivery to the previous disposition. Any signal that arrives after `holdInterrupt` was called is delivered when you call `releaseInterrupt`.

- `interrupted`

  This method returns `True` if a signal caused the communicator to shut down, `False` otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by a signal. This is useful, for example, for logging purposes.

- `interruptCallback`

  A subclass overrides this function to respond to signals. The function may be called concurrently with any other thread and must not raise exceptions.

By default, `Ice.Application` behaves as if `destroyOnInterrupt` was invoked, therefore our server program requires no change to ensure that the program terminates cleanly on receipt of a signal. (You can disable the signal-handling functionality of `Ice.Application` by passing `1` to the constructor. In that case, signals retain their default behavior, that is, terminate the process.) However, we add a diagnostic to report the occurrence of a signal, so our program now looks like:

```
import sys, Ice

class MyApplication(Ice.Application):
    def run(self, args):

        # Server code here...

        if self.interrupted():
            print self.appName() + ": terminating"

        return 0
```

```
app = MyApplication()
status = app.main(sys.argv)
sys.exit(status)
```

Note that, if your server is interrupted by a signal, the Ice run time waits for all currently executing operations to finish. This means that an operation that updates persistent state cannot be interrupted in the middle of what it was doing and cause partial update problems.

`Ice.Application` and Properties

Apart from the functionality shown in this section, `Ice.Application` also takes care of initializing the Ice run time with property values. Properties allow you to configure the run time in various ways. For example, you can use properties to control things such as the thread pool size or port number for a server. The `main` method of `Ice.Application` accepts an optional second parameter allowing you to specify the name of a configuration file that will be processed during initialization. We discuss Ice properties in more detail in Chapter 30.

Limitations of `Ice.Application`

`Ice.Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use `Ice.Application`. Instead, you must structure your code as we saw in Chapter 3 (taking care to always destroy the communicator).

24.4 Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing methods in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

24.4.1 Skeleton Classes

On the client side, interfaces map to proxy classes (see Section 5.12). On the server side, interfaces map to *skeleton* classes. A skeleton is an abstract base class from which you derive your servant class and define a method for each operation on the corresponding interface. For example, consider the Slice definition for the `Node` interface we defined in Chapter 5 once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The Python mapping generates the following definition for this interface:

```
class Node(Ice.Object):
    def __init__(self):
        # ...

    #
    # Operation signatures.
    #
    # def name(self, current=None):
```

The important points to note here are:

- As for the client side, Slice modules are mapped to Python modules with the same name, so the skeleton class definitions are part of the `Filesystem` module.
- The name of the skeleton class is the same as the name of the Slice interface (`Node`).
- The skeleton class contains a comment summarizing the method signature of each operation in the Slice interface.
- The skeleton class is an abstract base class because its constructor prevents direct instantiation of the class.
- The skeleton class inherits from `Ice.Object` (which forms the root of the Ice object hierarchy).

24.4.2 Servant Classes

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

```
import Filesystem

class NodeI(Filesystem.Node):
    def __init__(self, name):
```

```
        self._name = name

    def name(self, current=None):
        return self._name
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.) Note that `NodeI` extends `Filesystem.Node`, that is, it derives from its skeleton class.

As far as Ice is concerned, the `NodeI` class must implement only a single method: the `name` method that is defined in the `Node` interface. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a `_name` member and a constructor. (Obviously, the constructor initializes the `_name` member and the `name` function returns its value.)

Normal and `idempotent` Operations

Whether an operation is an ordinary operation or an `idempotent` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

```
interface Example {
               void normalOp();
    idempotent void idempotentOp();
};
```

The mapping for this interface is shown below:

```
class Example(Ice.Object):
    # ...

    #
    # Operation signatures.
    #
    # def normalOp(self, current=None):
    # def idempotentOp(self, current=None):
```

Note that the signatures of the methods are unaffected by the `idempotent` qualifier.

24.5 Parameter Passing

For each `in` parameter of a Slice operation, the Python mapping generates a corresponding parameter for the corresponding method. In addition, every operation has an additional, trailing parameter of type `Ice.Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` method in a Python servant has a `current` parameter. We explain the purpose of this parameter in Section 32.6 and will ignore it for now.

An operation returning multiple values[1] returns them in a tuple consisting of a non-`void` return value, if any, followed by the `out` parameters in the order of declaration. An operation returning only one value simply returns the value itself.

To illustrate these rules, consider the following interface that passes string parameters in all possible directions:

```
interface Example {
    string op1(string sin);
    void op2(string sin, out string sout);
    string op3(string sin, out string sout);
};
```

The generated skeleton class for this interface looks as follows:

```
class Example(Ice.Object):
    def __init__(self):
        # ...

    #
    # Operation signatures.
    #
    # def op1(self, sin, current=None):
    # def op2(self, sin, current=None):
    # def op3(self, sin, current=None):
```

The signatures of the Python methods are identical because they all accept a single `in` parameter, but their implementations differ in the way they return values. For example, we could implement the operations as follows:

1. An operation returns multiple values when it declares multiple `out` parameters, or when it declares a non-`void` return type and at least one `out` parameter.

```
class ExampleI(Example):
    def op1(self, sin, current=None):
        print sin             # In params are initialized
        return "Done"         # Return value

    def op2(self, sin, current=None):
        print sin             # In params are initialized
        return "Hello World!" # Out parameter

    def op3(self, sin, current=None):
        print sin             # In params are initialized
        return ("Done", "Hello World!")
```

Notice that `op1` and `op2` return their string values directly, whereas `op3` returns a tuple consisting of the return value followed by the `out` parameter.

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal Python rules and do not require special-purpose API calls.

24.6 Raising Exceptions

To throw an exception from an operation implementation, you simply instantiate the exception, initialize it, and throw it. For example:

```
class FileI(Filesystem.File):
    # ...

    def write(self, text, current=None):
        # Try to write the file contents here...
        # Assume we are out of space...
        if error:
            e = Filesystem.GenericError()
            e.reason = "file too large"
            raise e
```

The mapping for exceptions (see Section 22.9) generates a constructor that accepts values for data members, so we can simplify this example by changing our `raise` statement to the following:

```
class FileI(Filesystem.File):
    # ...

    def write(self, text, current=None):
        # Try to write the file contents here...
        # Assume we are out of space...
        if error:
            raise Filesystem.GenericError("file too large")
```

If you throw an arbitrary Python run-time exception, the Ice run time catches the exception and then returns an `UnknownException` to the client. Similarly, if you throw an "impossible" user exception (a user exception that is not listed in the exception specification of the operation), the client receives an `UnknownUserException`.

If you throw an Ice run-time exception, such `MemoryLimitException`, the client receives an `UnknownLocalException`.[2] For that reason, you should never throw system exceptions from operation implementations. If you do, all the client will see is an `UnknownLocalException`, which does not tell the client anything useful.

24.7 Object Incarnation

Having created a servant class such as the rudimentary `NodeI` class in Section 24.4.2, you can instantiate the class to create a concrete servant that can receive invocations from a client. However, merely instantiating a servant class is insufficient to incarnate an object. Specifically, to provide an implementation of an Ice object, you must take the following steps:

1. Instantiate a servant class.
2. Create an identity for the Ice object incarnated by the servant.
3. Inform the Ice run time of the existence of the servant.
4. Pass a proxy for the object to a client so the client can reach it.

2. There are three run-time exceptions that are not changed to `UnknownLocalException` when returned to the client: `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`. We discuss these exceptions in more detail in Chapter 33.

24.7.1 Instantiating a Servant

Instantiating a servant means to allocate an instance:

```
servant = NodeI("Fred")
```

This statement creates a new `NodeI` instance and assigns its reference to the variable `servant`.

24.7.2 Creating an Identity

Each Ice object requires an identity. That identity must be unique for all servants using the same object adapter.[3] An Ice object identity is a structure with the following Slice definition:

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
    // ...
};
```

The full identity of an object is the combination of both the `name` and `category` fields of the `Identity` structure. For now, we will leave the `category` field as the empty string and simply use the `name` field. (See Section 32.6 for a discussion of the `category` field.)

To create an identity, we simply assign a key that identifies the servant to the `name` field of the `Identity` structure:

```
id = Ice.Identity()
id.name = "Fred" # Not unique, but good enough for now
```

Note that the mapping for structures (see Section 22.7.2) allows us to write the following equivalent code:

```
id = Ice.Identity("Fred") # Not unique, but good enough for now
```

3. The Ice object model assumes that all objects (regardless of their adapter) have a globally unique identity. See Chapter 34 for further discussion.

24.7.3 Activating a Servant

Merely creating a servant instance does nothing: the Ice run time becomes aware of the existence of a servant only once you explicitly tell the object adapter about the servant. To activate a servant, you invoke the `add` operation on the object adapter. Assuming that we have access to the object adapter in the `adapter` variable, we can write:

```
adapter.add(servant, id)
```

Note the two arguments to `add`: the servant and the object identity. Calling `add` on the object adapter adds the servant and the servant's identity to the adapter's servant map and links the proxy for an Ice object to the correct servant instance in the server's memory as follows:

1. The proxy for an Ice object, apart from addressing information, contains the identity of the Ice object. When a client invokes an operation, the object identity is sent with the request to the server.
2. The object adapter receives the request, retrieves the identity, and uses the identity as an index into the servant map.
3. If a servant with that identity is active, the object adapter retrieves the servant from the servant map and dispatches the incoming request into the correct member function on the servant.

Assuming that the object adapter is in the active state (see Section 32.4), client requests are dispatched to the servant as soon as you call `add`.

24.7.4 UUIDs as Identities

The Ice object model assumes that object identities are globally unique. One way of ensuring that uniqueness is to use UUIDs (Universally Unique Identifiers) [14] as identities. The `Ice.generateUUID` function creates such identities:

```
import Ice
print Ice.generateUUID()
```

When executed, this program prints a unique string such as `5029a22c-e333-4f87-86b1-cd5e0fcce509`. Each call to `generateUUID` creates a string that differs from all previous ones.[4] You can use a UUID

4. Well, almost: eventually, the UUID algorithm wraps around and produces strings that repeat themselves, but this will not happen until approximately the year 3400.

such as this to create object identities. For convenience, the object adapter has an operation `addWithUUID` that generates a UUID and adds a servant to the servant map in a single step. Using this operation, we can create an identity and register a servant with that identity in a single step as follows:

```
adapter.addWithUUID(NodeI("Fred"))
```

24.7.5 Creating Proxies

Once we have activated a servant for an Ice object, the server can process incoming client requests for that object. However, clients can only access the object once they hold a proxy for the object. If a client knows the server's address details and the object identity, it can create a proxy from a string, as we saw in our first example in Chapter 3. However, creation of proxies by the client in this manner is usually only done to allow the client access to initial objects for bootstrapping. Once the client has an initial proxy, it typically obtains further proxies by invoking operations.

The object adapter contains all the details that make up the information in a proxy: the addressing and protocol information, and the object identity. The Ice run time offers a number of ways to create proxies. Once created, you can pass a proxy to the client as the return value or as an out-parameter of an operation invocation.

Proxies and Servant Activation

The `add` and `addWithUUID` servant activation operations on the object adapter return a proxy for the corresponding Ice object. This means we can write:

```
proxy = adapter.addWithUUID(NodeI("Fred"))
nodeProxy = Filesystem.NodePrx.uncheckedCast(proxy)

# Pass nodeProxy to client...
```

Here, `addWithUUID` both activates the servant and returns a proxy for the Ice object incarnated by that servant in a single step.

Note that we need to use an `uncheckedCast` here because `addWithUUID` returns a proxy of type `Ice.ObjectPrx`.

Direct Proxy Creation

The object adapter offers an operation to create a proxy for a given identity:

```
module Ice {
    local interface ObjectAdapter {
        Object* createProxy(Identity id);
        // ...
    };
};
```

Note that `createProxy` creates a proxy for a given identity whether a servant is activated with that identity or not. In other words, proxies have a life cycle that is quite independent from the life cycle of servants:

```
id = Ice.Identity()
id.name = Ice.generateUUID()
proxy = adapter.createProxy(id)
```

This creates a proxy for an Ice object with the identity returned by `generateUUID`. Obviously, no servant yet exists for that object so, if we return the proxy to a client and the client invokes an operation on the proxy, the client will receive an `ObjectNotExistException`. (We examine these life cycle issues in more detail in Chapter 34.)

24.8 Asynchronous Method Dispatch (AMD)

The number of simultaneous synchronous requests a server is capable of supporting is determined by the number of threads in the server's thread pool (see Section 32.10). If all of the threads are busy dispatching long-running operations, then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (*AMD*), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server sends a response explicitly using a callback object provided by the Ice run time.

AMD is transparent to the client, that is, there is no way for a client to distinguish a request that, in the server, is processed synchronously from a request that is processed asynchronously.

In practical terms, an AMD operation typically queues the request data (i.e., the callback object and operation arguments) for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch

threads and becomes capable of efficiently supporting thousands of simultaneous clients.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results while still in the dispatch thread, and then continues using the dispatch thread for additional work.

24.8.1 Enabling AMD with Metadata

To enable asynchronous dispatch, you must add an `["amd"]` metadata directive to your Slice definitions. The directive applies at the interface and the operation level. If you specify `["amd"]` at the interface level, all operations in that interface use asynchronous dispatch; if you specify `["amd"]` for an individual operation, only that operation uses asynchronous dispatch. In either case, the metadata directive *replaces* synchronous dispatch, that is, a particular operation implementation must use synchronous or asynchronous dispatch and cannot use both.

Consider the following Slice definitions:

```
["amd"] interface I {
  bool isValid();
  float computeRate();
};

interface J {
  ["amd"] void startProcess();
  int endProcess();
};
```

In this example, both operations of interface `I` use asynchronous dispatch, whereas, for interface `J`, `startProcess` uses asynchronous dispatch and `endProcess` uses synchronous dispatch.

Specifying metadata at the operation level (rather than at the interface or class level) minimizes the amount of generated code and, more importantly, minimizes complexity: although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations that need it, while using the simpler synchronous model for the rest.

24.8.2 AMD Mapping

For each AMD operation, the Python mapping emits a dispatch method with the same name as the operation and the suffix `_async`. This method returns `None`. The first parameter is a reference to a callback object, as described below. The remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

The callback object defines two methods:

- `def ice_response(self, <params>)`

  The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-`void` return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's `out` parameters follow the return value, in the order of declaration.

- `def ice_exception(self, ex)`

  The `ice_exception` method allows the server to report an exception.

Neither `ice_response` nor `ice_exception` throw any exceptions to the caller.

Suppose we have defined the following operation:

```
interface I {
  ["amd"] int foo(short s, out long l);
};
```

The callback interface generated for operation `foo` is shown below:

```
class ...
    #
    # Operation signatures.
    #
    # def ice_response(self, _result, l)
    # def ice_exception(self, ex)
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

```
def foo_async(self, __cb, s)
```

24.8.3 Exceptions

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (the thread

that receives the invocation), and the response thread (the thread that sends the response).[5] Although we recommend that the callback object be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run time; the application's run time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the callback object. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

Whether raised in a dispatch thread or reported via the callback object, user exceptions are validated as described in Section 4.10.2, and local exceptions may undergo the translation described in Section 4.10.4.

24.8.4 Example

To demonstrate the use of AMD in Ice, let us define the Slice interface for a simple computational engine:

```
module Demo {
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {};

    interface Model {
        ["amd"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    };
};
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way.

Our servant class derives from `Demo.Model` and supplies a definition for the `interpolate_async` method that creates a `Job` to hold the callback object

5. These are not necessarily two different threads: it is legal to send the response from the dispatch thread.

and arguments, and adds the `Job` to a queue. The method uses a lock to guard access to the queue:

```
class ModelI(Demo.Model):
    def __init__(self):
        self._mutex = threading.Lock()
        self._jobs = []

    def interpolate_async(self, cb, data, factor, current=None):
        self._mutex.acquire()
        try:
            self._jobs.append(Job(cb, data, factor))
        finally:
            self._mutex.release()
```

After queuing the information, the operation returns control to the Ice run time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute`, which uses `interpolateGrid` (not shown) to perform the computational work:

```
class Job(object):
    def __init__(self, cb, grid, factor):
        self._cb = cb
        self._grid = grid
        self._factor = factor

    def execute(self):
        if not self.interpolateGrid():
            self._cb.ice_exception(Demo.RangeError())
            return
        self._cb.ice_response(self._grid)

    def interpolateGrid(self):
        # ...
```

If `interpolateGrid` returns `False`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

24.9 Summary

This chapter presented the server-side Python mapping. Because the mapping for Slice data types is identical for clients and servers, the server-side mapping only adds a few additional mechanism to the client side: a small API to initialize and finalize the run time, plus a few rules for how to derive servant classes from skeletons and how to register servants with the server-side run time.

Even though the examples in this chapter are very simple, they accurately reflect the basics of writing an Ice server. Of course, for more sophisticated servers (which we discuss in Chapter 32), you will be using additional APIs, for example, to improve performance or scalability. However, these APIs are all described in Slice, so, to use these APIs, you need not learn any Python mapping rules beyond those we described here.

Chapter 25
Developing a File System Server in Python

25.1 Chapter Overview

In this chapter, we present the source code for a fully-functional Python server that implements the file system we developed in Chapter 5 (see Chapter 23 for the corresponding client).

25.2 Implementing a File System Server

We have now seen enough of the server-side Python mapping to implement a server for the file system we developed in Chapter 5. (You may find it useful to review the Slice definition for our file system in Section 5.4 before studying the source code.)

Our server is implemented in a single source file, `Server.py`, containing our server's main program as well as the definitions of our `Directory` and `File` servant subclasses.

25.2.1 The Server Main Program

Our server main program uses the `Ice.Application` class we discussed in Section 24.3.1. The `run` method installs a signal handler, creates an object

adapter, instantiates a few servants for the directories and files in the file system, and then activates the adapter. This leads to a main program as follows:

```
import sys, threading, Ice, Filesystem

# DirectoryI servant class ...
# FileI servant class ...

class Server(Ice.Application):
    def run(self, args):
        # Terminate cleanly on receipt of a signal
        #
        self.shutdownOnInterrupt()

        # Create an object adapter (stored in the _adapter
        # static members)
        #
        adapter = self.communicator().\
                    createObjectAdapterWithEndpoints(
                        "SimpleFilesystem", "default -p 10000")
        DirectoryI._adapter = adapter
        FileI._adapter = adapter

        # Create the root directory (with name "/" and no parent)
        #
        root = DirectoryI("/", None)

        # Create a file called "README" in the root directory
        #
        file = FileI("README", root)
        text = [ "This file system contains a collection of " +
                 "poetry." ]
        try:
            file.write(text)
        except Filesystem.GenericError, e:
            print e.reason

        # Create a directory called "Coleridge"
        # in the root directory
        #
        coleridge = DirectoryI("Coleridge", root)

        # Create a file called "Kubla_Khan"
        # in the Coleridge directory
        #
        file = FileI("Kubla_Khan", coleridge)
```

```
        text = [ "In Xanadu did Kubla Khan",
                 "A stately pleasure-dome decree:",
                 "Where Alph, the sacred river, ran",
                 "Through caverns measureless to man",
                 "Down to a sunless sea." ]
        try:
            file.write(text)
        except Filesystem.GenericError, e:
            print e.reason

        # All objects are created, allow client requests now
        #
        adapter.activate()

        # Wait until we are done
        #
        self.communicator().waitForShutdown()

        if self.interrupted():
            print self.appName() + ": terminating"

        return 0

app = Server()
sys.exit(app.main(sys.argv))
```

The code defines the `Server` class, which derives from `Ice.Application` and contains the main application logic in its `run` method. Much of this code is boiler plate that we saw previously: we create an object adapter, and, towards the end, activate the object adapter and call `waitForShutdown`.

The interesting part of the code follows the adapter creation: here, the server instantiates a few nodes for our file system to create the structure shown in Figure 25.1.

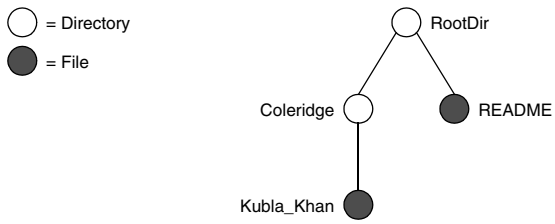


Figure 25.1. A small file system.

As we will see shortly, the servants for our directories and files are of type `DirectoryI` and `FileI`, respectively. The constructor for either type of servant accepts two parameters, the name of the directory or file to be created and a reference to the servant for the parent directory. (For the root directory, which has no parent, we pass `None`.) Thus, the statement

```
root = DirectoryI("/", None)
```

creates the root directory, with the name `"/"` and no parent directory.

Here is the code that establishes the structure in Figure 25.1:

```
# Create the root directory (with name "/" and no parent)
#
root = DirectoryI("/", None)

# Create a file called "README" in the root directory
#
file = FileI("README", root)
text = [ "This file system contains a collection of " +
         "poetry." ]
try:
    file.write(text)
except Filesystem.GenericError, e:
    print e.reason

# Create a directory called "Coleridge"
# in the root directory
#
coleridge = DirectoryI("Coleridge", root)

# Create a file called "Kubla_Khan"
# in the Coleridge directory
#
file = FileI("Kubla_Khan", coleridge)
text = [ "In Xanadu did Kubla Khan",
         "A stately pleasure-dome decree:",
         "Where Alph, the sacred river, ran",
         "Through caverns measureless to man",
         "Down to a sunless sea." ]
try:
    file.write(text)
except Filesystem.GenericError, e:
    print e.reason
```

We first create the root directory and a file `README` within the root directory. (Note that we pass a reference to the root directory as the parent when we create the new node of type `FileI`.)

The next step is to fill the file with text:

```
text = [ "This file system contains a collection of " +
         "poetry." ]
try:
    file.write(text)
except Filesystem.GenericError, e:
    print e.reason
```

Recall from Section 14.7.3 that Slice sequences map to Python lists. The Slice type `Lines` is simply a list of strings; we add a line of text to our `README` file by initializing the `text` list to contain one element.

Finally, we call the Slice `write` operation on our `FileI` servant by simply writing:

```
    file.write(text)
```

This statement is interesting: the server code invokes an operation on one of its own servants. Because the call happens via a reference to the servant (of type `FileI`) and *not* via a proxy (of type `FilePrx`), the Ice run time does not know that this call is even taking place—such a direct call into a servant is not mediated by the Ice run time in any way and is dispatched as an ordinary Python method call.

In similar fashion, the remainder of the code creates a subdirectory called `Coleridge` and, within that directory, a file called `Kubla_Khan` to complete the structure in Figure 25.1.

25.2.2 The `FileI` Servant Class

Our `FileI` servant class has the following basic structure:

```
class FileI(Filesystem.File):
    # Constructor and operations here...

    _adapter = None
```

The class has a number of data members:

- `_adapter`

  This class member stores a reference to the single object adapter we use in our server.

- `_name`

  This instance member stores the name of the file incarnated by the servant.

- `_parent`

  This instance member stores the reference to the servant for the file's parent directory.

- `_lines`

  This instance member holds the contents of the file.

The `_name`, `_parent`, and `_lines` data members are initialized by the constructor:

```
def __init__(self, name, parent):
    self._name = name
    self._parent = parent
    self._lines = []

    assert(self._parent != None)

    # Create an identity
    #
    myID = Ice.Identity()
    myID.name = Ice.generateUUID()

    # Add the identity to the object adapter
    #
    self._adapter.add(self, myID)

    # Create a proxy for the new node and
    # add it as a child to the parent
    #
    thisNode = Filesystem.NodePrx.uncheckedCast(
                    self._adapter.createProxy(myID))
    self._parent.addChild(thisNode)
```

After initializing the instance members, the code verifies that the reference to the parent is not `None` because every file must have a parent directory. The constructor then generates an identity for the file by calling `Ice.generateUUID` and adds itself to the servant map by calling `ObjectAdapter.add`. Finally, the constructor creates a proxy for this file and calls the `addChild` method on its parent directory. `addChild` is a helper function that a child directory or file calls to add itself to the list of descendant nodes of its parent directory. We will see the implementation of this function on page 748.

The remaining methods of the `FileI` class implement the Slice operations we defined in the `Node` and `File` Slice interfaces:

```
# Slice Node::name() operation

def name(self, current=None):
    return self._name

# Slice File::read() operation

def read(self, current=None):
    return self._lines

# Slice File::write() operation

def write(self, text, current=None):
    self._lines = text
```

The `name` method is inherited from the generated `Node` class. It simply returns the value of the `_name` instance member.

The `read` and `write` methods are inherited from the generated `File` class and simply return and set the `_lines` instance member.

25.2.3 The DirectoryI Servant Class

The `DirectoryI` class has the following basic structure:

```
class DirectoryI(Filesystem.Directory):
    # Constructor and operations here...

    _adapter = None
```

As for the `FileI` class, we have data members to store the object adapter, the name, and the parent directory. (For the root directory, the `_parent` member holds `None`.) In addition, we have a `_contents` data member that stores the list of child directories. These data members are initialized by the constructor:

```
def __init__(self, name, parent):
    self._name = name
    self._parent = parent
    self._contents = []

    # Create an identity. The
    # parent has the fixed identity "RootDir"
    #
    myID = Ice.Identity()
```

```
        if(self._parent):
            myID.name = Ice.generateUUID()
        else:
            myID.name = "RootDir"

        # Add the identity to the object adapter
        #
        self._adapter.add(self, myID)

        # Create a proxy for the new node and
        # add it as a child to the parent
        #
        thisNode = Filesystem.NodePrx.uncheckedCast(
                        self._adapter.createProxy(myID))
        if self._parent:
            self._parent.addChild(thisNode)
```

The constructor creates an identity for the new directory by calling `Ice.generateUUID`. (For the root directory, we use the fixed identity `"RootDir"`.) The servant adds itself to the servant map by calling `ObjectAdapter.add` and then creates a proxy to itself and passes it to the `addChild` helper function.

`addChild` simply adds the passed reference to the `_contents` list:

```
def addChild(self, child):
    self._contents.append(child)
```

The remainder of the operations, `name` and `list`, are trivial:

```
def name(self, current=None):
    return self._name

def list(self, current=None):
    return self._contents
```

25.3 Thread Safety

The server code we developed in Section 25.2 is not quite correct as it stands: if two clients access the same file in parallel, each via a different thread, one thread may read the `_lines` data member while another thread updates it. Obviously, if that happens, we may write or return garbage or, worse, crash the server. However, we can make the `read` and `write` operations thread-safe with a few trivial changes to the `FileI` class:

```
def __init__(self, name, parent):
    self._name = name
    self._parent = parent
    self._lines = []
    self._mutex = threading.Lock()

    # ...

def name(self, current=None):
    return self._name

def read(self, current=None):
    self._mutex.acquire()
    lines = self._lines[:] # Copy the list
    self._mutex.release()
    return lines

def write(self, text, current=None):
    self._mutex.acquire()
    self._lines = text
    self._mutex.release()
```

We modified the constructor to add the instance member `_mutex`, and then enclosed our `read` and `write` implementations in a critical section. (The `name` method does not require a critical section because the file's name is immutable.)

No changes for thread safety are necessary in the `DirectoryI` class because the `Directory` interface, in its current form, defines no operations that modify the object.

25.4 Summary

This chapter showed how to implement a complete server for the file system we defined in Chapter 5. Note that the server is remarkably free of code that relates to distribution: most of the server code is simply application logic that would be present just the same as a non-distributed version. Again, this is one of the major advantages of Ice: distribution concerns are kept away from application code so that you can concentrate on developing application logic instead of networking infrastructure.

Part VIII

Ruby Mapping

Chapter 26
Client-Side Slice-to-Ruby Mapping

26.1 Chapter Overview

In this chapter, we present the client-side Slice-to-Ruby mapping. One part of the client-side Ruby mapping concerns itself with rules for representing each Slice data type as a corresponding Ruby type; we cover these rules in Section 26.3 to Section 26.10. Another part of the mapping deals with how clients can invoke operations, pass and receive parameters, and handle exceptions. These topics are covered in Section 26.11 to Section 26.13. Slice classes have the characteristics of both data types and interfaces and are covered in Section 26.14. Code generation issues are discussed in Section 26.15, while Section 26.17 addresses the use of Slice checksums.

26.2 Introduction

The client-side Slice-to-Ruby mapping defines how Slice data types are translated to Ruby types, and how clients invoke operations, pass parameters, and handle errors. Much of the Ruby mapping is intuitive. For example, Slice sequences map to Ruby arrays, so there is essentially nothing new you have to learn in order to use Slice sequences in Ruby.

The Ruby API to the Ice run time is fully thread-safe. Obviously, you must still synchronize access to data from different threads. For example, if you have two threads sharing a sequence, you cannot safely have one thread insert into the sequence while another thread is iterating over the sequence. However, you only need to concern yourself with concurrent access to your own data—the Ice run time itself is fully thread safe, and none of the Ice API calls require you to acquire or release a lock before you safely can make the call.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least Section 26.11 to Section 26.13 in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

A word of advice before you start: in order to use the Ruby mapping, you should need no more than the Slice definition of your application and knowledge of the Ruby mapping rules. In particular, looking through the generated code in order to discern how to use the Ruby mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

26.3 Mapping for Identifiers

Slice identifiers map to an identical Ruby identifier. For example, the Slice identifier `Clock` becomes the Ruby identifier `Clock`. There are two exceptions to this rule:

1. If a Slice identifier maps to the name of a Ruby class, module, or constant, and the Slice identifier does not begin with an upper case letter, the mapping replaces the leading character with its upper case equivalent.[1] For example, the Slice identifier `bankAccount` is mapped as `BankAccount`.
2. If a Slice identifier is the same as a Ruby keyword, the corresponding Ruby identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.[2]

1. Ruby requires the names of classes, modules, and constants to begin with an upper case letter.
2. As suggested in Section 4.5.3 on page 92, you should try to avoid such identifiers as much as possible.

26.4 Mapping for Modules

Slice modules map to Ruby modules with the same name as the Slice module. The mapping preserves the nesting of the Slice definitions.

26.5 The `Ice` Module

All of the APIs for the Ice run time are nested in the `Ice` module, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` module are generated from Slice definitions; other parts of the `Ice` module provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` module throughout the remainder of the book.

A Ruby application can load the Ice run time using the `require` statement:

```
require 'Ice'
```

If the statement executes without error, the Ice run time is loaded and available for use. You can determine the version of the Ice run time you have just loaded by calling the `stringVersion` function:

```
icever = Ice::stringVersion()
```

26.6 Mapping for Simple Built-In Types

The Slice built-in types are mapped to Ruby types as shown in Table 26.1.

Table 26.1. Mapping of Slice built-in types to Ruby.

Slice	Ruby
`bool`	`true` or `false`
`byte`	`Fixnum`
`short`	`Fixnum`
`int`	`Fixnum` or `Bignum`

Table 26.1. Mapping of Slice built-in types to Ruby.

Slice	Ruby
`long`	`Fixnum` or `Bignum`
`float`	`Float`
`double`	`Float`
`string`	`String`

Although Ruby supports arbitrary precision in its integer types, the Ice run time validates integer values to ensure they have valid ranges for their declared Slice types.

26.6.1 String Mapping

String values returned as the result of a Slice operation (including return values, out parameters, and data members) contain UTF-8 encoded strings unless the program has installed a string converter, in which case string values use the converter's native encoding instead. See Section 32.24 for more information on string converters.

As string input values for a remote Slice operation, Ice accepts `nil` in addition to `String` objects; each occurrence of `nil` is marshaled as an empty string. Ice assumes that all `String` objects contain valid UTF-8 encoded strings unless the program has installed a string converter, in which case Ice assumes that `String` objects use the native encoding expected by the converter.

26.7 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

26.7.1 Mapping for Enumerations

Ruby does not have an enumerated type, so the Slice enumerations are emulated using a Ruby class: the name of the Slice enumeration becomes the name of the Ruby class; for each enumerator, the class contains a constant with the same name as the enumerator (see Section 26.3 for more information on identifiers). For example:

```
enum Fruit { Apple, Pear, Orange };
```

The generated Ruby class looks as follows:

```
class Fruit
    include Comparable

    Apple = # ...
    Pear = # ...
    Orange = # ...

    def Fruit.from_int(val)

    def to_i

    def to_s

    def <=>(other)

    def hash

    # ...
end
```

The compiler generates a class constant for each enumerator that holds a corresponding instance of `Fruit`. The `from_int` class method returns an instance given its integer value, while `to_i` returns the integer value of an enumerator and `to_s` returns its Slice identifier. The comparison operators are available as a result of including `Comparable`, which means a program can compare enumerators according to their integer values.

Given the above definitions, we can use enumerated values as follows:

```
f1 = Fruit::Apple
f2 = Fruit::Orange

if f1 == Fruit::Apple   # Compare for equality
    # ...
```

```
if f1 < f2              # Compare two enums
    # ...

case f2
when Fruit::Orange
    puts "found Orange"
else
    puts "found #{f2.to_s}"
end
```

As you can see, the generated class enables natural use of enumerated values.

26.7.2 Mapping for Structures

Slice structures map to Ruby classes with the same name. For each Slice data member, the Ruby class contains a corresponding instance variable as well as accessors to read and write its value. For example, here is our `Employee` structure from Section 4.9.4 once more:

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The Ruby mapping generates the following definition for this structure:

```
class Employee
    def initialize(number=0, firstName='', lastName='')
        @number = number
        @firstName = firstName
        @lastName = lastName
    end

    def hash
        # ...
    end

    def ==
        # ...
    end

    def inspect
        # ...
```

```
    end

    attr_accessor :number, :firstName, :lastName
end
```

The constructor initializes each of the instance variables to a default value appropriate for its type. You can also declare different default values for members of primitive and enumerated types, as discussed in Section 4.9.2.

The compiler generates a definition for the `hash` method, which allows instances to be used as keys in a hash collection. The `hash` method returns a hash value for the structure based on the value of its data members.

The == method returns true if all members of two structures are (recursively) equal.

The `inspect` method returns a string representation of the structure.

26.7.3 Mapping for Sequences

Slice sequences map to Ruby arrays; the only exception is a sequence of bytes, which maps to a string (see below). The use of a Ruby array means that the mapping does not generate a separate named type for a Slice sequence. It also means that you can take advantage of all the array functionality provided by Ruby. For example:

```
sequence<Fruit> FruitPlatter;
```

We can use the `FruitPlatter` sequence as shown below:

```
platter = [ Fruit::Apple, Fruit::Pear ]
platter.push(Fruit::Orange)
```

The Ice run time validates the elements of a sequence to ensure that they are compatible with the declared type; a `TypeError` exception is raised if an incompatible type is encountered.

Mapping for Byte Sequences

A Ruby string can contain arbitrary 8-bit binary data, therefore it is a more efficient representation of a byte sequence than a Ruby array in both memory utilization and throughput performance.

When receiving a byte sequence (as the result of an operation, as an out parameter, or as a member of a data structure), the value is always represented as a string. When sending a byte sequence as an operation parameter or data member,

the Ice run time accepts both a string and an array of integers as legal values. For example, consider the following Slice definitions:

```
// Slice
sequence<byte> Data;

interface I {
    void sendData(Data d);
    Data getData();
};
```

The interpreter session below uses these Slice definitions to demonstrate the mapping for a sequence of bytes:

```
> proxy = ...
> proxy.sendData("\0\1\2\3")   # Send as a string
> proxy.sendData([0, 1, 2, 3]) # Send as an array
> d = proxy.getData()
> d.class
=> String
> d
=> "\000\001\002\003"
```

The two invocations of `sendData` are equivalent; however, the second invocation incurs additional overhead as the Ice run time validates the type and range of each array element.

26.7.4 Mapping for Dictionaries

Here is the definition of our `EmployeeMap` from Section 4.9.4 once more:

```
dictionary<long, Employee> EmployeeMap;
```

As for sequences, the Ruby mapping does not create a separate named type for this definition. Instead, *all* dictionaries are simply instances of Ruby's hash collection type. For example:

```
em = {}

e = Employee.new
e.number = 31
e.firstName = "James"
e.lastName = "Gosling"

em[e.number] = e
```

The Ice run time validates the elements of a dictionary to ensure that they are compatible with the declared type; a `TypeError` exception is raised if an incompatible type is encountered.

26.8 Mapping for Constants

Here are the constant definitions we saw in Section 4.9.5 on page 103 once more:

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

The generated definitions for these constants are shown below:

```
AppendByDefault = true
LowerNibble = 15
Advice = "Don't Panic!"
TheAnswer = 42
PI = 3.1416
FavoriteFruit = Fruit::Pear
```

As you can see, each Slice constant is mapped to a Ruby constant with the same name.

26.9 Mapping for Exceptions

The mapping for exceptions is based on the inheritance hierarchy shown in Figure 26.1

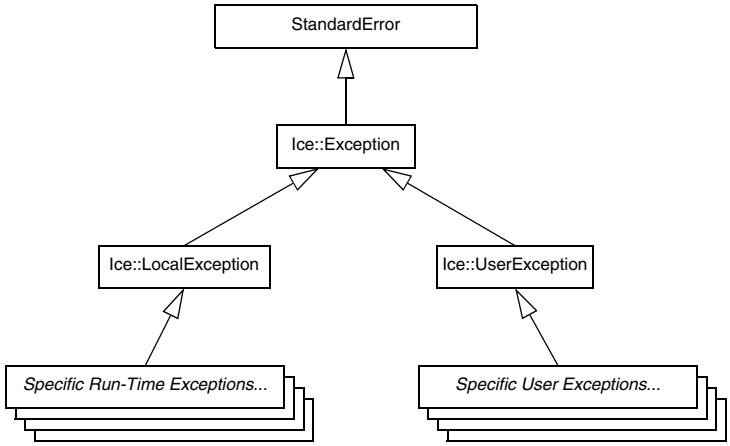


Figure 26.1. Inheritance structure for Ice exceptions.

The ancestor of all exceptions is `StandardError`, from which `Ice::Exception` is derived. `Ice::LocalException` and `Ice::UserException` are derived from `Ice::Exception` and form the base for all run-time and user exceptions.

Here is a fragment of the Slice definition for our world time server from Section 4.10.5 on page 120 once more:

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

These exception definitions map to the abbreviated Ruby class definitions shown below:

```
class GenericError < Ice::UserException
    def initialize(reason='')

    def to_s
```

```
    def inspect

    attr_accessor :reason
end

class BadTimeVal < GenericError
    def initialize(reason='')

    def to_s

    def inspect
end

class BadZoneName < GenericError
    def initialize(reason='')

    def to_s

    def inspect
end
```

Each Slice exception is mapped to a Ruby class with the same name. The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception member corresponds to an instance variable of the instance, which the constructor initializes to a default value appropriate for its type. You can also declare different default values for members of primitive and enumerated types, as discussed in Section 4.10.2. Accessors are provided to read and write the data members.

Although `BadTimeVal` and `BadZoneName` do not declare data members, their constructors still accept a value for the inherited data member `reason` in order to pass it to the constructor of the base exception `GenericError`.

Each exception also defines the standard methods `to_s` and `inspect` to return the name of the exception and a stringified representation of the exception and its members, respectively.

All user exceptions are derived from the base class `Ice::UserException`. This allows you to catch all user exceptions generically by installing a handler for `Ice::UserException`. Similarly, you can catch all Ice run-time exceptions with a handler for `Ice::LocalException`, and you can catch all Ice exceptions with a handler for `Ice::Exception`.

26.10 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice::LocalException` (which, in turn, derives from `Ice::Exception`).

An inheritance diagram for user and run-time exceptions appears in Figure 4.4 on page 117. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice::LocalException`

  This is the root of the inheritance tree for run-time exceptions.

- `Ice::UserException`

  This is the root of the inheritance tree for user exceptions.

- `Ice::TimeoutException`

  This is the base exception for both operation-invocation and connection-establishment timeouts.

- `Ice::ConnectTimeoutException`

  This exception is raised when the initial attempt to establish a connection to a server times out.

You will probably have little need to catch the remaining run-time exceptions; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. However, there is one exception you will probably be interested in specifically: `Ice::ObjectNotExistException`. This exception is raised if a client invokes an operation on an Ice object that no longer exists. In other words, the client holds a dangling reference to an object that probably existed some time in the past but has since been permanently destroyed.

26.11 Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a method on a local class instance that represents the remote object. This makes the mapping easy and intuitive to use because, for all intents and purposes (apart from error semantics), making a remote procedure call is no different from making a local procedure call.

26.11.1 Proxy Classes

On the client side, Slice interfaces map to Ruby classes with methods that correspond to the operations on those interfaces. Consider the following simple interface:

```
interface Simple {
    void op();
};
```

The Ruby mapping generates the following definition for use by the client:

```
class SimplePrx < Ice::ObjectPrx
    def op(_ctx=nil)
        # ...

    # ...
end
```

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice::ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a method of the same name. In the preceding example, we find that the operation `op` has been mapped to the method `op`. Note that `op` accepts an optional trailing parameter `_ctx` representing the operation context. This parameter is a Ruby hash value for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the context parameter in detail in Chapter 32. The parameter is also used by IceStorm—see Chapter 44.)

Proxy instances are always created on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

A value of `nil` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

26.11.2 The `Ice::ObjectPrx` Class

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice::ObjectPrx`. `ObjectPrx` provides a number of methods:

```
class ObjectPrx
    def eql?(proxy)
    def ice_getIdentity
    def ice_isA(id)
    def ice_id
    def ice_ping
    # ...
end
```

The methods behave as follows:

- `eql?`

  The implementation of this standard method compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `eql?` returns `false` even though the proxies denote the same object.

- `ice_getIdentity`

  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  ```
  module Ice {
      struct Identity {
          string name;
          string category;
      };
  };
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

  ```
  proxy1 = ...
  proxy2 = ...
  id1 = proxy1.ice_getIdentity
  id2 = proxy2.ice_getIdentity

  if id1 == id2
      # proxy1 and proxy2 denote the same object
  else
      # proxy1 and proxy2 denote different objects
  end
  ```

- `ice_isA`

  This method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID (see Section 4.13). For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

  ```
  proxy = ...
  if proxy && proxy.ice_isA("::Printer")
      # proxy denotes a Printer object
  else
      # proxy denotes some other type of object
  end
  ```

  Note that we are testing whether the proxy is `nil` before attempting to invoke the `ice_isA` method. This avoids getting a run-time error if the proxy is `nil`.

- `ice_id`

  This method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type `BasePrx`, with a static type ID of `::Base`, the return value of `ice_id` might be `"::Base"`, or it might be something more derived, such as `"::Derived"`.

- `ice_ping`

  This method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as `ObjectNotExist-Exception` or `ConnectTimeoutException`.

Note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call. (We discuss these methods in Chapter 32.)

26.11.3 Casting Proxies

The Ruby mapping for a proxy also generates two class methods:

```
class SimplePrx < Ice::ObjectPrx
    # ...

    def SimplePrx.checkedCast(proxy, facet='', ctx={})

    def SimplePrx.uncheckedCast(proxy, facet='')
end
```

Both the `checkedCast` and `uncheckedCast` methods implement a downcast: if the passed proxy is a proxy for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is `nil`), the cast returns `nil`.

The method names `checkedCast` and `uncheckedCast` are reserved for use in proxies. If a Slice interface defines an operation with either of those names, the mapping escapes the name in the generated proxy by prepending an underscore. For example, an interface that defines an operation named `checkedCast` is mapped to a proxy with a method named `_checkedCast`.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

```
obj = ...       # Get a proxy from somewhere...

simple = SimplePrx::checkedCast(obj)
if simple
    # Object supports the Simple interface...
else
    # Object is not of type Simple...
end
```

Note that a `checkedCast` contacts the server. This is necessary because only the server implementation has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`.

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but

different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather non-sensical things. To illustrate this, consider the following two interfaces:

```
interface Process {
    void launch(int stackSize, int dataSize);
};

// ...

interface Rocket {
    void launch(float xCoord, float yCoord);
};
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

```
obj = ...                                # Get proxy...
process = ProcessPrx::uncheckedCast(obj) # No worries...
process.launch(40, 60)                   # Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

26.11.4 Using Proxy Methods

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy (see Section 32.11). Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second timeout as shown below:

```
proxy = communicator.stringToProxy(...)
proxy = proxy.ice_timeout(10000)
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method. The example below demonstrates these semantics:

```
base = communicator.stringToProxy(...)
hello = Demo::HelloPrx::checkedCast(base)
hello = hello.ice_timeout(10000) # Type is not discarded
hello.sayHello()
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

26.11.5 Object Identity and Proxy Comparison

Proxy objects support comparison using the comparison operators `==`, `!=`, and `<=>`, as well as the `eql?` method. Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if p1 != p2
    # p1 and p2 denote different objects       # WRONG!
else
    # p1 and p2 denote the same object         # Correct
end
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each uses a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare

unequal, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function in the `Ice` module:

```
def proxyIdentityCompare(lhs, rhs)
def proxyIdentityAndFacetCompare(lhs, rhs)
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

```
p1 = ...        # Get a proxy...
p2 = ...        # Get another proxy...

if Ice.proxyIdentityCompare(p1, p2) != 0
    # p1 and p2 denote different objects       # Correct
else
    # p1 and p2 denote the same object         # Correct
end
```

The function returns 0 if the identities are equal, –1 if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major sort key and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name (see Chapter 33).

26.12 Mapping for Operations

As we saw in Section 26.11, for each operation on an interface, the proxy class contains a corresponding method with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system from Section 5.4:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

```
node = ...          # Initialize proxy
name = node.name()  # Get name via RPC
```

26.12.1 Normal and `idempotent` Operations

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

```
interface Example {
                string op1();
    idempotent  string op2();
};
```

The proxy class for this is:

```
class ExamplePrx < Ice::ObjectPrx
    def op1(_ctx=nil)

    def op2(_ctx=nil)
end
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to look the same.

26.12.2 Passing Parameters

In Parameters

All parameters are passed by reference in the Ruby mapping; it is guaranteed that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
```

```
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following proxy for this definition:

```
class ClientToServerPrx < Ice::ObjectPrx
    def op1(i, f, b, s, _ctx=nil)

    def op2(ns, ss, st, _ctx=nil)

    def op3(proxy, _ctx=nil)
end
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

```
p = ...                                  # Get proxy...

p.op1(42, 3.14, true, "Hello world!")    # Pass simple literals

i = 42
f = 3.14
b = true
s = "Hello world!"
p.op1(i, f, b, s)                        # Pass simple variables

ns = NumberAndString.new()
ns.x = 42
ns.str = "The Answer"
ss = [ "Hello world!" ]
st = {}
st[0] = ns
p.op2(ns, ss, st)                        # Pass complex variables

p.op3(p)                                 # Pass proxy
```

Out Parameters

As in Java, Ruby functions do not support reference arguments. That is, it is not possible to pass an uninitialized variable to a Ruby function in order to have its value initialized by the function. The Java mapping (see Section 10.12.2) overcomes this limitation with the use of "holder classes" that represent each `out` parameter. The Ruby mapping takes a different approach, one that is more natural for Ruby users.

The semantics of `out` parameters in the Ruby mapping depend on whether the operation returns one value or multiple values. An operation returns multiple values when it has declared multiple `out` parameters, or when it has declared a non-`void` return type and at least one `out` parameter.

If an operation returns multiple values, the client receives them in the form of a *result array*. A non-`void` return value, if any, is always the first element in the result array, followed by the `out` parameters in the order of declaration.

If an operation returns only one value, the client receives the value itself.

Here is the same Slice definition we saw on page 772 once more, but this time with all parameters being passed in the `out` direction:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    int op1(out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Ruby mapping generates the following code for this definition:

```
class ClientToServerPrx < Ice::ObjectPrx
    def op1(_ctx=nil)

    def op2(_ctx=nil)

    def op3(_ctx=nil)
end
```

Given a proxy to a `ServerToClient` interface, the client code can receive the results as in the following example:

```
p = ...                # Get proxy...
i, f, b, s = p.op1()
ns, ss, st = p.op2()
stcp = p.op3()
```

The operations have no `in` parameters, therefore no arguments are passed to the proxy methods. Since `op1` and `op2` return multiple values, their result arrays are unpacked into separate values, whereas the return value of `op3` requires no unpacking.

Parameter Type Mismatches

Although the Ruby compiler cannot check the types of arguments passed to a method, the Ice run time does perform validation on the arguments to a proxy invocation and reports any type mismatches as a `TypeError` exception.

Null Parameters

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `nil`, but the corresponding Slice types do not have the of a null value. To make life with these types easier, whenever you pass `nil` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid a run-time error. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, it makes no difference to the receiver whether you send a string as `nil` or as an empty string: either way, the receiver sees an empty string.

26.13 Exception Handling

Any operation invocation may throw a run-time exception (see Section 26.10) and, if the operation has an exception specification, may also throw user exceptions (see Section 26.9). Suppose we have the following simple interface:

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as Ruby exceptions, so you can simply enclose one or more operation invocations in a `begin–rescue` block:

```
child = ...        # Get child proxy...

begin
    child.askToCleanUp()
rescue Tantrum => t
    puts "The child says: #{t.reason}"
end
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will usually be handled by exception handlers higher in the hierarchy. For example:

```
def run()
    child = ...        # Get child proxy...
    begin
        child.askToCleanUp()
    rescue Tantrum => t
        puts "The child says: #{t.reason}"
        child.scold()  # Recover from error...
    end
    child.praise()     # Give positive feedback...
end

begin
    # ...
    run()
    # ...
rescue Ice::Exception => ex
    print ex.backtrace.join("\n")
end
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Chapter 3.)

26.14 Mapping for Classes

Slice classes are mapped to Ruby classes with the same name. For each Slice data member, the generated class contains an instance variable and accessors to read and write it, just as for structures and exceptions. Consider the following class definition:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Ruby mapping generates the following code for this definition:

```
module TimeOfDay_mixin
    include ::Ice::Object_mixin

    # ...

    def inspect
        # ...
    end

    #
    # Operation signatures.
    #
    # def format()

    attr_accessor :hours, :minutes, :seconds
end
class TimeOfDay
    include TimeOfDay_mixin

    def initialize(hour=0, minute=0, second=0)
        @hour = hour
        @minute = minute
        @second = second
    end

    def TimeOfDay.ice_staticId()
        '::M::TimeOfDay'
```

```
    end

    # ...
end
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` includes the mixin module `TimeOfDay_mixin`, which in turn includes `Ice::Object_mixin`. This reflects the semantics of Slice classes in that all classes implicitly inherit from `Object`, which is the ultimate ancestor of all classes. Note that `Object` is *not* the same as `Ice::ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The constructor defines an instance variable for each Slice data member.
3. The class defines the class method `ice_staticId`.
4. A comment summarizes the method signatures for each Slice operation.

We will discuss these items in the subsections below.

26.14.1 Inheritance from `Object`

In other language mappings, the inheritance relationship between `Object` and a user-defined Slice class is stated explicitly, in that the generated class derives from a language-specific representation of `Object`. Although its class type allows single inheritance, Ruby's loosely-typed nature places less emphasis on class hierarchies and relies more on *duck typing*.[3]

The Slice mapping for a class follows this convention by placing most of the necessary machinery in a mixin module that the generated class includes into its definition. The Ice run time requires an instance of a Slice class to include the mixin module and define values for the declared data members, but does not require that the object be an instance of the generated class.

3. In Ruby, an object's type is typically less important than the methods it supports. "If it looks like a duck, and acts like a duck, then it *is* a duck."

As shown in Figure 26.2, classes have no relationship to `Ice::ObjectPrx` (which is at the base of the inheritance hierarchy for proxies), therefore you cannot pass a class where a proxy is expected (and vice versa).

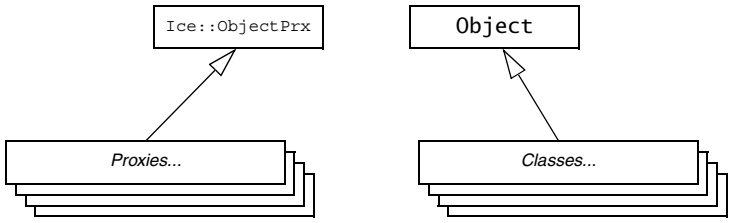


Figure 26.2. Inheritance from `Ice::ObjectPrx` and `Object`.

An instance of a Slice class `C` supports a number of methods:

```
def ice_isA(id, current=nil)

def ice_ping(current=nil)

def ice_ids(current=nil)

def ice_id(current=nil)

def C.ice_staticId()

def ice_preMarshal()

def ice_postUnmarshal()
```

The methods behave as follows:

- `ice_isA`

  This method returns `true` if the object supports the given type ID, and `false` otherwise.

- `ice_ping`

  As for interfaces, `ice_ping` provides a basic reachability test for the class.

- `ice_ids`

  This method returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.

- `ice_id`

  This method returns the actual run-time type ID of the object. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.

- `ice_staticId`

  This method returns the static type ID of the class.

- `ice_preMarshal`

  If the object supports this method, the Ice run time invokes it just prior to marshaling the object's state, providing the opportunity for the object to validate its declared data members.

- `ice_postUnmarshal`

  If the object supports this method, the Ice run time invokes it after unmarshaling the object's state. An object typically defines this method when it needs to perform additional initialization using the values of its declared data members.

The mixin module `Ice::Object_mixin` supplies default definitions of `ice_isA` and `ice_ping`. For each Slice class, the generated mixin module defines `ice_ids` and `ice_id`, and the generated class defines the `ice_staticId` method.

Note that neither `Ice::Object` nor the generated class override `hash` and `==`, so the default implementations apply.

26.14.2 Data Members of Classes

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding instance variable and accessor methods.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

```
class TimeOfDay {
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler produces the following generated code for this definition:

```
module TimeOfDay_mixin
    include ::Ice::Object_mixin

    # ...

    #
    # Operation signatures.
    #
    # def format()

    attr_accessor :hours, :minutes, :seconds
    protected :hours, :hours=
    protected :minutes, :minutes=
    protected :seconds, :seconds=
end
class TimeOfDay
    include TimeOfDay_mixin

    def initialize(hour=0, minute=0, second=0)
        @hour = hour
        @minute = minute
        @second = second
    end

    # ...
end
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

```
["protected"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

26.14.3 Class Constructors

Classes have a constructor that assigns to each data member a default value appropriate for its type. You can also declare different default values for data members of primitive and enumerated types, as discussed in Section 4.11.1.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

26.14.4 Operations of Classes

Operations of classes are mapped to methods in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), objects representing instances of `TimeOfDay` must define equivalent methods. For example:

```
class TimeOfDayI < TimeOfDay
    def format(current=nil)
        sprintf("%02d:%02d:%02d", @hour, @minute, @second)
    end
end
```

In this case our implementation class `TimeOfDayI` derives from the generated class `TimeOfDay`. An alternative is to include the generated mixin module, which makes it possible for the class to derive from a different base class if necessary:

```
class TimeOfDayI < SomeOtherClass
    include TimeOfDay_mixin

    def format(current=nil)
        sprintf("%02d:%02d:%02d", @hour, @minute, @second)
    end
end
```

As described in Section 26.14.1, an implementation of a Slice class must include the mixin module but is not required to derive from the generated class.

Ruby allows an existing class to be reopened in order to augment or replace its functionality. This feature provides another way for us to implement a Slice class: reopen the generated class and define the necessary methods:

```
class TimeOfDay
    def format(current=nil)
        sprintf("%02d:%02d:%02d", @hour, @minute, @second)
    end
end
```

As an added benefit, this strategy eliminates the need to define a class factory. The next section describes this subject in more detail.

A Slice class such as `TimeOfDay` that declares or inherits an operation is inherently abstract. Ruby does not support the notion of abstract classes or abstract methods, therefore the mapping merely summarizes the required method signatures in a comment for your convenience.

You may notice that the mapping for an operation adds an optional trailing parameter named `current`. For now, you can ignore this parameter and pretend it does not exist. (We look at it in more detail in Section 32.6.)

26.14.5 Receiving Objects

We have discussed the ways you can implement a Slice class, but we also need to examine the semantics of receiving an object as the return value or as an out-parameter from an operation invocation. Consider the following simple interface:

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. Unless we tell it otherwise, the Ice run time in Ruby does exactly that: it instantiates the generated class `TimeOfDay`. Although `TimeOfDay` is logically an abstract class because its Slice equivalent defined an operation, Ruby has no notion of abstract classes and therefore it is legal to create an instance of this class. Furthermore, there are situations in which this is exactly the behavior you want:

- when you have reopened the generated class to define its operations, or
- when your program uses only the data members of an object and does not invoke any of its operations.

On the other hand, if you have defined a Ruby class that implements the Slice class, you need the Ice run time to return an instance of your class and not an instance of the generated class. The Ice run time cannot magically know about

your implementation class, therefore you must inform the Ice run time by installing a class factory.

26.14.6 Class Factories

The Ice run time invokes a class factory when it needs to instantiate an object of a particular type. If no factory is found, the Ice run time instantiates the generated class as described in Section 26.14.5. To install a factory, we use operations provided by the `Ice::Communicator` interface:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must create an object that supports the `Ice::ObjectFactory` interface:

```
class ObjectFactory
    def create(type)
        fail unless type == M::TimeOfDay::ice_staticId()
        TimeOfDayI.new
    end

    def destroy
        # Nothing to do
    end
end
```

The object factory's `create` method is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` method is called by the Ice run time when its communicator is destroyed.

The `create` method is passed the type ID (see Section 4.13) of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::M::TimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instanti-

ates and returns a `TimeOfDayI` object. For other type IDs, the method fails because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoObjectFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

```
ic = ...   # Get Communicator...
ic.addObjectFactory(ObjectFactory.new,
                    M::TimeOfDay::ice_staticId())
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::M::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator—if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called. However, calls to `create` can be made concurrently.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class.

26.15 Code Generation

The Ruby mapping supports two forms of code generation: dynamic and static.

26.15.1 Dynamic Code Generation

Using dynamic code generation, Slice files are "loaded" at run time and dynamically translated into Ruby code, which is immediately compiled and available for use by the application. This is accomplished using the `Ice::loadSlice` method, as shown in the following example:

```
Ice::loadSlice("Color.ice")
puts "My favorite color is #{M::Color.blue.to_s}"
```

For this example, we assume that `Color.ice` contains the following definitions:

```
module M {
    enum Color { red, green, blue };
};
```

`Ice::loadSlice` Options

The `Ice::loadSlice` method behaves like a Slice compiler in that it accepts command-line arguments for specifying preprocessor options and controlling code generation. The arguments must include at least one Slice file.

The function has the following Ruby definition:

```
def loadSlice(cmd, args=[])
```

The command-line arguments can be specified entirely in the first argument, `cmd`, which must be a string. The optional second argument can be used to pass additional command-line arguments as a list; this is useful when the caller already has the arguments in list form. The function always returns `nil`.

For example, the following calls to `Ice::loadSlice` are functionally equivalent:

```
Ice::loadSlice("-I/opt/IceRuby/slice Color.ice")
Ice::loadSlice("-I/opt/IceRuby/slice", ["Color.ice"])
Ice::loadSlice("", ["-I/opt/IceRuby/slice", "Color.ice"])
```

In addition to the standard compiler options described in Section 4.20, `Ice::loadSlice` also supports the following command-line options:

- **`--all`**

  Generate code for all Slice definitions, including those from included files.

- **`--checksum`**

  Generate checksums for Slice definitions. See Section 26.17 for more information.

Locating Slice Files

If your Slice files depend on Ice types, you can avoid hard-coding the path name of your Ice installation directory by calling the `Ice::getSliceDir` function:

```
Ice::loadSlice("-I" + Ice::getSliceDir() + " Color.ice")
```

This function attempts to locate the `slice` subdirectory of your Ice installation using an algorithm that succeeds for the following scenarios:

- Installation of a binary Ice archive
- Installation of an Ice source distribution using **`make install`**
- Installation via a Windows installer
- RPM installation on Linux
- Execution inside a compiled Ice source distribution

If the `slice` subdirectory can be found, `getSliceDir` returns its absolute path name, otherwise the function returns `nil`.

Loading Multiple Files

You can specify as many Slice files as necessary in a single invocation of `Ice::loadSlice`, as shown below:

```
Ice::loadSlice("Syscall.ice Process.ice")
```

Alternatively, you can call `Ice::loadSlice` several times:

```
Ice::loadSlice("Syscall.ice")
Ice::loadSlice("Process.ice")
```

If a Slice file includes another file, the default behavior of `Ice::loadSlice` generates Ruby code only for the named file. For example, suppose `Syscall.ice` includes `Process.ice` as follows:

```
// Syscall.ice
#include <Process.ice>
...
```

If you call `Ice::loadSlice("-I. Syscall.ice")`, Ruby code is not generated for the Slice definitions in `Process.ice` or for any definitions that may be included by `Process.ice`. If you also need code to be generated for

included files, one solution is to load them individually in subsequent calls to `Ice::loadSlice`. However, it is much simpler, not to mention more efficient, to use the **`--all`** option instead:

```
Ice::loadSlice("--all -I. Syscall.ice")
```

When you specify **`--all`**, `Ice::loadSlice` generates Ruby code for all Slice definitions included directly or indirectly from the named Slice files.

There is no harm in loading a Slice file multiple times, aside from the additional overhead associated with code generation. For example, this situation could arise when you need to load multiple top-level Slice files that happen to include a common subset of nested files. Suppose that we need to load both `Syscall.ice` and `Kernel.ice`, both of which include `Process.ice`. The simplest way to load both files is with a single call to `Ice::loadSlice`:

```
Ice::loadSlice("--all -I. Syscall.ice Kernel.ice")
```

Although this invocation causes the Ice extension to generate code twice for `Process.ice`, the generated code is structured so that the interpreter ignores duplicate definitions. We could have avoided generating unnecessary code with the following sequence of steps:

```
Ice::loadSlice("--all -I. Syscall.ice")
Ice::loadSlice("-I. Kernel.ice")
```

In more complex cases, however, it can be difficult or impossible to completely avoid this situation, and the overhead of code generation is usually not significant enough to justify such an effort.

Limitations

The `Ice::loadSlice` method must be called outside of any module scope. For example, the following code is incorrect:

```
# WRONG
module M
    Ice::loadSlice("--all -I. Syscall.ice Kernel.ice")
    ...
end
```

26.15.2 Static Code Generation

You should be familiar with static code generation if you have used other Slice language mappings, such as C++ or Java. Using static code generation, the Slice

compiler **slice2rb** (see Section 26.15.4) generates Ruby code from your Slice definitions.

Compiler Output

For each Slice file `X.ice`, **slice2rb** generates Ruby code into a file named `X.rb` in the output directory. The default output directory is the current working directory, but a different directory can be specified using the **--output-dir** option.

Include Files

It is important to understand how **slice2rb** handles include files. In the absence of the **--all** option, the compiler does not generate Ruby code for Slice definitions in included files. Rather, the compiler translates Slice `#include` statements into Ruby `require` statements in the following manner:

1. Determine the full pathname of the included file.
2. Create the shortest possible relative pathname for the included file by iterating over each of the include directories (specified using the **-I** option) and removing the leading directory from the included file if possible.

   For example, if the full pathname of an included file is `/opt/App/slice/OS/Process.ice`, and we specified the options **-I/opt/App** and **-I/opt/App/slice**, then the shortest relative pathname is `OS/Process.ice` after removing `/opt/App/slice`.
3. Replace the `.ice` extension with `.rb`. Continuing our example from the previous step, the translated `require` statement becomes

   ```
   require "OS/Process.rb"
   ```

As a result, you can use **-I** options to tailor the `require` statements generated by the compiler in order to avoid absolute pathnames and match the organizational structure of your application's source files.

26.15.3 Static Versus Dynamic Code Generation

There are several issues to consider when evaluating your requirements for code generation.

Application Considerations

The requirements of your application generally dictate whether you should use dynamic or static code generation. Dynamic code generation is convenient for a number of reasons:

- It avoids the intermediate compilation step required by static code generation.
- It makes the application more compact because the application requires only the Slice files, not the additional files produced by static code generation.
- It reduces complexity, which is especially helpful during testing, or when writing short or transient programs.

Static code generation, on the other hand, is appropriate in many situations:

- when an application uses a large number of Slice definitions and the startup delay must be minimized
- when it is not feasible to deploy Slice files with the application
- when a number of applications share the same Slice files
- when Ruby code is required in order to utilize third-party Ruby tools.

Mixing Static and Dynamic Generation

You can safely use a combination of static and dynamic translation in an application. For it to work properly, you must correctly manage the include paths for Slice translation and the Ruby interpreter so that the statically-generated code can be imported properly by `require`.

For example, suppose you want to dynamically load the following Slice definitions:

```
#include <Glacier2/Session.ice>

module MyApp {
    interface MySession extends Glacier2::Session {
        // ...
    };
};
```

Whether the included file `Glacier2/Session.ice` is loaded dynamically or statically is determined by the presence of the **`--all`** option:

```
sliceDir = "-I#{ENV['ICE_HOME']}/slice"

# Load Glacier2/Session.ice dynamically:
Ice::loadSlice(sliceDir + " --all MySession.ice")

# Load Glacier2/Session.ice statically:
Ice::loadSlice(sliceDir + " MySession.ice")
```

In this example, the first invocation of `loadSlice` uses the **`--all`** option so that code is generated dynamically for all included files. The second invocation omits **`--all`**, therefore the Ruby interpreter executes the equivalent of the following statement:

```
require "Glacier2/Session.rb"
```

As a result, before we can call `loadSlice` we must first ensure that the interpreter can locate the statically-generated file `Glacier2/Session.rb`. We can do this in a number of ways, including

- adding the parent directory (e.g., `/opt/IceRuby/ruby`) to the **`RUBYLIB`** environment variable
- specifying the **`-I`** option when starting the interpreter
- modifying the search path at run time, as shown below:

  ```
  $:.unshift("/opt/IceRuby/ruby")
  ```

26.15.4 `slice2rb` Command-Line Options

The Slice-to-Ruby compiler, **`slice2rb`**, offers the following command-line options in addition to the standard options described in Section 4.20:

- **`--all`**

  Generate code for all Slice definitions, including those from included files.

- **`--checksum`**

  Generate checksums for Slice definitions.

26.16 The `main` Program

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. You must initialize the Ice run time by calling `Ice::initialize` before you can do anything else in your program.

`Ice::initialize` returns a reference to an instance of an `Ice:::Communicator`:

```
require 'Ice'

status = 0
ic = nil
begin
    ic = Ice::initialize(ARGV)
    # ...
rescue => ex
    puts ex
    status = 1
end

# ...
```

`Ice::initialize` accepts the argument list that is passed to the program by the operating system. The function scans the argument list for any command-line options that are relevant to the Ice run time; any such options are removed from the argument list so, when `Ice::initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

Before leaving your program, you *must* call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your program to terminate without calling `destroy` first; doing so has undefined behavior.

The general shape of our program is therefore as follows:

```
require 'Ice'

status = 0
ic = nil
begin
    ic = Ice::initialize(ARGV)
    # ...
rescue => ex
    puts ex
    status = 1
end

if ic
```

```
    begin
        ic.destroy()
    rescue => ex
        puts ex
        status = 1
    end
end

exit(status)
```

Note that the code places the call to `Ice::initialize` into a `begin` block and takes care to return the correct exit status to the operating system. Also note that an attempt to destroy the communicator is made only if the initialization succeeded.

26.16.1 The `Ice::Application` Class

The preceding program structure is so common that Ice offers a class, `Ice::Application`, that encapsulates all the correct initialization and finalization activities. The synopsis of the class is as follows (with some detail omitted for now):

```
module Ice
    class Application
        def main(args, configFile=nil, initData=nil)

        def run(args)

        def Application.appName()

        def Application.communicator()
    end
end
```

The intent of this class is that you specialize `Ice::Application` and implement the abstract `run` method in your derived class. Whatever code you would normally place in your main program goes into `run` instead. Using `Ice::Application`, our program looks as follows:

```
require 'Ice'

class Client < Ice::Application
    def run(args)
        # Client code here...
        return 0
```

```
    end
end

app = Client.new()
status = app.main(ARGV)
exit(status)
```

If you prefer, you can also reopen `Ice::Application` and define `run` directly:

```
require 'Ice'

class Ice::Application
    def run(args)
        # Client code here...
        return 0
    end
end

app = Ice::Application.new()
status = app.main(ARGV)
exit(status)
```

You also can call `main` with an optional file name or an `InitializationData` structure (see Section 32.3 and Section 30.9). If you pass a configuration file name to `main`, the settings in this file are overridden by settings in a file identified by the `ICE_CONFIG` environment variable (if defined). Property settings supplied on the command line take precedence over all other settings.

The `Application.main` function does the following:

1. It installs an exception handler. If your code fails to handle an exception, `Application.main` prints the exception information before returning with a non-zero return value.
2. It initializes (by calling `Ice::initialize`) and finalizes (by calling `Communicator.destroy`) a communicator. You can get access to the communicator for your program by calling the static `communicator` accessor.
3. It scans the argument list for options that are relevant to the Ice run time and removes any such options. The argument list that is passed to your `run` method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.
4. It provides the name of your application via the static `appName` member function. The return value from this call is the first element of the argument

vector passed to Application.main, so you can get at this name from anywhere in your code by calling Ice::Application::appName (which is usually required for error messages).

5. It installs a signal handler that properly shuts down the communicator.

Using Ice::Application ensures that your program properly finalizes the Ice run time, whether your program terminates normally or in response to an exception or signal. We recommend that all your programs use this class; doing so makes your life easier. In addition Ice::Application also provides features for signal handling and configuration that you do not have to implement yourself when you use this class.

Catching Signals

A program typically needs to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is particularly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

To make it easier to deal with signals, Ice::Application encapsulates Ruby's signal handling capabilities, allowing you to cleanly shut down on receipt of a signal:

```
class Application
    def Application.destroyOnInterrupt()

    def Application.ignoreInterrupt()

    def Application.callbackOnInterrupt()

    def Application.holdInterrupt()

    def Application.releaseInterrupt()

    def Application.interrupted()

    def interruptCallback(sig):
        # Default implementation does nothing.
    end
    # ...
end
```

The methods behave as follows:

- `destroyOnInterrupt`

  This method installs a signal handler that destroys the communicator if it is interrupted. This is the default behavior.

- `ignoreInterrupt`

  This method causes signals to be ignored.

- `callbackOnInterrupt`

  This function configures `Ice::Application` to invoke `interruptCallback` when a signal occurs, thereby giving the subclass responsibility for handling the signal.

- `holdInterrupt`

  This method temporarily blocks signal delivery.

- `releaseInterrupt`

  This method restores signal delivery to the previous disposition. Any signal that arrives after `holdInterrupt` was called is delivered when you call `releaseInterrupt`.

- `interrupted`

  This method returns `True` if a signal caused the communicator to shut down, `False` otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by a signal. This is useful, for example, for logging purposes.

- `interruptCallback`

  A subclass implements this function to respond to signals. The function may be called concurrently with any other thread and must not raise exceptions.

By default, `Ice::Application` behaves as if `destroyOnInterrupt` was invoked, therefore our program requires no change to ensure that the program terminates cleanly on receipt of a signal. (You can disable the signal-handling functionality of `Ice::Application` by passing the constant `NoSignalHandling` to the constructor. In that case, signals retain their default behavior, that is, terminate the process.) However, we add a diagnostic to report the occurrence of a signal, so our program now looks like:

```
require 'Ice'

class MyApplication < Ice::Application
    def run(args)
        # Client code here...
```

```
        if Ice::Application::interrupted()
            print Ice::Application::appName() + ": terminating"
        end

        return 0
    end
end

app = MyApplication.new()
status = app.main(ARGV)
exit(status)
```

`Ice::Application` and Properties

Apart from the functionality shown in this section, `Ice::Application` also takes care of initializing the Ice run time with property values. Properties allow you to configure the run time in various ways. For example, you can use properties to control things such as the thread pool size or the trace level for diagnostic output. The `main` method of `Ice::Application` accepts an optional second parameter allowing you to specify the name of a configuration file that will be processed during initialization. We discuss Ice properties in more detail in Chapter 30.

Limitations of `Ice::Application`

`Ice::Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use `Ice::Application`. Instead, you must structure your code as we saw in Chapter 3 (taking care to always destroy the communicator).

26.17 Using Slice Checksums

As described in Section 4.21, the Slice compilers can optionally generate checksums of Slice definitions. For **`slice2rb`**, the **`--checksum`** option causes the compiler to generate code that adds checksums to the hash collection `Ice::SliceChecksums`. The checksums are installed automatically when the Ruby code is first parsed; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the two hash objects using a comparison operator. However, this is not feasible if it is possible that the server might return a superset of the client's checksums. A more general solution is to iterate over the local checksums as demonstrated below:

```
serverChecksums = ...
for i in Ice::SliceChecksums.keys
    if not serverChecksums.has_key?(i)
        # No match found for type id!
    elif Ice::SliceChecksums[i] != serverChecksums[i]
        # Checksum mismatch!
    end
end
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

Chapter 27
Developing a File System Client in Ruby

27.1 Chapter Overview

In this chapter, we present the source code for a Ruby client that accesses the file system we developed in Chapter 5.

27.2 The Ruby Client

We now have seen enough of the client-side Ruby mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
```

```
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

```
require 'Filesystem.rb'

# Recursively print the contents of directory "dir"
# in tree fashion. For files, show the contents of
# each file. The "depth" parameter is the current
# nesting level (for indentation).

def listRecursive(dir, depth)
    indent = ''
    depth = depth + 1
    for i in (0...depth)
        indent += "\t"
    end

    contents = dir.list()

    for node in contents
        subdir = Filesystem::DirectoryPrx::checkedCast(node)
        file = Filesystem::FilePrx::uncheckedCast(node)
        print indent + node.name() + " "
        if subdir
            puts "(directory):"
            listRecursive(subdir, depth)
        else
            puts "(file):"
            text = file.read()
            for line in text
                puts indent + "\t" + line
```

```
            end
        end
    end
end

status = 0
ic = nil
begin
    # Create a communicator
    #
    ic = Ice::initialize(ARGV)

    # Create a proxy for the root directory
    #
    obj = ic.stringToProxy("RootDir:default -p 10000")

    # Down-cast the proxy to a Directory proxy
    #
    rootDir = Filesystem::DirectoryPrx::checkedCast(obj)

    # Recursively list the contents of the root directory
    #
    puts "Contents of root directory:"
    listRecursive(rootDir, 0)
rescue => ex
    puts ex
    print ex.backtrace.join("\n")
    status = 1
end

if ic
    # Clean up
    #
    begin
        ic.destroy()
    rescue => ex
        puts ex
        print ex.backtrace.join("\n")
        status = 1
    end
end

exit(status)
```

The program first defines the `listRecursive` function, which is a helper function to print the contents of the file system, and the main program follows. Let us look at the main program first:

1. The structure of the code follows what we saw in Chapter 3. After initializing the run time, the client creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.

2. The client down-casts the proxy to `DirectoryPrx` and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code does a `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, as well as an `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the `DirectoryPrx` returned by the `checkedCast`; if the `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, an `uncheckedCast` is sufficient to get a `FilePrx`.

   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use an `uncheckedCast` instead of a `checkedCast` because an `uncheckedCast` does not incur any network traffic.

2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.

3. The code checks the type of the node:

   - If it is a directory, the code recurses, incrementing the indent level.

   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of a two files and a a directory as follows:

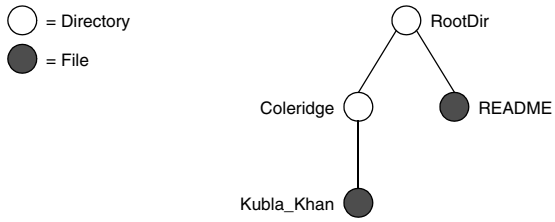


Figure 27.1. A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client is not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in Chapter 38 and Chapter 34.

27.3 Summary

This chapter presented a very simple client to access a server that implements the file system we developed in Chapter 5. As you can see, the Ruby code hardly differs from the code you would write for an ordinary Ruby program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local Ruby object. This allows you to put your effort where

you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs.

Part IX

PHP Mapping

Chapter 28
Client-Side Slice-to-PHP Mapping

28.1 Chapter Overview

In this chapter, we present the client-side Slice-to-PHP mapping. One part of the client-side PHP mapping concerns itself with rules for representing each Slice data type as a corresponding PHP type; we cover these rules in Section 28.3 to Section 28.10. Another part of the mapping deals with how clients can invoke operations, pass and receive parameters, and handle exceptions. These topics are covered in Section 28.11 to Section 28.13. Slice classes have the characteristics of both data types and interfaces and are covered in Section 28.14. Code generation issues are discussed in Section 28.15 and PHP-specific features in Section 28.16. Finally, Section 28.17 addresses the use of Slice checksums.

28.2 Introduction

The client-side Slice-to-PHP mapping defines how Slice data types are translated to PHP types, and how clients invoke operations, pass parameters, and handle errors. Much of the PHP mapping is intuitive. For example, Slice sequences map to PHP arrays, so there is essentially nothing new you have to learn in order to use Slice sequences in PHP.

Much of what appears in this chapter is reference material. We suggest that you skim the material on the initial reading and refer back to specific sections as needed. However, we recommend that you read at least Section 28.11 to Section 28.13 in detail because these sections cover how to call operations from a client, pass parameters, and handle exceptions.

A word of advice before you start: in order to use the PHP mapping, you should need no more than the Slice definition of your application and knowledge of the PHP mapping rules. In particular, looking through the generated code in order to discern how to use the PHP mapping is likely to be inefficient, due to the amount of detail. Of course, occasionally, you may want to refer to the generated code to confirm a detail of the mapping, but we recommend that you otherwise use the material presented here to see how to write your client-side code.

28.3 Mapping for Identifiers

Slice identifiers map to an identical PHP identifier. For example, the Slice identifier `Clock` becomes the PHP identifier `Clock`. There is one exception to this rule: if a Slice identifier is the same as a PHP keyword or is an identifier reserved by the Ice run time (such as `checkedCast`), the corresponding PHP identifier is prefixed with an underscore. For example, the Slice identifier `while` is mapped as `_while`.[1]

A single Slice identifier often results in several PHP identifiers. For example, for a Slice interface named `Foo`, the generated PHP code uses the identifiers `Foo` and `FooPrx` (among others). If the interface has the name `while`, the generated identifiers are `_while` and `whilePrx` (*not* `_whilePrx`), that is, the underscore prefix is applied only to those generated identifiers that actually require it.

28.4 Mapping for Modules

By default, identifiers defined within a Slice module are mapped to a flattened symbol that uses underscores as module separators. Consider the following Slice definition:

1. As suggested in Section 4.5.3 on page 92, you should try to avoid such identifiers as much as possible.

```
module M {
    module N {
        enum Color { red, green, blue };
    };
};
```

The Slice identifier `Color` maps to `M_N_Color` by default because PHP releases prior to version 5.3 lacked language support for namespaces. If you prefer to use namespaces instead, you can enable an alternate mapping in which Slice modules map to PHP namespaces with the same name as the Slice module. This mapping preserves the nesting of the Slice definitions. Using the namespace mapping, the Slice identifier `Color` maps to `\M\N\Color`.

Be aware that using underscores in your Slice definitions can lead to name collisions in the flattened mapping. Consider the following example:

```
module M {
    module N {
        enum Color { red, green, blue };
    };
};

module M_N {
    interface Color { };
};
```

Although these definitions are syntactically correct, they both map to the flattened PHP symbol `M_N_Color`.

28.5 The Ice Module

All of the APIs for the Ice run time are nested in the `Ice` module, to avoid clashes with definitions for other libraries or applications. Some of the contents of the `Ice` module are generated from Slice definitions; other parts of the `Ice` module provide special-purpose definitions that do not have a corresponding Slice definition. We will incrementally cover the contents of the `Ice` module throughout the remainder of the book.

A PHP application can load the Ice run time using the `require` statement:

```
require 'Ice.php';
```

If the statement executes without error, the Ice run time is loaded and available for use. You can determine the version of the Ice run time you have just loaded by calling the `stringVersion` function:

```
$icever = Ice_stringVersion();
```

Using the namespace mapping, you can refer to a global Ice function such as `stringVersion` either by its flattened name (as shown above) or by its namespace equivalent:

```
$icever = \Ice\stringVersion();
```

28.6 Mapping for Simple Built-In Types

PHP has a limited set of primitive types: `boolean`, `integer`, `double`, and `string`. The Slice built-in types are mapped to PHP types as shown in Table 28.1.

Table 28.1. Mapping of Slice built-in types to PHP.

Slice	Ruby
`bool`	`true` or `false`
`byte`	`integer`
`short`	`integer`
`int`	`integer`
`long`	`integer`
`float`	`double`
`double`	`double`
`string`	`string`

PHP's `integer` type may not accommodate the range of values supported by Slice's `long` type, therefore `long` values that are outside this range are mapped as

strings. Scripts must be prepared to receive an integer or string from any operation that returns a `long` value.

28.6.1 String Mapping

String values returned as the result of a Slice operation (including return values, out parameters, and data members) contain UTF-8 encoded strings unless the program has installed a string converter, in which case string values use the converter's native encoding instead. See Section 32.24 for more information on string converters.

As string input values for a remote Slice operation, Ice accepts `null` in addition to `string` objects; each occurrence of `null` is marshaled as an empty string. Ice assumes that all `string` objects contain valid UTF-8 encoded strings unless the program has installed a string converter, in which case Ice assumes that `string` objects use the native encoding expected by the converter.

28.7 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

28.7.1 Mapping for Enumerations

PHP does not have an enumerated type, so Slice enumerations are mapped to constants in a PHP class: the name of the Slice enumeration becomes the name of the PHP class; for each enumerator, the class contains a constant with the same name as the enumerator (see Section 28.3 for more information on identifiers). For example:

```
enum Fruit { Apple, Pear, Orange };
```

The generated PHP class looks as follows:

```
class Fruit
{
    const Apple = 0;
    const Pear = 1;
    const Orange = 2;
}
```

Since enumerated values are mapped to integers, application code is not required to use the generated constants. When an enumerated value enters the Ice run time, Ice validates that the given integer is within the expected range for the enumeration. However, to minimize the potential for defects in your code, we recommend using the generated constants instead of literal integers.

28.7.2 Mapping for Structures

A Slice structure maps to a PHP class containing a public variable for each member of the structure. The class also provides a constructor whose arguments correspond to the data members. This allows you to instantiate and initialize the class in a single statement (instead of having to first instantiate the class and then assign to its members). Each argument provides a default value appropriate for the member's type. You can also declare different default values for members of primitive and enumerated types, as discussed in Section 4.9.2. For example, here is our `Employee` structure from Section 4.9.4 once more:

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
```

The PHP mapping generates the following definition for this structure:

```
class Employee
{
    public function __construct($number=0, $firstName='',
                                $lastName='');

    public function __toString();

    public $number;
    public $firstName;
    public $lastName;
}
```

The mapping includes a definition for the `__toString` magic method, which returns a string representation of the structure.

28.7.3 Mapping for Sequences

Slice sequences are mapped to native PHP indexed arrays. The first element of the Slice sequence is contained at index 0 (zero) of the PHP array, followed by the remaining elements in ascending index order.

Consider this example:

```
sequence<Fruit> FruitPlatter;
```

You can create an instance of `FruitPlatter` as shown below:

```
// Make a small platter with one Apple and one Orange
//
$platter = array(Fruit::Apple, Fruit::Orange);
```

The Ice run time validates the elements of an array to ensure that they are compatible with the declared type and raises `InvalidArgumentException` if an incompatible type is encountered.

28.7.4 Mapping for Dictionaries

Slice dictionaries map to native PHP associative arrays. The PHP mapping does not currently support all Slice dictionary types, however, because native PHP associative arrays support only integer and string key types. A Slice dictionary whose key type is an enumeration or one of the primitive types `boolean`, `byte`, `short`, `int`, or `long` is mapped as an associative array with an integer key.[2] A Slice dictionary with a string key type is mapped as associative array with a string key. All other key types cause a warning to be generated.

Here is the definition of our `EmployeeMap` from Section 4.9.4:

```
dictionary<long, Employee> EmployeeMap;
```

You can create an instance of this dictionary as shown below:

```
$e1 = new Employee;
$e1->number = 42;
$e1->firstName = "Stan";
$e1->lastName = "Lipmann";

$e2 = new Employee;
```

2. Boolean values are treated as integers, with false equivalent to 0 (zero) and true equivalent to 1 (one).

```
$e2->number = 77;
$e2->firstName = "Herb";
$e2->lastName = "Sutter";

$em = array($e1->number => $e1, $e2->number => $e2);
```

The Ice run time validates the elements of a dictionary to ensure that they are compatible with the declared type; `InvalidArgumentException` exception is raised if an incompatible type is encountered.

28.8 Mapping for Constants

Slice constants are mapped to PHP constants. Consider the following definitions:

```
module M {
    const bool      AppendByDefault = true;
    const byte      LowerNibble = 0x0f;
    const string    Advice = "Don't Panic!";
    const short     TheAnswer = 42;
    const double    PI = 3.1416;

    enum Fruit { Apple, Pear, Orange };
    const Fruit     FavoriteFruit = Pear;
};
```

The mapping for these constants is shown below:

```
define('M_AppendByDefault', true);
define('M_LowerNibble', 15);
define('M_Advice', "Don't Panic!");
define('M_TheAnswer', 42);
define('M_PI', 3.1416);
define('M_FavoriteFruit', M_Fruit::Pear);
```

An application refers to a constant using its flattened name:

```
$ans = M_TheAnswer;
```

Using the namespace mapping, Slice constants are mapped to PHP constants in the enclosing namespace:

```
$ans = \M\TheAnswer;
```

28.9 Mapping for Exceptions

The mapping for exceptions is based on the inheritance hierarchy shown in Figure 28.1

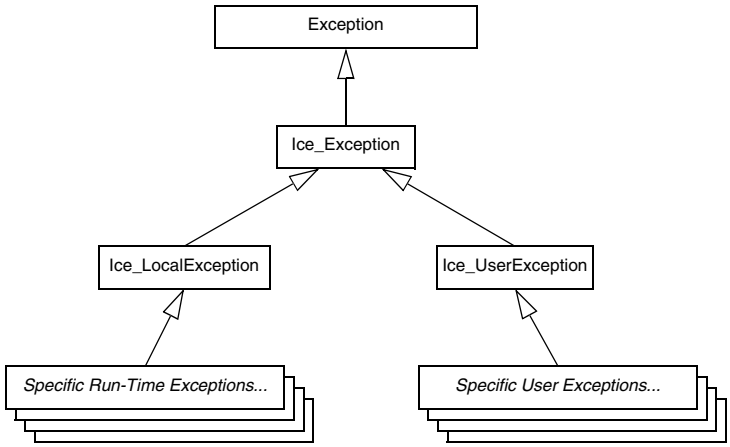


Figure 28.1. Inheritance structure for Ice exceptions.

The ancestor of all exceptions is `Exception`, from which `Ice_Exception` is derived. `Ice_LocalException` and `Ice_UserException` are derived from `Ice_Exception` and form the base for all run-time and user exceptions.

Consider the following Slice definitions:

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

These exception definitions map to the abbreviated PHP class definitions shown below:

```
class GenericError extends Ice_UserException
{
    public function __construct($reason='');
    public function ice_name();
    public function __toString();

    public $reason;
```

```
}

class BadTimeVal extends GenericError
{
    public function __construct($reason='');
    public function ice_name();
    public function __toString();
}

class BadZoneName extends GenericError
{
    public function __construct($reason='');
    public function ice_name();
    public function __toString();
}
```

Each Slice exception is mapped to a PHP class with the same name. The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception member corresponds to an instance variable of the instance, which the constructor initializes to a default value appropriate for its type. You can also declare different default values for members of primitive and enumerated types, as discussed in Section 4.10.2.

Although `BadTimeVal` and `BadZoneName` do not declare data members, their constructors still accept a value for the inherited data member `reason` in order to pass it to the constructor of the base exception `GenericError`.

Each exception also defines the `ice_name` method to return the exception's type name, as well as the `__toString` magic method to return a stringified representation of the exception and its members.

All user exceptions are derived from the base class `Ice_UserException`. This allows you to catch all user exceptions generically by installing a handler for `Ice_UserException`. Similarly, you can catch all Ice run-time exceptions with a handler for `Ice_LocalException`, and you can catch all Ice exceptions with a handler for `Ice_Exception`.

28.10 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `Ice_LocalException` (which, in turn, derives from `Ice_Exception`).

An inheritance diagram for user and run-time exceptions appears in Figure 4.4 on page 117. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `Ice_LocalException`

  This is the root of the inheritance tree for run-time exceptions.
- `Ice_UserException`

  This is the root of the inheritance tree for user exceptions.
- `Ice_TimeoutException`

  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `Ice_ConnectTimeoutException`

  This exception is raised when the initial attempt to establish a connection to a server times out.

You will probably have little need to catch the remaining run-time exceptions; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. However, there is one exception you will probably be interested in specifically: `Ice_ObjectNotExistException`. This exception is raised if a client invokes an operation on an Ice object that no longer exists. In other words, the client holds a dangling reference to an object that probably existed some time in the past but has since been permanently destroyed.

28.11 Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a method on a local class instance that represents the remote object. This makes the mapping easy and intuitive to use because, for all intents and purposes (apart from error semantics), making a remote procedure call is no different from making a local procedure call.

28.11.1 Proxy Objects

Slice interfaces are implemented by instances of the `Ice_ObjectPrx` class. In the client's address space, an instance of `ObjectPrx` is the local ambassador for a remote instance of an interface in a server and is known as a *proxy instance*. All

the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

The PHP mapping for proxies differs from that of other Ice language mappings in that the `ObjectPrx` class is used to implement *all* Slice interfaces. The primary motivation for this design is minimizing the amount of code that is generated for each interface. As a result, a proxy object returned by the communicator operations `stringToProxy` and `propertyToProxy` is *untyped*, meaning it is not associated with a user-defined Slice interface. Once you narrow the proxy to a particular interface, you can use that proxy to invoke your Slice operations.

Proxy instances are always created on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.

A value of `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

For each operation in the interface, the proxy object supports a method of the same name. Each operation accepts an optional trailing parameter representing the operation context. This parameter is an associative string array for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the context parameter in detail in Chapter 32. The parameter is also used by IceStorm—see Chapter 44.)

28.11.2 The `Ice_ObjectPrx` Class

In the PHP language mapping, all proxies are instances of `Ice_ObjectPrx`. This class provides a number of methods:

```
class Ice_ObjectPrx
{
    function ice_getIdentity();
    function ice_isA($id);
    function ice_id();
    function ice_ping();
    # ...
}
```

The methods behave as follows:

- `ice_getIdentity`

  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
};
```

To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

```
$proxy1 = ...
$proxy2 = ...
$id1 = $proxy1->ice_getIdentity();
$id2 = $proxy2->ice_getIdentity();

if($id1 == $id2)
    // proxy1 and proxy2 denote the same object
else
    // proxy1 and proxy2 denote different objects
```

- `ice_isA`

  This method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID (see Section 4.13). For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

```
$proxy = ...
if($proxy != null && $proxy->ice_isA("::Printer"))
    // proxy denotes a Printer object
else
    // proxy denotes some other type of object
```

  Note that we are testing whether the proxy is `null` before attempting to invoke the `ice_isA` method. This avoids getting a run-time error if the proxy is `null`.

- `ice_id`

  This method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the narrowed type of the proxy. For example, if we have a proxy that has been narrowed to the type ID `::Base`, the return value of `ice_id` might be `"::Base"`, or it might be something more derived, such as `"::Derived"`.

- `ice_ping`

  This method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as `ObjectNotExistException` or `ConnectTimeoutException`.

The `ObjectPrx` class also defines an operator for comparing two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `==` returns `false` even though the proxies denote the same object.

There are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call. (We discuss these methods in Chapter 32.)

28.11.3 Casting Proxies

The PHP mapping for a proxy generates a class with two static methods. For example, the following class is generated for the Slice interface named `Simple`:

```
class SimplePrxHelper
{
    public static function
    checkedCast($proxy, $facetOrCtx=null, $ctx=null);

    public static function
    uncheckedCast($proxy, $facet=null);
}
```

Both the `checkedCast` and `uncheckedCast` methods implement a downcast: if the passed proxy is a proxy for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a proxy narrowed to that type; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is `null`), the cast returns `null`.

The method names `checkedCast` and `uncheckedCast` are reserved for use in proxies. If a Slice interface defines an operation with either of those names, the mapping escapes the name in the generated proxy by prepending an underscore. For example, an interface that defines an operation named `checkedCast` is mapped to a proxy with a method named `_checkedCast`.

The arguments are described below:

- `$proxy`

  The proxy to be narrowed.

- `$facetOrCtx`

  This optional argument can be either a string representing a desired facet (see Chapter 33), or an associative string array representing a context (see Section 32.12).

- `$ctx`

  If `$facetOrCtx` contains a facet name, use this argument to supply an associative string array representing a context (see Section 32.12).

- `$facet`

  Specifies the name of the desired facet (see Chapter 33).

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

```
$obj = ...      // Get a proxy from somewhere...

$simple = SimplePrx::checkedCast($obj);
if($simple != null)
    // Object supports the Simple interface...
else
    // Object is not of type Simple...
```

Note that a `checkedCast` contacts the server. This is necessary because only the server implementation has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`.

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather non-sensical things. To illustrate this, consider the following two interfaces:

```
interface Process {
    void launch(int stackSize, int dataSize);
};

// ...

interface Rocket {
    void launch(float xCoord, float yCoord);
};
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

```
$obj = ...                                    // Get proxy...
$process = ProcessPrx::uncheckedCast($obj); // No worries...
$process->launch(40, 60);                     // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

Backward Compatibility

Prior releases of the PHP language mapping provided two proxy methods for narrowing a proxy:

```
class Ice_ObjectPrx
{
    function ice_checkedCast($type, $facetOrCtx=null, $ctx=null);
    function ice_uncheckedCast($type, $facet=null);
    # ...
}
```

For example, a proxy can be narrowed as follows:

```
$proxy = $proxy->ice_checkedCast("::Demo::Hello");
```

The disadvantage of this API is that embedded type ID strings in an application are a potential source of defects because the strings are not validated until run time. Although these methods are still supported for the sake of backward compatibility, we recommend using the static methods that are generated for each interface. Not only are these static methods consistent with the APIs of other Ice language mappings, they also avoid the need to hard-code type ID strings in your application.

28.11.4 Using Proxy Methods

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy (see Section 32.11). Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second timeout as shown below:

```
$proxy = $communicator->stringToProxy(...);
$proxy = $proxy->ice_timeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a down-cast after using a factory method. The example below demonstrates these semantics:

```
$base = $communicator->stringToProxy(...);
$hello = Demo_HelloPrxHelper::checkedCast($base);
$hello = $hello->ice_timeout(10000); // Type is not discarded
$hello->sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return an untyped proxy that you must subsequently down-cast to an appropriate type.

28.11.5 Object Identity and Proxy Comparison

Proxy objects support comparison using the comparison operators `==` and `!=`. Note that proxy comparison uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint

information, must be the same. In other words, comparison tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

```
$p1 = ...        // Get a proxy...
$p2 = ...        // Get another proxy...

if($p1 != $p2)
    // p1 and p2 denote different objects       // WRONG!
else
    // p1 and p2 denote the same object         // Correct
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each uses a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function in the `Ice` module:

```
function Ice_proxyIdentityCompare($lhs, $rhs);
function Ice_proxyIdentityAndFacetCompare($lhs, $rhs);
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

```
$p1 = ...        // Get a proxy...
$p2 = ...        // Get another proxy...

if(Ice_proxyIdentityCompare($p1, $p2) != 0)
    // p1 and p2 denote different objects       // Correct
else
    // p1 and p2 denote the same object         // Correct
```

The function returns 0 if the identities are equal, –1 if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major sort key and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name (see Chapter 33).

28.12 Mapping for Operations

As we learned in Section 28.11, for each operation on an interface, a proxy object narrowed to that type supports a corresponding method with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system from Section 5.4:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

```
$node = ...              // Initialize proxy
$name = $node->name();   // Get name via RPC
```

28.12.1 Normal and `idempotent` Operations

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect.

28.12.2 Passing Parameters

In Parameters

The PHP mapping for in parameters guarantees that the value of a parameter will not be changed by the invocation.

Here is an interface with operations that pass parameters of various types from client to server:

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;
```

```
interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

A proxy object narrowed to the `ClientToServer` interface supports the following methods:

```
function op1($i, $f, $b, $s, $_ctx=null);
function op2($ns, $ss, $st, $_ctx=null);
function op3($proxy, $_ctx=null);
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

```
$p = ...                                  // Get proxy...

$p->op1(42, 3.14, true, "Hello world!"); // Pass simple literals

$i = 42;
$f = 3.14;
$b = true;
$s = "Hello world!";
$p->op1($i, $f, $b, $s);                  // Pass simple variables

$ns = new NumberAndString;
$ns->x = 42;
$ns->str = "The Answer";
$ss = array("Hello world!");
$st = array();
$st[0] = $ns;
$p->op2($ns, $ss, $st);                   // Pass complex variables

$p->op3($p);                              // Pass proxy
```

Out Parameters

Out parameters are passed by reference. Here is the same Slice definition we saw on page 825 once more, but this time with all parameters being passed in the `out` direction:

```
struct NumberAndString {
    int x;
    string str;
};
```

```
sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    int op1(out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The PHP mapping looks the same as it did for the in-parameters version:

```
function op1($i, $f, $b, $s, $_ctx=null);
function op2($ns, $ss, $st, $_ctx=null);
function op3($proxy, $_ctx=null);
```

Given a proxy to a `ServerToClient` interface, the client code can receive the results as in the following example:

```
$p = ...                  // Get proxy...
$p->op1($i, $f, $b, $s);
$p->op2($ns, $ss, $st);
$p->op3($stcp);
```

Note that it is not necessary to use the reference operator (`&`) before each argument because the Ice run time forces each out parameter to have reference semantics.

Parameter Type Mismatches

The Ice run time performs validation on the arguments to a proxy invocation and reports any type mismatches as `InvalidArgumentException`.

Null Parameters

Some Slice types naturally have “empty” or “not there” semantics. Specifically, sequences, dictionaries, and strings all can be `null`, but the corresponding Slice types do not have the of a null value. To make life with these types easier, whenever you pass `null` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly

initialize, for example, every string element in a large sequence before sending the sequence in order to avoid a run-time error. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, it makes no difference to the receiver whether you send a string as `null` or as an empty string: either way, the receiver sees an empty string.

28.13 Exception Handling

Any operation invocation may throw a run-time exception (see Section 28.10) and, if the operation has an exception specification, may also throw user exceptions (see Section 28.9). Suppose we have the following simple interface:

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as PHP exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

```
$child = ...        // Get child proxy...

try
{
    $child->askToCleanUp();
}
catch(Tantrum $t)
{
    echo "The child says: " . $t->reason . "\n";
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will usually be handled by exception handlers higher in the hierarchy. For example:

```
function run()
{
    $child = ...          // Get child proxy...
    try
    {
        $child->askToCleanUp();
    }
    catch(Tantrum $t)
    {
        echo "The child says: " . $t->reason . "\n";
        $child->scold();  // Recover from error...
    }
    $child->praise();     // Give positive feedback...
}

try
{
    // ...
    run();
    // ...
}
catch(Ice_Exception $ex)
{
    echo $ex->__toString() . "\n";
}
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application in Chapter 3.)

28.14 Mapping for Classes

Slice classes are mapped to PHP classes with the same name. For each Slice data member, the generated class contains a member variable, just as for structures and exceptions. Consider the following class definition:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The PHP mapping generates the following code for this definition:

```
abstract class TimeOfDay extends Ice_ObjectImpl
{
    public function __construct($hour=0, $minute=0, $second=0)
    {
        $this->hour = $hour;
        $this->minute = $minute;
        $this->second = $second;
    }

    abstract public function format();

    public static function ice_staticId()
    {
        return '::TimeOfDay';
    }

    public function __toString()
    {
        // ...
    }

    public $hour;
    public $minute;
    public $second;
}
```

There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` inherits from `Ice_ObjectImpl`. This reflects the semantics of Slice classes in that all classes implicitly inherit from `Object`, which is the ultimate ancestor of all classes. Note that `Object` is *not* the same as `Ice_ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.

2. The constructor initializes an instance variable for each Slice data member.

3. The class includes an abstract function declaration corresponding to the Slice operation `format`.

4. The class defines the class method `ice_staticId`.

We will discuss these items in the subsections below.

28.14.1 Inheritance from `Object`

Like interfaces, classes implicitly inherit from a common base class, `Ice_Object`. However, classes inherit from `Ice_Object` instead of `Ice_ObjectPrx`, therefore you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`Ice_Object` contains a number of member functions:

```
interface Ice_Object
{
    public function ice_isA($id);

    public function ice_ping();

    public function ice_ids();

    public function ice_id();

    public function ice_preMarshal();

    public function ice_postUnmarshal();
}
```

The member functions of `Ice_Object` behave as follows:

- `ice_isA`

  This function returns `true` if the object supports the given type ID, and `false` otherwise.

- `ice_ping`

  As for interfaces, `ice_ping` provides a basic reachability test for the class.

- `ice_ids`

  This function returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.

- `ice_id`

  This function returns the actual run-time type ID for a class. If you call `ice_id` through a reference to a base instance, the returned type ID is the actual (possibly more derived) type ID of the instance.

- `ice_preMarshal`

  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.

- `ice_postUnmarshal`

  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.

All Slice classes derive from `Ice_Object` via the `Ice_ObjectImpl` abstract base class, which provides default implementations of the `Ice_Object` methods.

28.14.2 Data Members of Classes

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding member variable.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

```
class TimeOfDay {
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler produces the following generated code for this definition:

```
abstract class TimeOfDay extends Ice_ObjectImpl
{
    public function __construct($hour=0, $minute=0, $second=0)
    {
        $this->hour = $hour;
        $this->minute = $minute;
        $this->second = $second;
    }

    abstract public function format();

    public static function ice_staticId()
    {
        return '::TimeOfDay';
```

```
    }

    public function __toString()
    {
        // ...
    }

    protected $hour;
    protected $minute;
    protected $second;
}
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

```
["protected"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

28.14.3 Class Constructors

Classes have a constructor that assigns to each data member a default value appropriate for its type. You can also declare different default values for data members of primitive and enumerated types, as discussed in Section 4.11.1.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

28.14.4 Operations of Classes

Operations of classes are mapped to abstract member functions in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

```
class TimeOfDayI extends TimeOfDay
{
    public function format()
    {
        return strftime("%X");
    }
}
```

Class Factories

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

```
class ObjectFactory implements Ice_ObjectFactory {
    public function create($type) {
        if ($type == TimeOfDay::ice_staticId())) {
            return new TimeOfDayI;
        }
        assert(false);
        return null;
    }

    public function destroy() {
        // Nothing to do
    }
}
```

The object factory's `create` method is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` method is called by the Ice run time when its communicator is destroyed.

The `create` method is passed the type ID (see Section 4.13) of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::TimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `TimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoObjectFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

```
$communicator = ...;
$communicator->addObjectFactory(
    new ObjectFactory, TimeOfDay::ice_staticId());
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory

while it is registered with the communicator—if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

28.14.5 Class Constructors

The generated class contains a constructor that accepts one argument for each member of the class, with suitable default arguments if you do not wish to supply a value. This allows you to create and initialize a class in a single statement, for example:

```
$tod = new TimeOfDayI(14, 45, 00); // 14:45pm
```

For derived classes, the constructor requires one argument of all of the members of the class, including members of the base class(es). For example, consider the the definition from Section 4.11.2 once more:

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

class DateTime extends TimeOfDay {
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
};
```

The constructors for the generated classes are as follows:

```
class TimeOfDay extends Ice_ObjectImpl
{
    public function __construct($hour=0, $minute=0, $second=0)
    {
        $this->hour = $hour;
        $this->minute = $minute;
        $this->second = $second;
    }

    // ...
}

class DateTime extends TimeOfDay
{
    public function __construct($hour=0, $minute=0, $second=0,
                                $day=0, $month=0, $year=0)
    {
        parent::__construct($hour, $minute, $second);
        $this->day = $day;
        $this->month = $month;
        $this->year = $year;
    }

    // ...
}
```

28.15 slice2php Command-Line Options

The Slice-to-PHP compiler, **slice2php**, offers the following command-line options in addition to the standard options described in Section 4.20:

- **--all**

  Generate code for all Slice definitions, including those included by the main Slice file.

- **-n, --namespace**

  Generate code using PHP namespaces. Note that namespaces are only supported in PHP 5.3 or later. Also note that the Ice extension for PHP must be built with namespace support enabled.

- **--checksum**

  Generate checksums for Slice definitions.

Compiler Output

For each Slice file `X.ice`, **`slice2php`** generates PHP code into a file named `X.php` in the output directory. The default output directory is the current working directory, but a different directory can be specified using the **`--output-dir`** option.

Include Files

It is important to understand how **`slice2php`** handles include files. In the absence of the **`--all`** option, the compiler does not generate PHP code for Slice definitions in included files. Rather, the compiler translates Slice `#include` statements into PHP `require` statements in the following manner:

1. Determine the full pathname of the included file.
2. Create the shortest possible relative pathname for the included file by iterating over each of the include directories (specified using the **`-I`** option) and removing the leading directory from the included file if possible.

   For example, if the full pathname of an included file is `/opt/App/slice/OS/Process.ice`, and we specified the options **`-I/opt/App`** and **`-I/opt/App/slice`**, then the shortest relative pathname is `OS/Process.ice` after removing `/opt/App/slice`.
3. Replace the `.ice` extension with `.php`. Continuing our example from the previous step, the translated `require` statement becomes

   ```
   require "OS/Process.php";
   ```

As a result, you can use **`-I`** options to tailor the `require` statements generated by the compiler in order to avoid absolute path names and match the organizational structure of your application's source files.

28.16 Application Notes

In PHP terminology, a *request* is the execution of a PHP script on behalf of a Web client. Each request essentially runs in its own instance of the PHP interpreter, isolated from any other requests that may be executing concurrently. Upon the completion of a request, the interpreter reclaims memory and other resources that were acquired during the request, including objects created by the Ice extension.

28.16.1 Using Communicators

A communicator represents an instance of the Ice run time. A PHP script that needs to invoke an operation on a remote Ice object must initialize a communicator, obtain and narrow a proxy, and make the invocation. For example, here is a minimal (but complete) Ice script:

```
<?php
require 'Ice.php';
require 'Hello.php';

$communicator = null;

try
{
    $data = new Ice_InitializationData;
    $data->properties = Ice_createProperties();
    $data->properties->load("props.cfg");
    $communicator = Ice_initialize($data);
    $proxy = $communicator->stringToProxy("...");
    $hello = Demo_HelloPrxHelper::checkedCast($proxy);
    $hello->sayHello();
}
catch(Ice_LocalException $ex)
{
    // Deal with exception...
}

if($communicator)
{
    try
    {
        $communicator->destroy();
    }
    catch(Ice_LocalException $ex)
    {
        // Ignore.
    }
}
?>
```

By default, the Ice extension automatically destroys any communicator that was created during a request. This means a script can usually omit the call to `destroy` unless there is an application-specific reason to destroy the communicator explicitly. Consequently, we can simplify our script to the following:

```
<?php
require 'Ice.php';
require 'Hello.php';

try
{
    $data = new Ice_InitializationData;
    $data->properties = Ice_createProperties();
    $data->properties->load("props.cfg");
    $communicator = Ice_initialize($data);
    $proxy = $communicator->stringToProxy("...");
    $hello = Demo_HelloPrxHelper::checkedCast($proxy);
    $hello->sayHello();
}
catch(Ice_LocalException $ex)
{
    // Deal with exception...
}
?>
```

Now we allow the Ice extension to destroy our communicator automatically. (The extension traps and ignores any exception raised by `destroy`.)

Although the automatic destruction of communicators is convenient, it is important to consider the performance characteristics of this script. Specifically, each execution of the script involves the following activities:

1. Create a property set
2. Load and parse a property file
3. Initialize a communicator with the given configuration properties
4. Obtain a proxy for the remote Ice object
5. Establish a socket connection to the server
6. Send a request message and wait for the reply
7. Destroy the communicator, which closes the socket connection

Of primary concern are the activities that involve system calls, such as opening and reading files, creating and using sockets, and so on. The overhead incurred by these calls may not matter if the script is only executed infrequently, but for an application with high request rates it is necessary to minimize this overhead:

- Using pre-configured property sets eliminates the need to parse a property file in each request (see Section 28.16.2).
- Using timeouts prevents a script from blocking indefinitely in case Ice encounters delays while performing socket operations (see Section 28.16.3).

- Using a registered communicator avoids the need to create and destroy a communicator in every request (see Section 28.16.4).
- Be aware of the number of "round trips" (request-reply pairs) your script makes. For example, the script above uses `checkedCast` to verify that the remote Ice object supports the desired Slice interface. However, calling `checkedCast` causes the Ice run time to send a request to the server and await its reply, therefore this script is actually making two remote invocations. It is unnecessary to perform a checked cast if it is safe for the client to assume that the Ice object supports the correct interface, in which case using an `uncheckedCast` instead avoids the extra round trip.

28.16.2 Configuration

A PHP application can manually construct a property set for configuring its communicator using the API described in Chapter 30. The Ice extension also provides a PHP-specific property set API that helps to minimize the overhead associated with initializing a communicator, allowing you to configure a default property set along with an unlimited number of named property sets (or *profiles*). You can populate a property set using a configuration file, command-line options, or both. Property sets are initialized using the normal Ice semantics: command-line options override any settings from a configuration file.

The Ice extension creates these property sets during web server startup, which means any subsequent changes you might make to the configuration have no effect until the web server is restarted. Also keep in mind that specifying a relative path name for a configuration file usually means the path name is evaluated relative to the web server's working directory.

Default Property Set

The INI directives `ice.config` and `ice.options` specify the configuration file and the command-line options for the default property set, respectively. These directives must appear in PHP's configuration file, which is usually named `php.ini`:

```
; Snippet from php.ini on Linux
extension=IcePHP.so
ice.config=/opt/MyApp/default.cfg
ice.options="--Ice.Override.Timeout=2000"
```

Profiles

Profiles are useful when several unrelated applications execute in the same web server, or when an application needs to choose among multiple configurations. To configure your profiles, add an `ice.profiles` directive to PHP's configuration file. The value of this directive is a file containing profile definitions:

```
; Snippet from php.ini on Linux
ice.profiles=/opt/MyApp/profiles
```

The profile definition file uses INI syntax:

```
[Production]
config=/opt/MyApp/prod.cfg
options="..."

[Debug]
config=/opt/MyApp/debug.cfg
options="--Ice.Trace.Network=3 ..."
```

The name of each profile is enclosed in square brackets. The configuration file and command-line options for each profile are defined using the `config` and `options` entries, respectively.

Using Property Sets

The `Ice_getProperties` function allows a script to obtain a copy of a property set. When called without an argument, or with an empty string, the function returns the default property set. Otherwise, the function expects the name of a configured profile and returns the property set associated with that profile. The return value is an instance of `Ice_Properties`, or `null` if no matching profile was found.

Note that the Ice extension always creates the default property set, which is empty if the `ice.config` and `ice.options` directives are not defined. Also note that changes a script might make to a property set returned by this function have no effect on other requests because the script is modifying a copy of the original property set.

Now we can modify our script to use `Ice_getProperties` and avoid the need to load a configuration file in each request:

```
<?php
require 'Ice.php';
require 'Hello.php';

try
```

```
{
    $data = new Ice_InitializationData;
    $data->properties = Ice_getProperties();
    $communicator = Ice_initialize($data);
    $proxy = $communicator->stringToProxy("...");
    $hello = Demo_HelloPrxHelper::checkedCast($proxy);
    $hello->sayHello();
}
catch(Ice_LocalException $ex)
{
    // Deal with exception...
}
?>
```

Security Considerations

Ice configuration properties may contain sensitive information such as the path name of the private key for an X.509 certificate. If multiple untrusted PHP applications run in the same web server, avoid the use of the default property set and choose sufficiently unique names for your named profiles. The Ice extension does not provide a means for enumerating the names of the configured profiles, therefore a malicious script would have to guess the name of a profile in order to examine its configuration properties.

To prevent a script from using the value of `ice.profiles` to open the profile definition file directly, enable the `ice.hide_profiles` directive to cause the Ice extension to replace the `ice.profiles` setting after it has processed the file. The `ice.hide_profiles` directive is enabled by default.

28.16.3 Timeouts

All twoway remote invocations made by a PHP script have synchronous semantics: the script does not regain control until Ice receives a reply from the server. As a result, we recommend configuring a suitable timeout value for all of your proxies as a defensive measure against network delays. Refer to Section 32.13 for more information on timeouts.

28.16.4 Registered Communicators

You can register a communicator to prevent it from being destroyed at the completion of a script. For example, a session-based PHP application can create a communicator for each new session and register it for reuse in subsequent requests

of the same session. Reusing a communicator in this way avoids the overhead associated with creating and destroying a communicator in each request. Furthermore, it allows socket connections established by the Ice run time to remain open and available for use in another request.

Limitations

A communicator object is local to the process that created it, which in the case of PHP is usually a web server process. The usefulness of a registered communicator is therefore limited to situations in which an application can ensure that subsequent page requests are handled by the same web server process as the one that originally created the registered communicator. For example, registered communicators would not be appropriate in a typical CGI configuration because the CGI process terminates at the end of each request. A simple (but often impractical) solution is to configure your web server to use a single persistent process. The topic of configuring a web server to take advantage of registered communicators is outside the scope of this book.

Using Registered Communicators

The API for registered communicators consists of three functions:

- `Ice_register($communicator, $name, $expires=0)`

  Registers a communicator with the given name. On success, the function returns true. If another communicator is already registered with the same name, the function returns false. The `expires` argument specifies a timeout value in minutes; if `expires` is greater than zero, the Ice extension automatically destroys the communicator if it has not been retrieved (via `Ice_find`) for the specified number of minutes. The default value (zero) means the communicator never expires, in which case the Ice extension only destroys the communicator when the current process terminates.

  It is legal to register a communicator with more than one name. In that case, the most recent value of `expires` takes precedence.

- `Ice_unregister($name)`

  Removes the registration for a communicator with the given name. Returns true if a match was found or false otherwise. Calling `Ice_unregister` does not cause the communicator to be destroyed; rather, the communicator is destroyed as soon as all pending requests that are currently using the communicator have completed. Destroying a registered communicator explicitly also removes its registration.

- `Ice_find($name)`

  Retrieves the communicator associated with the given name. Returns `null` if no match is found.

An application typically uses registered communicators as follows:

```
<?php
require 'Ice.php';

$communicator = Ice_find('MyCommunicator');
$expires = ...;
if($communicator == null)
{
    $communicator = Ice_initialize(...);
    Ice_register($communicator, 'MyCommunicator', $expires);
}

...
?>
```

Note that communicators consume resources such as threads, sockets, and memory, therefore an application should be designed to minimize the number of communicators it registers. Using a suitable expiration timeout prevents registered communicators from accumulating indefinitely.

A simple application that demonstrates the use of registered communicators can be found in the `Glacier2/hello` subdirectory of the PHP sample programs.

Security Considerations

There are risks associated with allowing untrusted applications to gain access to a registered communicator. For example, if a malicious script obtains a registered communicator that is configured with SSL credentials, the script could potentially make secure invocations as if it were the trusted script.

Registering a communicator with a sufficiently unique name reduces the chance that a malicious script could guess the communicator's name. For applications that make use of PHP's session facility, the session ID is a reasonable choice for a communicator name. The sample application in `Glacier2/hello` demonstrates this solution.

Object Factories

PHP reclaims all memory at the end of each request, which means any object factories that a script might have installed in a registered communicator are

destroyed when the request completes even if the communicator is not destroyed. As a result, a script must install its object factories in a registered communicator for *every* request, as shown in the example below:

```
<?php
require 'Ice.php';

$communicator = Ice_find('MyCommunicator');
$expires = ...;
if($communicator == null)
{
    $communicator = Ice_initialize(...);
    Ice_register($communicator, 'MyCommunicator', $expires);
}

$communicator->addObjectFactory(new MyFactory,
                                MyClass::ice_staticId());
...
?>
```

The Ice extension invokes the `destroy` method of each factory prior to the completion of a request.

28.17 Using Slice Checksums

As described in Section 4.21, the Slice compilers can optionally generate checksums of Slice definitions. For **`slice2php`**, the **`--checksum`** option causes the compiler to generate code that adds checksums to the global array `Ice_sliceChecksums`. The checksums are installed automatically when the PHP code is first parsed; no action is required by the application.

In order to verify a server's checksums, a client could simply compare the two array objects using a comparison operator. However, this is not feasible if it is possible that the server might return a superset of the client's checksums. A more general solution is to iterate over the local checksums as demonstrated below:

```
global $Ice_sliceChecksums;
$serverChecksums = ...
foreach($Ice_sliceChecksums as $key => $value)
{
    if(!isset($serverChecksums[$key]))
```

```
        // No match found for type id!
    elseif($Ice_sliceChecksums[$key] != $serverChecksums[$key])
        // Checksum mismatch!
}
```

In this example, the client first verifies that the server's dictionary contains an entry for each Slice type ID, and then it proceeds to compare the checksums.

Chapter 29
Developing a File System Client in PHP

29.1 Chapter Overview

In this chapter, we present the source code for a PHP client that accesses the file system we developed in Chapter 5. This client can interact with a server written in any of the other language mappings.

29.2 The PHP Client

We now have seen enough of the PHP mapping to develop a complete client to access our remote file system. For reference, here is the Slice definition once more:

```
module Filesystem {
    interface Node {
        idempotent string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;
```

```
    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

To exercise the file system, the client does a recursive listing of the file system, starting at the root directory. For each node in the file system, the client shows the name of the node and whether that node is a file or directory. If the node is a file, the client retrieves the contents of the file and prints them.

The body of the client code looks as follows:

```
<?php
require 'Ice.php';
require 'Filesystem.php';

// Recursively print the contents of directory "dir"
// in tree fashion. For files, show the contents of
// each file. The "depth" parameter is the current
// nesting level (for indentation).

function listRecursive($dir, $depth = 0)
{
    $indent = str_repeat("\t", ++$depth);

    $contents = $dir->_list(); // list is a reserved word in PHP

    foreach ($contents as $i) {
        $dir = Filesystem_DirectoryPrxHelper::checkedCast($i);
        $file = Filesystem_FilePrxHelper::uncheckedCast($i);
        echo $indent . $i->name() .
            ($dir ? " (directory):" : " (file):") . "\n";
        if ($dir) {
            listRecursive($dir, $depth);
        } else {
            $text = $file->read();
            foreach ($text as $j)
                echo $indent . "\t" . $j . "\n";
        }
```

```
    }
}

$ic = null;
try
{
    // Create a communicator
    //
    $ic = Ice_initialize();

    // Create a proxy for the root directory
    //
    $obj = $ic->stringToProxy("RootDir:default -p 10000");

    // Down-cast the proxy to a Directory proxy
    //
    $rootDir = Filesystem_DirectoryPrxHelper::checkedCast($obj);

    // Recursively list the contents of the root directory
    //
    echo "Contents of root directory:\n";
    listRecursive($rootDir);
}
catch(Ice_LocalException $ex)
{
    print_r($ex);
}

if($ic)
{
    // Clean up
    //
    try
    {
        $ic->destroy();
    }
    catch(Exception $ex)
    {
        print_r($ex);
    }
}
?>
```

The program first defines the `listRecursive` function, which is a helper function to print the contents of the file system, and the main program follows. Let us look at the main program first:

1. The client first creates a proxy to the root directory of the file system. For this example, we assume that the server runs on the local host and listens using the default protocol (TCP/IP) at port 10000. The object identity of the root directory is known to be `RootDir`.

2. The client down-casts the proxy to the `Directory` interface and passes that proxy to `listRecursive`, which prints the contents of the file system.

Most of the work happens in `listRecursive`. The function is passed a proxy to a directory to list, and an indent level. (The indent level increments with each recursive call and allows the code to print the name of each node at an indent level that corresponds to the depth of the tree at that node.) `listRecursive` calls the `list` operation on the directory and iterates over the returned sequence of nodes:

1. The code uses `checkedCast` to narrow the `Node` proxy to a `Directory` proxy, and uses `uncheckedCast` to narrow the `Node` proxy to a `File` proxy. Exactly one of those casts will succeed, so there is no need to call `checkedCast` twice: if the `Node` *is-a* `Directory`, the code uses the proxy returned by `checkedCast`; if `checkedCast` fails, we *know* that the Node *is-a* File and, therefore, `uncheckedCast` is sufficient to get a `File` proxy.

   In general, if you know that a down-cast to a specific type will succeed, it is preferable to use `uncheckedCast` instead of `checkedCast` because `uncheckedCast` does not incur any network traffic.

2. The code prints the name of the file or directory and then, depending on which cast succeeded, prints `"(directory)"` or `"(file)"` following the name.

3. The code checks the type of the node:

   - If it is a directory, the code recurses, incrementing the indent level.

   - If it is a file, the code calls the `read` operation on the file to retrieve the file contents and then iterates over the returned sequence of lines, printing each line.

Assume that we have a small file system consisting of a two files and a a directory as follows:

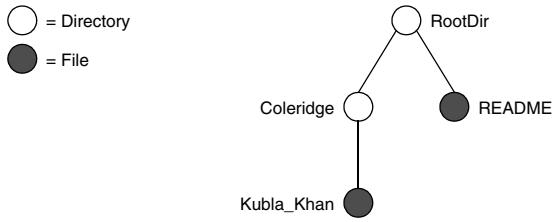


Figure 29.1. A small file system.

The output produced by the client for this file system is:

```
Contents of root directory:
        README (file):
                This file system contains a collection of poetry.
        Coleridge (directory):
                Kubla_Khan (file):
                        In Xanadu did Kubla Khan
                        A stately pleasure-dome decree:
                        Where Alph, the sacred river, ran
                        Through caverns measureless to man
                        Down to a sunless sea.
```

Note that, so far, our client is not very sophisticated:

- The protocol and address information are hard-wired into the code.
- The client makes more remote procedure calls than strictly necessary; with minor redesign of the Slice definitions, many of these calls can be avoided.

We will see how to address these shortcomings in Chapter 38 and Chapter 34.

29.3 Summary

This chapter presented a very simple client to access a server that implements the file system we developed in Chapter 5. As you can see, the PHP code hardly differs from the code you would write for an ordinary PHP program. This is one of the biggest advantages of using Ice: accessing a remote object is as easy as accessing an ordinary, local PHP object. This allows you to put your effort where

you should, namely, into developing your application logic instead of having to struggle with arcane networking APIs.

Part X

Advanced Ice

Chapter 30
Ice Properties and Configuration

30.1 Chapter Overview

Ice uses a configuration mechanism that allows you to control many aspects of the behavior of your Ice applications at run time, such as maximum message size, number of threads, or whether to produce network trace messages. The configuration mechanism is not only useful to configure Ice, but you can also use it to provide configuration parameters to your own applications. The configuration mechanism is simple to use with a minimal API, yet flexible enough to cope with the needs of most applications.

Sections 30.2 to 30.8 describe the basics of the configuration mechanism and explain how to configure Ice via configuration files and command line options. Section 30.9 shows how you can create your own application-specific properties and how to access their values from within a program.

30.2 Properties

Ice and its various subsystems are configured by *properties*. A property is a name–value pair, for example:

```
Ice.UDP.SndSize=65535
```

In this example, the *property name* is `Ice.UDP.SndSize`, and the *property value* is `65535`.

You can find a complete list of the properties used to configure Ice in Appendix D.

Note that Ice reads properties that control the Ice run time and its services (that is, properties that start with one of the reserved prefixes, such as `Ice`, `Glacier2`, etc.) only once on start-up, when you create a communicator. This means that you must set Ice-related properties to their correct values *before* you create a communicator. If you change the value of an Ice-related property after that point, it is likely that the new setting will simply be ignored.

30.2.1 Property Categories

By convention, Ice properties use the following naming scheme:

```
<application>.<category>[.<sub-category>]
```

Note that the sub-category is optional and not used by all Ice properties.

This two- or three-part naming scheme is by convention only—if you use properties to configure your own applications, you can use property names with any number of categories.

30.2.2 Reserved Prefixes

Ice reserves properties with the prefixes `Ice`, `IceBox`, `IceGrid`, `IcePatch2`, `IceSSL`, `IceStorm`, `Freeze`, and `Glacier2`. You cannot use a property beginning with one of these prefixes to configure your own application.

30.2.3 Property Syntax

A property name consists of any number of characters. For example, the following are valid property names:

```
foo
Foo
foo.bar
foo bar      White space is allowed
foo=bar      Special characters are allowed
.
```

Note that there is no special significance to a period in a property name. (Periods are used to make property names more readable and are not treated specially by the property parser.)

Property names cannot contain leading or trailing white space. (If you create a property name with leading or trailing white space, that white space is silently stripped.)

30.2.4 Value Syntax

A property value consists of any number of characters. The following are examples of property values:

```
65535
yes
This is a = property value.
../../config
```

30.3 Configuration Files

Properties are usually set in a configuration file. A configuration file contains a number of name–value pairs, with each pair on a separate line. Empty lines and lines consisting entirely of white space characters are ignored. The # character introduces a comment that extends to the end of the current line.

Here is a simple configuration file:

```
# Example config file for Ice

Ice.MessageSizeMax = 2048    # Largest message size is 2MB
Ice.Trace.Network=3          # Highest level of tracing for network
Ice.Trace.Protocol=          # Disable protocol tracing
```

Leading and trailing white space is always ignored for property *names* (whether the white space is escaped or not), but white space within property *values* is preserved.

For property values, you can indicate leading and trailing whitespace by escaping the white space with a backslash. For example:

```
# White space example

My.Prop = a property                # Value is "a property"
My.Prop =     a     property        # Value is "a     property"
My.Prop = \ \ a     property\ \     # Value is "  a     property  "
My.Prop = \ \ a  \ \  property\ \   # Value is "  a     property  "
My.Prop = a \\ property                # Value is "a \ property"
```

This example shows that leading and trailing white space for property values is ignored unless escaped with a backslash whereas, white space that is surrounded by non-white space characters is preserved exactly, whether it is escaped or not. As usual, you can insert a literal backslash into a property value by using a double backslash (`\\`).

If you set the same property more than once, the last setting prevails and overrides any previous setting. Note that assigning nothing to a property clears that property (that is, sets it to the empty string).

A property that contains the empty string (such as `Ice.Trace.Protocol` in the preceding example) is indistinguishable from a property that is not mentioned at all. This is because the API to retrieve the property value returns the empty string for non-existent properties (see page 871).

One of the simplest ways of loading a configuration file is to pass the command-line option `--Ice.Config` to an Ice program, for example:

```
$ ./server --Ice.Config=/opt/Ice/default_config
```

This causes the server to read its property settings from the configuration file in `/opt/Ice/default_config`. Section 30.5 discusses this option in more detail.

Property values can include characters from non-English alphabets. The Ice run time expects the configuration file to use UTF-8 encoding for such characters. (With C++, you can specify a string converter when you read the file. See page 872, Section 32.3, and Section 32.24.)

30.3.1 The `ICE_CONFIG` Environment Variable

Another way to load a configuration file is to set the `ICE_CONFIG` environment variable to the relative or absolute path name of the file, for example:

```
$ export ICE_CONFIG=/opt/Ice/default_config
$ ./server
```

This causes the server to read its property settings from the configuration file in `/opt/Ice/default_config`.

Note that, for this to work, the server must call an overload of `Ice::initialize` (or the equivalent in other languages) that accepts an argument vector. Overloads of `initialize` that do not accept an argument vector ignore **`ICE_CONFIG`**.

NOTE: In Java, the value of **`ICE_CONFIG`** can refer to class loader resources as well as regular files.

30.3.2 Special Characters

The characters = and # have special meaning in a configuration file:

- = marks the end of the property name and the beginning of the property value
- # starts a comment that extends to the end of the line

These characters must be escaped when they appear in a property name. Consider the following examples:

`foo\=bar=1`	*Name is "`foo=bar`", value is "`1`"*
`foo\#bar = 2`	*Name is "`foo#bar`", value is "`2`"*
`foo bar =3`	*Name is "`foo bar`", value is "`3`"*

In a property value, a # character must be escaped to prevent it from starting a comment, but an = character does not require an escape. Consider these examples:

`A=1`	*Name is "`A`", value is "`1`"*
`B= 2 3 4`	*Name is "`B`", value is "`2 3 4`"*
`C=5=\#6 # 7`	*Name is "`C`", value is "`5=#6`"*

Note that, two successive backslashes in a property value become a single backslash. To get two consecutive backslashes, you must escape each one with another backslash:

`AServer=\\\\server\dir`	*Value is "`\\server\dir`"*
`BServer=\\server\\dir`	*Value is "`\server\dir`"*

The preceding example also illustrates that, if a backslash is not followed by \, #, or =, the backslash and the character following it are both preserved.

30.4 Setting Properties on the Command Line

In addition to setting properties in a configuration file, you can also set properties on the command line, for example:

```
$ ./server --Ice.UDP.SndSize=65535 --IceSSL.Trace.Security=2
```

Any command line option that begins with `--` and is followed by one of the reserved prefixes (see page 858) is read and converted to a property setting when you create a communicator. Property settings on the command line override settings in a configuration file. If you set the same property more than once on the same command line, the last setting overrides any previous ones.

For convenience, any property not explicitly set to a value is set to the value `1`. For example,

```
$ ./server --Ice.Trace.Protocol
```

is equivalent to

```
$ ./server --Ice.Trace.Protocol=1
```

Note that this feature only applies to properties that are set on the command line, but not to properties that are set from a configuration file.

You can also clear a property from the command line as follows:

```
$ ./server --Ice.Trace.Protocol=
```

As for properties set from a configuration file, assigning nothing to a property clears that property.

30.5 Using Configuration Files

The ability to configure an application's properties externally provides a great deal of flexibility: you can use any combination of command-line options and configuration files to achieve the desired settings, all without having to modify your application. This section describes two ways of loading property settings from a file.

30.5.1 Prerequisites

The Ice run time automatically loads a configuration file during the creation of a *property set*, which is an instance of the `Ice::Properties` interface. Every communicator has its own property set from which it derives its configuration. If

an application does not supply a property set when it calls `Ice::initialize` (or the equivalent in other language mappings), the Ice run time internally creates a property set for the new communicator.

Note however that Ice loads a configuration file automatically *only* when the application creates a property set using an argument vector. This occurs when the application passes an argument vector to create a property set explicitly (as discussed in Section 30.9), or when the application passes an argument vector to `Ice::initialize`.

Both of the mechanisms described below can also retrieve property settings from additional sources. Refer to Section 30.6 for more information.

30.5.2 The `ICE_CONFIG` Environment Variable

Ice automatically loads the contents of the configuration file named in the **`ICE_CONFIG`** environment variable (assuming the prerequisites described in Section 30.5.1 are met). For example:

```
$ export ICE_CONFIG=/usr/local/filesystem/config
$ ./server
```

This causes the server to read its property settings from the configuration file in `/usr/local/filesystem/config`.

If you use the **`ICE_CONFIG`** environment variable together with command-line options for other properties, the settings on the command line override the settings in the configuration file. For example:

```
$ export ICE_CONFIG=/usr/local/filesystem/config
$ ./server --Ice.MessageSizeMax=4096
```

This sets the value of the `Ice.MessageSizeMax` property to `4096` regardless of any setting of this property in `/usr/local/filesystem/config`.

You can use multiple configuration files by specifying a list of configuration file names, separated by commas. For example:

```
$ export ICE_CONFIG=/usr/local/filesystem/config,./config
$ ./server
```

This causes property settings to be retrieved from `/usr/local/filesystem/config`, followed by any settings in the file `config` in the current directory; settings in `./config` override settings `/usr/local/filesystem/config`.

30.5.3 The `Ice.Config` Property

The `Ice.Config` property has special meaning to the Ice run time: it determines the path name of a configuration file from which to read property settings. For example:

```
$ ./server --Ice.Config=/usr/local/filesystem/config
```

This causes property settings to be read from the configuration file in `/usr/local/filesystem/config`.

The **`--Ice.Config`** command-line option overrides any setting of the **`ICE_CONFIG`** environment variable, that is, if the **`ICE_CONFIG`** environment variable is set and you also use the **`--Ice.Config`** command-line option, the configuration file specified by the **`ICE_CONFIG`** environment variable is ignored.

If you use the **`--Ice.Config`** command-line option together with settings for other properties, the settings on the command line override the settings in the configuration file. For example:

```
$ ./server --Ice.Config=/usr/local/filesystem/config \
> --Ice.MessageSizeMax=4096
```

This sets the value of the `Ice.MessageSizeMax` property to `4096` regardless of any setting of this property in `/usr/local/filesystem/config`. The placement of the **`--Ice.Config`** option on the command line has no influence on this precedence. For example, the following command is equivalent to the preceding one:

```
$ ./server --Ice.MessageSizeMax=4096 \
> --Ice.Config=/usr/local/filesystem/config
```

Settings of the `Ice.Config` property inside a configuration file are ignored, that is, you can set `Ice.Config` only on the command line.

If you use the **`--Ice.Config`** option more than once, only the last setting of the option is used and the preceding ones are ignored. For example:

```
$ ./server --Ice.Config=file1 --Ice.Config=file2
```

This is equivalent to using:

```
$ ./server --Ice.Config=file2
```

You can use multiple configuration files by specifying a list of configuration file names, separated by commas. For example:

```
$ ./server --Ice.Config=/usr/local/filesystem/config,./config
```

This causes property settings to be retrieved from `/usr/local/file-system/config`, followed by any settings in the file `config` in the current directory; settings in `./config` override settings `/usr/local/file-system/config`.

30.6 Alternate Property Stores

In addition to regular files, Ice also supports storing property settings in the Windows registry and Java resources.

30.6.1 Windows Registry

You can use the Windows registry to store property settings. Property settings must be stored with a key underneath `HKEY_LOCAL_MACHINE`. To inform the Ice run time of this key, you must set the **`Ice.Config`** property to the key. For example:

```
$ client --Ice.Config=HKLM\MyCompany\MyApp
```

The Ice run time examines the value of **`Ice.Config`**; if that value begins with `HKLM\`, the remainder of the property is taken to be a key to a number of string values. For the preceding example, the Ice run time looks for the key `HKEY_LOCAL_MACHINE\MyCompany\MyApp`. The string values stored under this key are used to initialize the properties.

The name of each string value is the name of the property (such as `Ice.Trace.Network`). Note that the value must be a string (even if the property setting is numeric). For example, to set Ice.Trace.Network to 3, you must store the string "3" as the value, not a binary or `DWORD` value.

String values in the registry can be regular strings (`REG_SZ`) or expandable strings (`REG_EXPAND_SZ`). Expandable strings allow you to include symbolic references to environment variables (such as `%ICE_HOME%`).

NOTE: Depending on whether you use 32-bit or 64-bit binaries, you must set the registry keys in the corresponding 32-bit or 64-bit registry. See http://support.microsoft.com/kb/305097 for more information.

30.6.2 Java Resources

The Ice run time for Java supports the ability to load a configuration file as a class loader resource, which is especially useful for deploying an Ice application in a self-contained JAR file. For example, suppose we define **`ICE_CONFIG`** as shown below:

```
$ export ICE_CONFIG=app_config
```

During the creation of a property set (which often occurs implicitly when initializing a new communicator), Ice asks the Java run time to search the application's class path for a file named `app_config`. This file might reside in the same JAR file as the application's class files, or in a different JAR file in the class path, or it might be a regular file located in one of the directories in the class path. If Java is unable to locate the configuration file in the class path, Ice attempts to open the file in the local file system.

The class path resource always takes precedence over a regular file. In other words, if a class path resource and a regular file are both present with the same path name, Ice always loads the class path resource in preference to the regular file.

The path name for a class path resource uses a relative Unix-like format such as `subdir/myfile`. Java searches for the resource relative to each JAR file or subdirectory in an application's class path.

30.7 Command-Line Parsing and Initialization

When you initialize the Ice run time by calling `Ice::initialize` (C++/Ruby), `Ice.Util.initialize` (Java/C#), `Ice.initialize` (Python), or `Ice_initialize` (PHP), you can pass an argument vector to the initialization call.[1]

For C++, `Ice::initialize` accepts a C++ *reference* to `argc`:

```
namespace Ice {
    CommunicatorPtr initialize(int& argc, char* argv[]);
}
```

1. See also Section 32.3.

`Ice::initialize` parses the argument vector and initializes its property settings accordingly. In addition, it removes any arguments from `argv` that are property settings. For example, assume we invoke a server as:

```
$ ./server --myoption --Ice.Config=config -x a \
--Ice.Trace.Network=3 -y opt file
```

Initially, `argc` has the value `9`, and `argv` has ten elements: the first nine elements contain the program name and the arguments, and the final element, `argv[argc]`, contains a null pointer (as required by the ISO C++ standard). When `Ice::initialize` returns, `argc` has the value `7` and `argv` contains the following elements:

```
./server
--myoption
-x
a
-y
opt
file
0                    # Terminating null pointer
```

This means that you should initialize the Ice run time before you parse the command line for your application-specific arguments. That way, the Ice-related options are stripped from the argument vector for you so you do not need to explicitly skip them. If you use the `Ice::Application` helper class (see Section 8.3.1), the `run` member function is passed an argument vector with the Ice-related options already stripped. The same is true for the `runWithSession` member function called by `Glacier2::Application` helper class (see Section 42.5).

For Java, `Ice.Util.initialize` is overloaded. The signatures are:

```
package Ice;
public final class Util {

    public static Communicator
    initialize();

    public static Communicator
    initialize(String[] args);

    public static Communicator
    initialize(StringSeqHolder args);
```

```
    public static Communicator
    initialize(InitializationData id);

    public static Communicator
    initialize(String[] args, InitializationData id);

    public static Communicator
    initialize(StringSeqHolder args, InitializationData id);


    // ...
}
```

The versions that accept an argument vector of type `String[]` do not strip Ice-related options for you, so, if you use one of these methods, your code must ignore options that start with one of the reserved prefixes (`--Ice`, `--IceBox`, `--IceGrid`, `--IcePatch2`, `--IceSSL`, `--IceStorm`, `--Freeze`, and `--Glacier2`). The versions that accept a `StringSeqHolder` behave like the C++ version and strip the Ice-related options from the passed argument vector.

In C#, the argument vector is passed by reference to the `initialize` method, allowing it to strip the Ice-related options:

```
namespace Ice {

    public sealed class Util {

        public static Communicator
        initialize();

        public static Communicator
        initialize(ref string[] args);

        public static Communicator
        initialize(InitializationData id);

        public static Communicator
        initialize(ref string[] args, InitializationData id);

        // ...

    }
}
```

The Python, Ruby, and PHP implementations of `initialize` have the same semantics as C++ and .NET; they expect the argument vector to be passed as a list from which all Ice-related options are removed.

If you use the `Ice.Application` helper class, the `run` method is passed the cleaned-up argument vector. The `Ice.Application` class is described in the server-side language mapping chapters.

30.8 The Ice.ProgramName property

For C++, Python, and Ruby, `initialize` sets the `Ice.ProgramName` property to the name of the current program (`argv[0]`). In C#, `initialize` sets `Ice.ProgramName` to the value of `System.AppDomain.CurrentDomain.FriendlyName`.

Your application code can read this property and use it for activities such as logging diagnostic or trace messages. (See Section 30.9.1 for how to access the property value in your program.)

Even though `Ice.ProgramName` is initialized for you, you can still override its value from a configuration file or by setting the property on the command line.

For Java, the program name is not supplied as part of the argument vector—if you want to use the `Ice.ProgramName` property in your application, you must set it before initializing a communicator.

30.9 Using Properties Programmatically

The Ice property mechanism is useful not only to configure Ice, but you can also use it as the configuration mechanism for your own applications. You can use the same configuration file and command-line mechanism to set application-specific properties. For example, we could introduce a property to control the maximum file size for our file system application:

```
# Configuration file for file system application

Filesystem.MaxFileSize=1024    # Max file size in kB
```

The Ice run time stores the `Filesystem.MaxFileSize` property like any other property and makes it accessible via the `Properties` interface.

To access property values from within your program, you need to acquire the communicator's properties by calling `getProperties`:

```
module Ice {

    local interface Properties; // Forward declaration

    local interface Communicator {

        Properties getProperties();

        // ...
    };
};
```

The `Properties` interface provides methods to read and write property settings:

```
module Ice {
    local dictionary<string, string> PropertyDict;

    local interface Properties {

        string getProperty(string key);
        string getPropertyWithDefault(string key, string value);
        int getPropertyAsInt(string key);
        int getPropertyAsIntWithDefault(string key, int value);
        PropertyDict getPropertiesForPrefix(string prefix);

        void setProperty(string key, string value);

        StringSeq getCommandLineOptions();
        StringSeq parseCommandLineOptions(string prefix,
                                          StringSeq options);
        StringSeq parseIceCommandLineOptions(StringSeq options);

        void load(string file);

        Properties clone();
    };
};
```

30.9.1 Reading Properties

The operations to read property values behave as follows:

- `getProperty`

  This operation returns the value of the specified property. If the property is not set, the operation returns the empty string.
- `getPropertyWithDefault`

  This operation returns the value of the specified property. If the property is not set, the operation returns the supplied default value.
- `getPropertyAsInt`

  This operation returns the value of the specified property as an integer. If the property is not set or contains a string that does not parse as an integer, the operation returns zero.
- `getPropertyAsIntWithDefault`

  This operation returns the value of the specified property as an integer. If the property is not set or contains a string that does not parse as an integer, the operation returns the supplied default value.
- `getPropertiesForPrefix`

  This operation returns all properties that begin with the specified prefix as a dictionary of type `PropertyDict`. This operation is useful if you want to extract the properties for a specific subsystem. For example,

  ```
  getPropertiesForPrefix("Filesystem")
  ```

  returns all properties that start with the prefix `Filesystem`, such as `Filesystem.MaxFileSize`. You can then use the usual dictionary lookup operations to extract the properties of interest from the returned dictionary.

With these lookup operations, using application-specific properties now becomes the simple matter of initializing a communicator as usual, getting access to the communicator's properties, and examining the desired property value. For example (in C++):

```
// ...

Ice::CommunicatorPtr ic;

// ...

ic = Ice::initialize(argc, argv);

// Get the maximum file size.
//
Ice::PropertiesPtr props = ic->getProperties();
Ice::Int maxSize
```

```
    = props->getPropertyAsIntWithDefault("Filesystem.MaxFileSize",
                                         1024);

// ...
```

Assuming that you have created a configuration file that sets the `Filesystem.MaxFileSize` property (and set the **`ICE_CONFIG`** variable or the **`--Ice.Config`** option accordingly), your application will pick up the configured value of the property. Refer to Section 30.5 for more information on loading a configuration file using **`ICE_CONFIG`** or **`--Ice.Config`**.

30.9.2 Setting Properties

The `setProperty` operation sets a property to the specified value. (You can clear a property by setting it to the empty string.) For properties that control the Ice run time and its services (that is, properties that start with one of the reserved prefixes, such as `Ice`, `Glacier2`, etc.), this operation is useful only if you call it *before* you call `initialize`. This is because property values are usually read by the Ice run time only once, when you call `initialize`, so the Ice run time does not pay attention to a property value that is changed after you have initialized a communicator. Of course, this begs the question of how you can set a property value and have it also recognized by a communicator.

To permit you to set properties before initializing a communicator, the Ice run time provides an overloaded helper function called `createProperties` that creates a property set. In C++, the function is in the `Ice` namespace:

```
namespace Ice {

PropertiesPtr createProperties(const StringConverterPtr& = 0);
PropertiesPtr createProperties(StringSeq&,
                               const PropertiesPtr& = 0,
                               const StringConverterPtr& = 0);
PropertiesPtr createProperties(int&, char*[],
                               const PropertiesPtr& = 0,
                               const StringConverterPtr& = 0);

}
```

The `StringConverter` parameter allows you to parse properties whose values contain non-ASCII characters and to correctly convert these characters into the native codeset. (See Section 32.24 for details.) The converter that is passed to

`createProperties` remains attached to the returned property set for the life time of the property set.

The function is overloaded to accept either an `argc`/`argv` pair, or a `StringSeq` (see Section 30.9.3 for details).

In Java, the functions are static methods of the `Util` class inside the `Ice` package:

```
package Ice;

public final class Util
{
    public static Properties
    createProperties();

    public static Properties
    createProperties(StringSeqHolder args);

    public static Properties
    createProperties(StringSeqHolder args, Properties defaults);

    public static Properties
    createProperties(String[] args);

    public static Properties
    createProperties(String[] args, Properties defaults);

    // ...
}
```

In C#, the `Util` class in the `Ice` namespace supplies equivalent methods:

```
namespace Ice {
    public sealed class Util {
        public static Properties createProperties();
        public static Properties
                createProperties(ref string[] args);
        public static Properties
                createProperties(ref string[] args,
                                 Properties defaults);
    }
}
```

The Python and Ruby methods reside in the `Ice` module:

```
def createProperties(args=[], defaults=None)
```

In PHP, use the `Ice_createProperties` method:

```
function Ice_createProperties(args=array(), defaults=null)
```

As for `initialize` (see Section 32.3), `createProperties` strips Ice-related command-line options from the passed argument vector. (For Java, only the versions that accept a `StringSeqHolder` do this.)

The functions behave as follows:

- The parameter-less version of `createProperties` simply creates an empty property set. It does *not* check **`ICE_CONFIG`** for a configuration file to parse.
- The other overloads of `createProperties` accept an argument vector and a default property set. The returned property set contains all the property settings that are passed as the default, plus any property settings in the argument vector. If the argument vector sets a property that is also set in the passed default property set, the setting in the argument vector overrides the default.

  The overloads that accept an argument vector also look for the **`--Ice.Config`** option; if the argument vector specifies a configuration file, the configuration file is parsed. The order of precedence of property settings, from lowest to highest, is:

  - Property settings passed in the default parameter
  - Property settings set in the configuration file
  - Property settings in the argument vector.

  The overloads that accept an argument vector also look for the setting of the **`ICE_CONFIG`** environment variable and, if that variable specifies a configuration file, parse that file. (However, an explicit **`--Ice.Config`** option in the argument vector or the `defaults` parameter overrides any setting of the **`ICE_CONFIG`** environment variable.)

`createProperties` is useful if you want to ensure that a property is set to a particular value, regardless of any setting of that property in a configuration file or in the argument vector. Here is a C++ example:

```
// Get the initialized property set.
//
Ice::PropertiesPtr props = Ice::createProperties(argc, argv);

// Make sure that network and protocol tracing are off.
//
props->setProperty("Ice.Trace.Network", "0");
props->setProperty("Ice.Trace.Protocol", "0");
```

```
// Initialize a communicator with these properties.
//
Ice::InitializationData id;
id.properties = props;
Ice::CommunicatorPtr ic = Ice::initialize(id);

// ...
```

The equivalent Java code looks as follows:

```
Ice.StringSeqHolder argsH = new Ice.StringSeqHolder(args);
Ice.Properties properties = Ice.Util.createProperties(argsH);
properties.setProperty("Ice.Warn.Connections", "0");
properties.setProperty("Ice.Trace.Protocol", "0");
Ice.InitializationData id = new Ice.InitializationData();
id.properties = properties;
communicator = Ice.Util.initialize(id);
```

We first convert the argument array to an initialized `StringSeqHolder`. This is necessary so `createProperties` can strip Ice-specific settings. In that way, we first obtain an initialized property set, then override the settings for the two tracing properties, and then set the properties in the `InitializationData` structure.

The equivalent Python code is shown next:

```
props = Ice.createProperties(sys.argv)
props.setProperty("Ice.Trace.Network", "0")
props.setProperty("Ice.Trace.Protocol", "0")
id = Ice.InitializationData()
id.properties = props
ic = Ice.initialize(id)
```

This is the equivalent code in Ruby:

```
props = Ice::createProperties(ARGV)
props.setProperty("Ice.Trace.Network", "0")
props.setProperty("Ice.Trace.Protocol", "0")
id = Ice::InitializationData.new
id.properties = props
ic = Ice::initialize(id)
```

Finally, we present the code in PHP:

```
$props = Ice_createProperties($args);
$props->setProperty("Ice.Trace.Network", "0");
$props->setProperty("Ice.Trace.Protocol", "0");
$id = new Ice_InitializationData();
$id->properties = $props;
$ic = Ice_initialize($id);
```

30.9.3 Parsing Properties

The Properties interface provides three operations to convert and parse properties:

- `getCommandLineOptions`

  This operation converts an initialized set of properties into a sequence of equivalent command-line options. For example, if you have set the `Filesystem.MaxFileSize` property to 1024 and call `getCommandLineOptions`, the setting is returned as the string `"Filesystem.MaxFileSize=1024"`. This operation is useful for diagnostic purposes, for example, to dump the setting of all properties to a logging facility (see Section 32.19), or if you want to fork a new process with the same property settings as the current process.

- `parseCommandLineOptions`

  This operation examines the passed argument vector for command-line options that have the specified prefix. Any options that match the prefix are converted to property settings (that is, they initialize the corresponding properties). The operation returns an argument vector that contains all those options that were *not* converted (that is, those options that did not match the prefix).

  Because `parseCommandLineOptions` expects a sequence of strings, but C++ programs are used to dealing with `argc` and `argv`, Ice provides two utility functions that convert an `argc`/`argv` vector into a sequence of strings and vice-versa:

  ```
  namespace Ice {

      StringSeq argsToStringSeq(int argc, char* argv[]);

      void stringSeqToArgs(const StringSeq& args,
                           int& argc, char* argv[]);
  ```

```
}
```

You need to use `parseCommandLineOptions` (and the utility functions) if you want to permit application-specific properties to be set from the command line. For example, to permit the **`--Filesystem.MaxFileSize`** option to be used on the command line, we need to initialize our program as follows:

```
int
main(int argc, char* argv[])
{
    // Create an empty property set.
    //
    Ice::PropertiesPtr props = Ice::createProperties();

    // Convert argc/argv to a string sequence.
    //
    Ice::StringSeq args = Ice::argsToStringSeq(argc, argv);

    // Strip out all options beginning with --Filesystem.
    //
    args = props->parseCommandLineOptions("Filesystem", args);

    // args now contains only those options that were not
    // stripped. Any options beginning with --Filesystem have
    // been converted to properties.

    // Convert remaining arguments back to argc/argv vector.
    //
    Ice::stringSeqToArgs(args, argc, argv);

    // Initialize communicator.
    //
    Ice::InitializationData id;
    id.properties = props;
    Ice::CommunicatorPtr ic = Ice::initialize(argc, argv, id);

    // At this point, argc/argv only contain options that
    // set neither an Ice property nor a Filesystem property,
    // so we can parse these options as usual.
    //
    // ...
}
```

Using this code, any options beginning with **`--Filesystem`** are converted to properties and are available via the property lookup operations as usual. The call to `initialize` then removes any Ice-specific command-line

options so, once the communicator is created, `argc`/`argv` only contains options and arguments that are not related to setting either a filesystem or an Ice property.

An easier way to achieve the same thing is to use the overload of `Ice::initialize` that accepts a string sequence, instead of an `argc`/`argv` pair:

```
int
main(int argc, char* argv[])
{
    // Create an empty property set.
    //
    Ice::PropertiesPtr props = Ice::createProperties();

    // Convert argc/argv to a string sequence.
    //
    Ice::StringSeq args = Ice::argsToStringSeq(argc, argv);

    // Strip out all options beginning with --Filesystem.
    //
    args = props->parseCommandLineOptions("Filesystem", args);

    // args now contains only those options that were not
    // stripped. Any options beginning with --Filesystem have
    // been converted to properties.

    // Initialize communicator.
    //
    Ice::InitializationData id;
    id.properties = props;
    Ice::CcommunicatorPtr ic = Ice::initialize(args, id);

    // At this point, args only contains options that
    // set neither an Ice property nor a Filesystem property,
    // so we can parse these options as usual.
    //
    // ...
}
```

This version of the code avoids having to convert the string sequence back into an `argc`/`argv` pair before calling `Ice::initialize`.

- `parseIceCommandLineOptions`

  This operation behaves like `parseCommandLineOptions`, but removes the reserved Ice-specific options from the argument vector (see Section 30.2.2). It

is used internally by the Ice run time to parse Ice-specific options in `initialize`.

30.9.4 Utility Operations

The `Properties` interface provides two utility operations:

- `clone`

  This operation makes a copy of an existing property set. The copy contains exactly the same properties and values as the original.

- `load`

  This operation accepts a path name to a configuration file and initializes the property set from that file. If the specified file cannot be read (for example, because it does not exist or the caller does not have read permission), the operation throws a `FileException`.

  In Java, the given path name can refer to a class loader resource or a regular file (see Section 30.3).

These operations are useful if you need to work with multiple communicators that use different property sets.

30.10 Unused Properties

The property `Ice.Warn.UnusedProperties`, when set to a non-zero value, causes the Ice run time to emit a warning for properties that were set but not read when you destroy a communicator. Setting this property is useful to detect mis-spelled properties, such as `Filesystem.MaxFilSize`. By default, the warning is disabled.

30.11 Summary

The Ice property mechanism provides a simple way to configure Ice by setting properties in configuration files or on the command line. This also applies to your own applications: you can easily use the `Properties` interface to access application-specific properties that you have created for your own needs. The API to access property values is small and simple, making it easy to retrieve property values at run time, yet is flexible enough to allow you to work with different property sets and configuration files if the need arises.

Chapter 31
Threads and Concurrency with C++

31.1 Chapter Overview

This chapter presents the C++ threading and signal handling abstractions that are provided by Ice. (For other language mappings, Ice uses the built-in threading and synchronization facilities.) We briefly describe how to use each of the available synchronization primitives (mutexes and monitors). We then cover how to create, control, and destroy threads. The threading discussion concludes with a brief example that shows how to create a thread-safe producer-consumer application that uses several threads. Finally, we introduce a portable abstraction for handling signals and signal-like events.

31.2 Introduction

Threading and concurrency control vary widely with different operating systems. To make threads programming easier and portable, Ice provides a simple thread abstraction layer that allows you to write portable source code regardless of the underlying platform. In this chapter, we take a closer look at the threading and concurrency control mechanisms in Ice for C++.

Note that we assume that you are familiar with light-weight threads and concurrency control. (See [8] for an excellent treatment of programming with

threads.) Also see Section 32.10, which provides a language-neutral introduction to the Ice threading model.

31.3 Library Overview

The Ice threading library provides the following thread-related abstractions:

- mutexes
- recursive mutexes
- monitors
- a thread abstraction that allows you to create, control, and destroy threads

The synchronization primitives permit you to implement concurrency control at different levels of granularity. In addition, the thread abstraction allows you to, for example, create a separate thread that can respond to GUI or other asynchronous events. All of the threading APIs are part of the `IceUtil` namespace.

31.4 Mutexes

The class `IceUtil::Mutex` (defined in `IceUtil/Mutex.h`) provides a simple non-recursive mutual exclusion mechanism:

```
namespace IceUtil {
    enum MutexProtocol { PrioInherit, PrioNone };

    class Mutex {
    public:
        Mutex();
        Mutex(MutexProtocol p);
        ~Mutex();

        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<Mutex> Lock;
        typedef TryLockT<Mutex> TryLock;
    };
}
```

The member functions of this class work as follows:

- `Mutex()`
  `Mutex(MutexProtocol p)`

  You can optionally specify a mutex protocol when you construct a mutex. The mutex protocol controls how the mutex behaves with respect to thread priorities. Default-constructed mutexes use a system-wide default. See Section 31.9 for more detail.
- `lock`

  The `lock` function attempts to acquire the mutex. If the mutex is already locked, it suspends the calling thread until the mutex becomes available. The call returns once the calling thread has acquired the mutex.
- `tryLock`

  The `tryLock` function attempts to acquire the mutex. If the mutex is available, the call returns with the mutex locked and returns `true`. Otherwise, if the mutex is locked by another thread, the call returns `false`.
- `unlock`

  The `unlock` function unlocks the mutex.

Note that `IceUtil::Mutex` is a non-recursive mutex implementation. This means that you must adhere to the following rules:

- Do not call `lock` on the same mutex more than once from a thread. The mutex is not recursive so, if the owner of a mutex attempts to lock it a second time, the behavior is undefined.
- Do not call `unlock` on a mutex unless the calling thread holds the lock. Calling `unlock` on a mutex that is not currently held by any thread, or calling `unlock` on a mutex that is held by a different thread, results in undefined behavior.

31.4.1 Thread-Safe File Access for the Filesystem Application

Recall that the implementation of the `read` and `write` operations for our file system server in Section 9.2.3 is not thread safe:

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    return _lines;      // Not thread safe!
}

void
```

```
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
    _lines = text;      // Not thread safe!
}
```

The problem here is that, if we receive concurrent invocations of `read` and `write`, one thread will be assigning to the `_lines` vector while another thread is reading that same vector. The outcome of such concurrent data access is undefined; to avoid the problem, we need to serialize access to the `_lines` member with a mutex. We can make the mutex a data member of the `FileI` class and lock and unlock it in the `read` and `write` operations:

```
#include <IceUtil/Mutex.h>
// ...

namespace Filesystem {
    // ...

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
    public:
        // As before...
    private:
        Lines _lines;
        IceUtil::Mutex _fileMutex;
    };
    // ...
}

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    _fileMutex.lock();
    Lines l = _lines;
    _fileMutex.unlock();
    return l;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
```

```
    _fileMutex.lock();
    _lines = text;
    _fileMutex.unlock();
}
```

The `FileI` class here is identical to the implementation in Section 9.2.2, except that we have added the `_fileMutex` data member. The `read` and `write` operations lock and unlock the mutex to ensure that only one thread can read or write the file at a time. Note that, by using a separate mutex for each `FileI` instance, it is still possible for multiple threads to concurrently read or write files, as long as they each access a *different* file. Only concurrent accesses to the *same* file are serialized.

The implementation of `read` is somewhat awkward here: we must make a local copy of the file contents while we are holding the lock and return that copy. Doing so is necessary because we must unlock the mutex before we can return from the function. However, as we will see in the next section, the copy can be avoided by using a helper class that unlocks the mutex automatically when the function returns.

31.4.2 Guaranteed Unlocking of Mutexes

Using the raw `lock` and `unlock` operations on mutexes has an inherent problem: if you forget to unlock a mutex, your program will deadlock. Forgetting to unlock a mutex is easier than you might suspect, for example:

```
Filesystem::Lines
Filesystem::File::read(const Ice::Current&) const
{
    _fileMutex.lock();                  // Lock the mutex
    Lines l = readFileContents();       // Read from database
    _fileMutex.unlock();                // Unlock the mutex
    return l;
}
```

Assume that we are keeping the contents of the file on secondary storage, such as a database, and that the `readFileContents` function accesses the file. The code is almost identical to the previous example but now contains a latent bug: if `readFileContents` throws an exception, the `read` function terminates without ever unlocking the mutex. In other words, this implementation of `read` is not exception-safe.

The same problem can easily arise if you have a larger function with multiple return paths. For example:

```
void
SomeClass::someFunction(/* params here... */)
{
    _mutex.lock();                      // Lock a mutex

    // Lots of complex code here...

    if (someCondition) {
        // More complex code here...
        return;                         // Oops!!!
    }

    // More code here...

    _mutex.unlock();                    // Unlock the mutex
}
```

In this example, the early return from the middle of the function leaves the mutex locked. Even though this example makes the problem quite obvious, in large and complex pieces of code, both exceptions and early returns can cause hard-to-track deadlock problems. To avoid this, the `Mutex` class contains two type definitions for helper classes, called `Lock` and `TryLock`:

```
namespace IceUtil {

    class Mutex {
        // ...

        typedef LockT<Mutex> Lock;
        typedef TryLockT<Mutex> TryLock;
    };
}
```

`LockT` and `TryLockT` are simple templates that primarily consist of a constructor and a destructor; the `LockT` constructor calls `lock` on its argument, the `TryLockT` constructor calls tryLock on its argument. The destructors call `unlock` if the mutex is lock when the template goes out of scope. By instantiating a local variable of type `Lock` or `TryLock`, we can avoid the deadlock problem entirely:[1]

1. This is an example of the *RAII* (*Resource Acquisition Is Initialization*) idiom [20].

```
void
SomeClass::someFunction(/* params here... */)
{
    IceUtil::Mutex::Lock lock(_mutex);  // Lock a mutex

    // Lots of complex code here...

    if (someCondition) {
        // More complex code here...
        return;                         // No problem
    }

    // More code here...

}   // Destructor of lock unlocks the mutex
```

On entry to `someFunction`, we instantiate a local variable `lock`, of type `IceUtil::Mutex::Lock`. The constructor of `lock` calls `lock` on the mutex so the remainder of the function is inside a critical region. Eventually, `someFunction` returns, either via an ordinary return (in the middle of the function or at the end) or because an exception was thrown somewhere in the function body. Regardless of how the function terminates, the C++ run time unwinds the stack and calls the destructor of `lock`, which unlocks the mutex, so we cannot get trapped by the deadlock problem we had previously.

Both the `Lock` and `TryLock` templates have a few member functions:

- `void acquire() const;`

  This function attempts to acquire the lock and blocks the calling thread until the lock becomes available. If the caller calls `acquire` on a mutex it has locked previously, the function throws `ThreadLockedException`.

- `bool tryAcquire() const;`

  This function attempts to acquire the mutex. If the mutex can be acquired, it returns true with the mutex locked; if the mutex cannot be acquired, it returns false. If the caller calls `tryAcquire` on a mutex it has locked previously, the function throws `ThreadLockedException`.

- `void release() const;`

  This function releases a previously locked mutex. If the caller calls release on a mutex it has unlocked previously, the function throws `ThreadLockedException`.

- `bool acquired() const;`

  This function returns true if the caller has locked the mutex previously and false, otherwise. If you use the `TryLock` template, you must call `acquired` after instantiating the template to test whether the lock actually was acquired.

These functions are useful if you want to use the `Lock` and `TryLock` templates for guaranteed unlocking, but need to temporarily release the lock:

```
{
    IceUtil::Mutex::TryLock m(someMutex);

    if (m.acquired())
    {

        // Got the lock, do processing here...

        if (release_condition) {
            m.release();
        }

        // Mutex is now unlocked, someone else can lock it.
        // ...

        m.acquire(); // Block until mutex becomes available.

        // ...

        if (release_condition) {
            m.release();
        }

        // Mutex is now unlocked, someone else can lock it.

        // ...

        // Spin on the mutex until it becomes available.
        while (!m.tryLock()) {
            // Do some other processing here...
        }

        // Mutex locked again at this point.
```

```
        // ...
    }

} // Close scope, m is unlocked by its destructor.
```

You should make it a habit to always use the `Lock` and `TryLock` helpers instead of calling `lock` and `unlock` directly. Doing so results in code that is easier to understand and maintain.

Using the `Lock` helper, we can rewrite the implementation of our `read` and `write` operations as follows:

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current&) const
{
    IceUtil::Mutex::Lock lock(_fileMutex);
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current&)
{
    IceUtil::Mutex::Lock lock(_fileMutex);
    _lines = text;
}
```

Note that this also eliminates the need to make a copy of the `_lines` data member: the return value is initialized under protection of the mutex and cannot be modified by another thread once the destructor of `lock` unlocks the mutex.

31.5 Recursive Mutexes

As we saw on page 883, a non-recursive mutex cannot be locked more than once, even by the thread that holds the lock. This frequently becomes a problem if a program contains a number of functions, each of which must acquire a mutex, and you want to call one function as part of the implementation of another function:

```
IceUtil::Mutex _mutex;

void
f1()
{
    IceUtil::Mutex::Lock lock(_mutex);
```

```
    // ...
}

void
f2()
{
    IceUtil::Mutex::Lock lock(_mutex);
    // Some code here...

    // Call f1 as a helper function
    f1();                                // Deadlock!

    // More code here...
}
```

`f1` and `f2` each correctly lock the mutex before manipulating data but, as part of its implementation, `f2` calls `f1`. At that point, the program deadlocks because `f2` already holds the lock that `f1` is trying to acquire. For this simple example, the problem is obvious. However, in complex systems with many functions that acquire and release locks, it can get very difficult to track down this kind of situation: the locking conventions are not manifest anywhere but in the source code and each caller must know which locks to acquire (or not to acquire) before calling a function. The resulting complexity can quickly get out of hand.

Ice provides a recursive mutex class `RecMutex` (defined in `IceUtil/RecMutex.h`) that avoids this problem:

```
namespace IceUtil {

    class RecMutex {
    public:
        RecMutex();
        RecMutex(MutexProtocol p);
        ~RecMutex();

        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<RecMutex> Lock;
        typedef TryLockT<RecMutex> TryLock;
    };
}
```

Note that the signatures of the operations are the same as for `IceUtil::Mutex`. However, `RecMutex` implements a recursive mutex:

- `RecMutex()`
  `RecMutex(MutexProtocol p)`

  You can optionally specify a mutex protocol when you construct a mutex. The mutex protocol controls how the mutex behaves with respect to thread priorities. Default-constructed mutexes use a system-wide default. See Section 31.9 for more detail.

- `lock`

  The `lock` function attempts to acquire the mutex. If the mutex is already locked by another thread, it suspends the calling thread until the mutex becomes available. If the mutex is available or is already locked by the calling thread, the call returns immediately with the mutex locked.

- `tryLock`

  The `tryLock` function works like `lock`, but, instead of blocking the caller, it returns `false` if the mutex is locked by another thread. Otherwise, the return value is `true`.

- `unlock`

  The `unlock` function unlocks the mutex.

As for non-recursive mutexes, you must adhere to a few simple rules for recursive mutexes:

- Do not call `unlock` on a mutex unless the calling thread holds the lock.
- You must call `unlock` as many times as you called `lock` for the mutex to become available to another thread. (Internally, a recursive mutex is implemented with a counter that is initialized to zero. Each call to `lock` increments the counter and each call to `unlock` decrements the counter; the mutex is made available to another thread when the counter returns to zero.)

Using recursive mutexes, the code fragment on page 889 works correctly:

```
#include <IceUtil/RecMutex.h>
// ...

IceUtil::RecMutex _mutex;       // Recursive mutex

void
f1()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // ...
}
```

```
void
f2()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // Some code here...

    // Call f1 as a helper function
    f1();                                // Fine

    // More code here...
}
```

Note that the type of the mutex is now `RecMutex` instead of `Mutex`, and that we are using the `Lock` type definition provided by the `RecMutex` class, not the one provided by the `Mutex` class.

31.6 Monitors

Mutexes implement a simple mutual exclusion mechanism that allows only a single thread (or, in the case of read-write mutexes, a single writer thread or multiple reader threads) to be active in a critical region at a time. In particular, for another thread to enter the critical region, another thread must leave it. This means that, with mutexes, it is impossible to suspend a thread inside a critical region and have that thread wake up again at a later time, for example, when a condition becomes true.

To address this problem, Ice provides a monitor. Briefly, a monitor is a synchronization mechanism that protects a critical region: as for a mutex, only one thread may be active at a time inside the critical region. However, a monitor allows you to suspend a thread inside the critical region; doing so allows another thread to enter the critical region. The second thread can either leave the monitor (thereby unlocking the monitor), or it can suspend itself inside the monitor; either way, the original thread is woken up and continues execution inside the monitor. This extends to any number of threads, so several threads can be suspended inside a monitor.[2]

Monitors provide a more flexible mutual exclusion mechanism than mutexes because they allow a thread to check a condition and, if the condition is false, put itself to sleep; the thread is woken up by some other thread that has changed the condition.

31.6.1 The `Monitor` Class

Ice provides monitors with the `IceUtil::Monitor` class (defined in `IceUtil/Monitor.h`):

```
namespace IceUtil {

    template <class T>
    class Monitor {
    public:
        void lock() const;
        void unlock() const;
        bool tryLock() const;

        void wait() const;
        bool timedWait(const Time&) const;
        void notify();
        void notifyAll();

        typedef LockT<Monitor<T> > Lock;
        typedef TryLockT<Monitor<T> > TryLock;
    };
}
```

Note that `Monitor` is a template class that requires either `Mutex` or `RecMutex` as its template parameter. (Instantiating a `Monitor` with a `RecMutex` makes the monitor recursive.)

The member functions behave as follows:

- `lock`

  This function attempts to lock the monitor. If the monitor is currently locked by another thread, the calling thread is suspended until the monitor becomes available. The call returns with the monitor locked.

- `tryLock`

  This function attempts to lock a monitor. If the monitor is available, the call returns `true` with the monitor locked. If the monitor is locked by another thread, the call returns `false`.

2. The monitors provided by Ice have *Mesa* semantics, so called because they were first implemented by the Mesa programming language [12]. Mesa monitors are provided by a number of languages, including Java and Ada. With Mesa semantics, the signalling thread continues to run and another thread gets to run only once the signalling thread suspends itself or leaves the monitor.

- `unlock`

  This function unlocks a monitor. If other threads are waiting to enter the monitor (are blocked inside a call to `lock`), one of the threads is woken up and locks the monitor.

- `wait`

  This function suspends the calling thread and, at the same time, releases the lock on the monitor. A thread suspended inside a call to `wait` can be woken up by another thread that calls `notify` or `notifyAll`. When the call returns, the suspended thread resumes execution with the monitor locked.

- `timedWait`

  This function suspends the calling thread for up to the specified timeout. If another thread calls `notify` or `notifyAll` and wakes up the suspended thread before the timeout expires, the call returns `true` and the suspended thread resumes execution with the monitor locked. Otherwise, if the timeout expires, the function returns `false`. (See Appendix F.12 for a description of the `Time` class.)

- `notify`

  This function wakes up a single thread that is currently suspended in a call to `wait` or `timedWait`. If no thread is suspended in a call to `wait` or `timedWait` at the time `notify` is called, the notification is lost (that is, calls to `notify` are *not* remembered if there is no thread to be woken up).

  Note that notifying does not run another thread immediately. Another thread gets to run only once the notifying thread either calls `wait` or `timedWait` or unlocks the monitor (Mesa semantics).

- `notifyAll`

  This function wakes up all threads that are currently suspended in a call to `wait` or `timedWait`. As for `notify`, calls to `notifyAll` are lost if no threads are suspended at the time.

  As for `notify`, `notifyAll` causes other threads to run only once the notifying thread has either called `wait` or `timedWait` or unlocked the monitor (Mesa semantics).

You must adhere to a few rules for monitors to work correctly:

- Do not call `unlock` unless you hold the lock. If you instantiate a monitor with a recursive mutex, you get recursive semantics, that is, you must call

unlock as many times as you have called lock (or tryLock) for the monitor to become available.

- Do not call wait or timedWait unless you hold the lock.
- Do not call notify or notifyAll unless you hold the lock.
- When returning from a wait call, you *must* re-test the condition before proceeding (see page 897).

31.6.2 Using Monitors

To illustrate how to use a monitor, consider a simple unbounded queue of items. A number of producer threads add items to the queue, and a number of consumer threads remove items from the queue. If the queue becomes empty, consumers must wait until a producer puts a new item on the queue. The queue itself is a critical region, that is, we cannot allow a producer to put an item on the queue while a consumer is removing an item. Here is a very simple implementation of a such a queue:

```
template<class T> class Queue {
public:
    void put(const T& item) {
        _q.push_back(item);
    }

    T get() {
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

As you can see, producers call the put method to enqueue an item, and consumers call the get method to dequeue an item. Obviously, this implementation of the queue is not thread-safe and there is nothing to stop a consumer from attempting to dequeue an item from an empty queue.

Here is a version of the queue that uses a monitor to suspend a consumer if the queue is empty:

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T& item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

Note that the `Queue` class now inherits from `IceUtil::Monitor<IceUtil::Mutex>`, that is, `Queue` *is-a* monitor.

Both the `put` and `get` methods lock the monitor when they are called. As for mutexes, instead of calling `lock` and `unlock` directly, we are using the `Lock` helper which automatically locks the monitor when it is instantiated and unlocks the monitor again when it is destroyed.

The `put` method first locks the monitor and then, now being in sole possession of the critical region, enqueues an item. Before returning (thereby unlocking the monitor), `put` calls `notify`. The call to `notify` will wake up any consumer thread that may be asleep in a `wait` call to inform the consumer that an item is available.

The `get` method also locks the monitor and then, before attempting to dequeue an item, tests whether the queue is empty. If so, the consumer calls `wait`. This suspends the consumer inside the wait call and unlocks the monitor, so a producer can enter the monitor to enqueue an item. Once that happens, the producer calls `notify`, which causes the consumer's `wait` call to complete, with the monitor again locked for the consumer. The consumer now dequeues an item and returns (thereby unlocking the monitor).

For this machinery to work correctly, the implementation of `get` does two things:

- `get` tests whether the queue is empty *after* acquiring the lock.
- `get` re-tests the condition in a loop around the call to `wait`; if the queue is still empty after `wait` returns, the `wait` call is re-entered.

You *must* always write your code to follow the same pattern:

- *Never* test a condition unless you hold the lock.
- *Always* re-test the condition in a loop around `wait`. If the test still shows the wrong outcome, call `wait` again.

Not adhering to these conditions will eventually result in a thread accessing shared data when it is not in its expected state, for the following reasons:

1. If you test a condition without holding the lock, there is nothing to prevent another thread from entering the monitor and changing its state before you can acquire the lock. This means that, by the time you get around to locking the monitor, the state of the monitor may no longer be in agreement with the result of the test.
2. Some thread implementations suffer from a problem known as *spurious wake-up*: occasionally, more than one thread may wake up in response to a call to `notify`, or a thread may wake up without any call to `notify` at all. As a result, each thread that returns from a call to `wait` must re-test the condition to ensure that the monitor is in its expected state: the fact that `wait` returns does *not* indicate that the condition has changed.

31.6.3 Efficient Notification

The previous implementation of our thread-safe queue on page 896 unconditionally notifies a waiting reader whenever a writer deposits an item into the queue. If no reader is waiting, the notification is lost and does no harm. However, unless there is only a single reader and writer, many notifications will be sent unnecessarily, causing unwanted overhead.

Here is one way to fix the problem:

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T& item) {
```

```
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_q.size() == 1)
            notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

The only difference between this code and the implementation on page 896 is that a writer calls `notify` only if the queue length has just changed from empty to non-empty. That way, unnecessary `notify` calls are never made. However, this approach works only for a single reader thread. To see why, consider the following scenario:

1. Assume that the queue currently contains a number of items and that we have five reader threads.
2. The five reader threads continue to call `get` until the queue becomes empty and all five readers are waiting in `get`.
3. The scheduler schedules a writer thread. The writer finds the queue empty, deposits an item, and wakes up a single reader thread.
4. The awakened reader thread dequeues the single item on the queue.
5. The reader calls `get` a second time, finds the queue empty, and goes to sleep again.

The net effect of this is that there is a good chance that only one reader thread will ever be active; the other four reader threads end up being permanently asleep inside the `get` method.

One way around this problem is call `notifyAll` instead of `notify` once the queue length exceeds a certain amount, for example:

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T& item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_q.size() >= _wakeupThreshold)
            notifyAll();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
    const int _wakeupThreshold = 100;
};
```

Here, we have added a private data member `_wakeupThreshold`; a writer wakes up *all* waiting readers once the queue length exceeds the threshold, in the expectation that all the readers will consume items more quickly than they are produced, thereby reducing the queue length below the threshold again.

This approach works, but has drawbacks as well:

- The appropriate value of `_wakeupThreshold` is difficult to determine and sensitive to things such as speed and number of processors and I/O bandwidth.
- If multiple readers are asleep, they are all made runnable by the thread scheduler once a writer calls `notifyAll`. On a multiprocessor machine, this may result in all readers running at once (one per CPU). However, as soon as the readers are made runnable, each of them attempts to reacquire the mutex that protects the monitor before returning from `wait`. Of course, only one of the readers actually succeeds and the remaining readers are suspended again, waiting for the mutex to become available. The net result is a large number of thread context switches as well as repeated and unnecessary locking of the system bus.

A better option than calling `notifyAll` is to wake up waiting readers one at a time. To do this, we keep track of the number of waiting readers and call `notify` only if a reader needs to be woken up:

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    Queue() : _waitingReaders(0) {}

    void put(const T& item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_waitingReaders)
            notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0) {
            try {
                ++_waitingReaders;
                wait();
                --_waitingReaders;
            } catch (...) {
                --_waitingReaders;
                throw;
            }
        }
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
    short _waitingReaders;
};
```

This implementation uses a member variable `_waitingReaders` to keep track of the number of readers that are suspended. The constructor initializes the variable to zero and the implementation of `get` increments and decrements the variable around the call to `wait`. Note that these statements are enclosed in a `try`–`catch` block; this ensures that the count of waiting readers remains accurate even

if `wait` throws an exception. Finally, `put` calls `notify` only if there is a waiting reader.

The advantage of this implementation is that it minimizes contention on the monitor mutex: a writer wakes up only a single reader at a time, so we do not end up with multiple readers simultaneously trying to lock the mutex. Moreover, the monitor `notify` implementation signals a waiting thread only *after* it has unlocked the mutex. This means that, when a thread wakes up from its call to `wait` and tries to reacquire the mutex, the mutex is likely to be unlocked. This results in more efficient operation because acquiring an unlocked mutex is typically very efficient, whereas forcefully putting a thread to sleep on a locked mutex is expensive (because it forces a thread context switch).

31.7 Condition Variables

Condition variables are similar to monitors in that they allow a thread to enter a critical region, test a condition, and sleep inside the critical region while releasing its lock. Another thread then is free to enter the critical region, change the condition, and eventually signal the sleeping thread, which resumes at the point where it went to sleep and with the critical region once again locked.

Note that condition variables provide subset of the functionality of monitors, so a monitor can always be used instead of a condition variable. However, condition variables are smaller, which may be important if you are seriously constrained with respect to memory.

Condition variables are provided by the `IceUtil::Cond` class. Here is its interface:

```
class Cond : private noncopyable {
public:

    Cond();
    ~Cond();

    void signal();
    void broadcast();

    template<typename Lock>
        void wait(const Lock& lock) const;
```

```
    template<typename Lock>
        bool timedWait(const Lock& lock,
                       const Time& timeout) const;
};
```

Using a condition variable is very similar to using a monitor. The main difference in the `Cond` interface is that the `wait` and `timedWait` member functions are template functions, instead of the entire class being a template. The member functions behave as follows:

- `wait`

  This function suspends the calling thread and, at the same time, releases the lock of the condition variable. A thread suspended inside a call to `wait` can be woken up by another thread that calls `signal` or `broadcast`. When `wait` completes, the suspended thread resumes execution with the lock held.

- `timedWait`

  This function suspends the calling thread for up to the specified timeout. If another thread calls `signal` or `broadcast` and wakes up the suspended thread before the timeout expires, the call returns `true` and the suspended thread resumes execution with the lock held. Otherwise, if the timeout expires, the function returns `false`.

- `signal`

  This function wakes up a single thread that is currently suspended in a call to `wait` or `timedWait`. If no thread is suspended in a call to `wait` or `timedWait` at the time `signal` is called, the signal is lost (that is, calls to `signal` are *not* remembered if there is no thread to be woken up).

  Note that signalling does not necessarily run another thread immediately; the thread calling signal may continue to run. However, depending on the threads library, signal may also cause an immediate context switch to another thread.

- `broadcast`

  This function wakes up all threads that are currently suspended in a call to `wait` or `timedWait`. As for `signal`, calls to `broadcast` are lost if no threads are suspended at the time.

You must adhere to a few rules for condition variables to work correctly:

- Do not call `wait` or `timedWait` unless you hold the lock.
- When returning from a `wait` call, you *must* re-test the condition before proceeding, just as for a monitor (see page 897).

In contrast to monitors, which require you to call `notify` and `notifyAll` with the lock held, condition variables permit you call `signal` and `broadcast` without holding the lock. Here is a code example that changes a condition and signals on a condition variable:

```
Mutex m;
Cond c;

// ...

{
     Mutex::Lock sync(m);

     // Change some condition other threads may be sleeping on...

     c.signal();

     // ...
} // m is unlocked here
```

This code is correct and will work as intended, but it is potentially inefficient. Consider the code executed by the waiting thread:

```
{
    Mutex::Lock sync(m);

    while(!condition) {
        c.wait(sync);
    }

    // Condition is now true, do
    // some processing...

} // m is unlocked here
```

Again, this code is correct and will work as intended. However, consider what can happen once the first thread calls `signal`. It is possible that the call to signal will cause an immediate context switch to the waiting thread. But, even if the thread implementation does not cause such an immediate context switch, it is possible for the signalling thread to be suspended after it has called `signal`, but before it unlocks the mutex `m`. If this happens, the following sequence of events occurs:

1. The waiting thread is still suspended inside the implementation of `wait` and is now woken up by the call to `signal`.

2. The now awake thread tries to acquire the mutex `m` but, because the signalling thread has not yet released the mutex, is suspended again waiting for the mutex to be unlocked.

3. The signal thread is scheduled again and leaves the scope enclosing `sync`, which unlocks the mutex, making the thread waiting for the mutex runnable.

4. The thread waiting for the mutex acquires the mutex and retests its condition.

While the preceding scenario is functionally correct, it is inefficient because it incurs two extra context switches between the signalling thread and the waiting thread. Because context switches are expensive, this can have quite a large impact on run-time performance, especially if the critical region is small and the condition changes frequently.

You can avoid the inefficiency by unlocking the mutex *before* calling signal:

```
Mutex m;
Cond c;

// ...

{
     Mutex::Lock sync(m);

     // Change some condition other threads may be sleeping on...

} // m is unlocked here

c.signal(); // Signal with the lock available
```

By arranging the code as shown, you avoid the additional context switches because, when the waiting thread is woken up by the call to `signal`, it succeeds in acquiring the mutex before returning from wait without being suspended and woken up again first.

As for monitors, you should exercise caution in using `broadcast`, particularly if you have many threads waiting on a condition. Condition variables suffer from the same potential problem as monitors with respect to `broadcast`, namely, that all threads that are currently suspended inside `wait` can immediately attempt to acquire the mutex, but only one of them can succeed and all other threads are suspended again. If your application is sensitive to this condition, you may want to consider waking threads in a more controlled manner, along the lines shown on page 900

31.8 Threads

As described in Section 32.10, the server-side Ice run time by default creates a thread pool for you and automatically dispatches each incoming request in its own thread. As a result, you usually only need to worry about synchronization among threads to protect critical regions when you implement a server. However, you may wish to create threads of your own. For example, you might need a dedicated thread that responds to input from a user interface. And, if you have complex and long-running operations that can exploit parallelism, you might wish to use multiple threads for the implementation of that operation.

Ice provides a simple thread abstraction that permits you to write portable source code regardless of the native threading platform. This shields you from the native underlying thread APIs and guarantees uniform semantics regardless of your deployment platform.

31.8.1 The `Thread` Class

The basic thread abstraction in Ice is provided by two classes, `ThreadControl` and `Thread` (defined in `IceUtil/Thread.h`):

```
namespace IceUtil {

    class Time;

    class ThreadControl {
    public:
#ifdef _WIN32
        typedef DWORD ID;
#else
        typedef pthread_t ID;
#endif

        ThreadControl();
#ifdef _WIN32
        ThreadControl(HANDLE, DWORD);
#else
        ThreadControl(explicit pthread_t);
#endif
        ID id() const;

        void join();
        void detach();
```

```
        static void sleep(const Time&);
        static void yield();

        bool operator==(const ThreadControl&) const;
        bool operator!=(const ThreadControl&) const;

    };

    class Thread {
    public:
        virtual void run() = 0;

        ThreadControl start(size_t stBytes = 0);
        ThreadControl start(size_t stBytes, int priority);
        ThreadControl getThreadControl() const;
        bool isAlive() const;

        bool operator==(const Thread&) const;
        bool operator!=(const Thread&) const;
        bool operator<(const Thread&) const;
    };
    typedef Handle<Thread> ThreadPtr;
}
```

The `Thread` class is an abstract base class with a pure virtual `run` method. To create a thread, you must specialize the Thread class and implement the `run` method (which becomes the starting stack frame for the new thread). Note that you must not allow any exceptions to escape from `run`. The Ice run time installs an exception handler that calls `::std::terminate` if `run` terminates with an exception.

The remaining member functions behave as follows:

- `start(size_t stBytes = 0)`
  `start(size_t stBytes, int priority)`

  This member function starts a newly-created thread (that is, calls the `run` method).

  The `stBytes` parameter specifies a stack size (in bytes) for the thread. The default value of zero creates the thread with a default stack size that is determined by the operating system.

  You can specify a priority for the thread. (If you do not supply a priority, the thread is created with the system default priority.)

  The priority value is system-dependent; on POSIX systems, the value must be a legal value for the `SCHED_RR` real-time scheduling policy. `SCHED_RR` requires root privileges. On Windows systems, the priority value is passed through to the Windows `setThreadPriority` function.

  Section 31.9 provides information about how you can deal with priority inversion.

  The return value is a `ThreadControl` object for the new thread (see Section 31.8.4).

  You can start a thread only once; calling start on an already-started thread raises `ThreadStartedException`.

  If the calling thread passes an invalid priority or, on POSIX systems, does not have root priviliges, `start` raises `ThreadSyscallException`.

- `getThreadControl`

  This member function returns a thread control object for the thread on which it is invoked (see Section 31.8.4). Calling this method before calling `start` raises a `ThreadNotStartedException`.

- `id`

  This method returns the underlying thread ID (`DWORD` for Windows and `pthread_t` for POSIX threads). This method is provided mainly for debugging purposes.[3]

3. `pthread_t` is, strictly-speaking, an opaque type, so you should not make any assumptions about what you can do with a thread ID.

- `isAlive`

  This method returns false before a thread's `start` method has been called and after a thread's `run` method has completed; otherwise, while the thread is still running, it returns true. `isAlive` is useful to implement a non-blocking join:

  ```
  ThreadPtr p = new MyThread();
  // ...
  while(p->isAlive()) {
      // Do something else...
  }
  t.join(); // Will not block
  ```

- `operator==`
  `operator!=`
  `operator<`

  These member functions compare the in-memory address of two threads. They are provided so you can use sorted STL containers with `Thread` objects.

Note that `IceUtil` also defines the type `ThreadPtr`. This is the usual reference-counted smart pointer (see Section 6.14.6) to guarantee automatic clean-up: the Thread destructor calls `delete this` once its reference count drops to zero.

31.8.2 Implementing Threads

To illustrate how to implement threads, consider the following code fragment:

```
#include <IceUtil/Thread.h>
// ...

Queue q;

class ReaderThread : public IceUtil::Thread {
    virtual void run() {
        for (int i = 0; i < 100; ++i)
            cout << q.get() << endl;
    }
};

class WriterThread : public IceUtil::Thread {
    virtual void run() {
```

```
        for (int i = 0; i < 100; ++i)
            q.put(i);
    }
};
```

This code fragment defines two classes, `ReaderThread` and `WriterThread`, that inherit from `IceUtil::Thread`. Each class implements the pure virtual `run` method it inherits from its base class. For this simple example, a writer thread places the numbers from 1 to 100 into an instance of the thread-safe `Queue` class we defined in Section 31.6, and a reader thread retrieves 100 numbers from the queue and prints them to `stdout`.

31.8.3 Creating Threads

To create a new thread, we simply instantiate the thread and call its `start` method:

```
IceUtil::ThreadPtr t = new ReaderThread;
t->start();
// ...
```

Note that we assign the return value from `new` to a smart pointer of type `ThreadPtr`. This ensures that we do not suffer a memory leak:[4]

1. When the thread is created, its reference count is set to zero.
2. Prior to calling `run` (which is called by the `start` method), `start` increments the reference count of the thread to 1.
3. For each `ThreadPtr` for the thread, the reference count of the thread is incremented by 1, and for each `ThreadPtr` that is destroyed, the reference count is decremented by 1.
4. When `run` completes, `start` decrements the reference count again and then checks its value: if the value is zero at this point, the `Thread` object deallocates itself by calling `delete this`; if the value is non-zero at this point, there are other smart pointers that reference this `Thread` object and deletion happens when the last smart pointer goes out of scope.

Note that, for all this to work, you *must* allocate your `Thread` objects on the heap—stack-allocated `Thread` objects will result in deallocation errors:

4. `ThreadPtr` is another example of an RAII class [20].

```
ReaderThread thread;
IceUtil::ThreadPtr t = &thread; // Bad news!!!
```

This is wrong because the destructor of `t` will eventually call `delete`, which has undefined behavior for a stack-allocated object.

Similarly, you *must* use a `ThreadPtr` for an allocated thread. Do not attempt to explicitly delete a a thread:

```
Thread* t = new ReaderThread();

// ...

delete t; // Disaster!
```

This will result in a double deallocation of the thread because the thread's destructor will call `delete this`.

It is legal for a thread to call `start` on itself from within its own constructor. However, if so, the thread must not be (very) short lived:

```
class ActiveObject : public Thread() {
public:
    ActiveObject() {
        start();
    }

    void done() {
        getThreadControl().join();
    }

    virtual void run() {
        // *Very* short lived...
    }
};
typedef Handle<ActiveObject> ActiveObjectPtr;

// ...

ActiveObjectPtr ao = new ActiveObject;
```

With this code, it is possible for `run` to complete before the assignment to the smart pointer `ao` completes; in that case, `start` will call `delete this;` before it returns and `ao` ends up deleting an already-deleted object. However, note that this problem can arise only if `run` is indeed *very* short-lived and moreover, the scheduler allows the newly-created thread to run to completion before the assignment of the return value of `operator new` to `ao` takes place. This is

highly unlikely to happen—if you are concerned about this scenario, do not call `start` from within a thread's own constructor. That way, the smart pointer is assigned first, and the thread started second (as in the example on page 909), so the problem cannot arise.

31.8.4 The `ThreadControl` Class

The `start` method returns an object of type `ThreadControl` (see page 905). The member functions of `ThreadControl` behave as follows:

- `ThreadControl`

  The default constructor returns a `ThreadControl` object that refers to the calling thread. This allows you to get a handle to the current (calling) thread even if you do not have saved a handle to that thread previously. For example:

  ```
  IceUtil::ThreadControl self;    // Get handle to self
  cout << self.id() << endl;      // Print thread ID
  ```

  This example also explains why we have two classes, `Thread` and `ThreadControl`: without a separate `ThreadControl`, it would not be possible to obtain a handle to an arbitrary thread. (Note that this code works even if the calling thread was not created by the Ice run time; for example, you can create a `ThreadControl` object for a thread that was created by the operating system.)

  The (implicit) copy constructor and assignment operator create a `ThreadControl` object that refers to the same underlying thread as the source `ThreadControl` object.

  Note that the constructor is overloaded. For Windows, the signature is

  ```
  ThreadControl(HANDLE, DWORD);
  ```

  For Unix, the signature is

  ```
  ThreadControl(pthread_t);
  ```

  These constructors allow you to create a `ThreadControl` object for the specified thread.

- `join`

  This method suspends the calling thread until the thread on which `join` is called has terminated. For example:

  ```
  IceUtil::ThreadPtr t = new ReaderThread; // Create a thread
  IceUtil::ThreadControl tc = t->start();  // Start it
  ```

```
tc.join();                                  // Wait for it
```

If the reader thread has finished by the time the creating thread calls `join`, the call to `join` returns immediately; otherwise, the creating thread is suspended until the reader thread terminates.

Note that the `join` method of a thread must be called from only one other thread, that is, only one thread can wait for another thread to terminate. Calling `join` on a thread from more than one other thread has undefined behavior.

Calling `join` on a thread that was previously joined with or calling `join` on a detached thread has undefined behavior.

You must join with each thread you create; failure to join with a thread has undefined behavior.

- `detach`

  This method detaches a thread. Once a thread is detached, it cannot be joined with.

  Calling `detach` on an already detached thread, or calling `detach` on a thread that was previously joined with has undefined behavior.

  Note that, if you have detached a thread, you must ensure that the detached thread has terminated before your program leaves its `main` function. This means that, because detached threads cannot be joined with, they must have a life time that is shorter than that of the main thread.

- `sleep`

  This method suspends the calling thread for the amount of time specified by the `Time` parameter (see Appendix F.12).

- `yield`

  This method causes the calling thread to relinquish the CPU, allowing another thread to run.

- `operator==`
  `operator!=`

  These operators compare thread IDs. (Note that `operator<` is *not* provided because it cannot be implemented portably.) These operators yield meaningful results only for threads that have not been detached or joined with.

As for all the synchronization primitives, you must adhere to a few rules when using threads to avoid undefined behavior:

- Do not allow `run` to throw an exception.

- Do not join with or detach a thread that you have not created yourself.
- For every thread you create, you must either join with that thread exactly once or detach it exactly once; failure to do so may cause resource leaks.
- Do not call `join` on a thread from more than one other thread.
- Do not leave `main` until all other threads you have created have terminated.
- Do not leave `main` until after you have destroyed all `Ice::Communicator` objects you have created (or use the `Ice::Application` class—see Section 8.3.1 on page 276)).
- A common mistake is to call `yield` from within a critical region. Doing so is usually pointless because the call to `yield` will look for another thread that can be run but, when that thread is run, it will most likely try to enter the critical region that is held by the yielding thread and go to sleep again. At best, this achieves nothing and, at worst, it causes many additional context switches for no gain.

  If you call `yield`, do so only in circumstances where there is at least a fair chance that another thread will actually be able to run and do something useful.

31.8.5 A Small Example

Following is a small example that uses the `Queue` class we defined in Section 31.6. We create five writer and five reader threads. The writer threads each deposit 100 numbers into the queue, and the reader threads each retrieve 100 numbers and print them to `stdout`:

```
#include <vector>
#include <IceUtil/Thread.h>
// ...

Queue q;

class ReaderThread : public IceUtil::Thread {
    virtual void run() {
        for (int i = 0; i < 100; ++i)
            cout << q.get() << endl;
    }
};

class WriterThread : public IceUtil::Thread {
    virtual void run() {
```

```
        for (int i = 0; i < 100; ++i)
            q.put(i);
    }
};

int
main()
{
    vector<IceUtil::ThreadControl> threads;
    int i;

    // Create five reader threads and start them
    //
    for (i = 0; i < 5; ++i) {
        IceUtil::ThreadPtr t = new ReaderThread;
        threads.push_back(t->start());
    }

    // Create five writer threads and start them
    //
    for (i = 0; i < 5; ++i) {
        IceUtil::ThreadPtr t = new WriterThread;
        threads.push_back(t->start());
    }

    // Wait for all threads to finish
    //
    for (vector<IceUtil::ThreadControl>::iterator i
            = threads.begin(); i != threads.end(); ++i) {
        i->join();
    }
}
```

The code uses the `threads` variable, of type `vector<IceUtil::ThreadControl>` to keep track of the created threads. The code creates five reader and five writer threads, storing the `ThreadControl` object for each thread in the `threads` vector. Once all the threads are created and running, the code joins with each thread before returning from `main`.

Note that you *must not* leave `main` without first joining with the threads you have created: many threading libraries crash if you return from `main` with other threads still running. (This is also the reason why you must not terminate a program without first calling `Communicator::destroy` (see page 274); the `destroy` implementation joins with all outstanding threads before it returns.)

31.9 Priority Inversion

In real-time systems, if you have threads with different priorities, it is possible to encounter a priority inversion. A priority inversion occurs when a low-priority thread prevents a higher-priority thread from running. This situation arises when a low-priority thread acquires a mutex and is pre-empted by one or more medium-priority threads that do not relinquish the CPU. If a high-priority thread then attempts to acquire the mutex locked by the low-priority thread, it will wait (potentially forever) for the medium-priority threads to complete.

One way to deal with this problem is to use a priority inheritance protocol. If thread holds a mutex and a high-priority thread attempts to acquire the mutex, the priority of the thread holding the mutex is temporarily raised to the level of the thread waiting for the mutex. This allows the low-priority thread to keep running until it releases the mutex; as soon as it does, its priority is reduced back to its previous level and the high-priority thread acquires the mutex.

Ice supports the priority inheritance protocol on many POSIX platforms. (Windows does not provide such a protocol.)

For POSIX platforms that support the priority inheritance protocol, mutexes by default do not use it. You can use the `getDefaultMutexProtocol` function to retrieve the current default for your platform:

```
namespace IceUtil {
    enum MutexProtocol { PrioInherit, PrioNone };

    MutexProtocol getDefaultMutexProtocol();

}
```

On Posix systems that do not support priority inheritance and on Windows, this function always returns `PrioNone`.

The return value of `getDefaultMutexProtocol` determines whether a default-constructed `Mutex` or `RecMutex` uses priority inheritance. By default, this function returns `PrioNone`. You can override this default by explicitly specifying a different protocol when you construct a `Mutex` or `RecMutex` (see Sections 31.4 and 31.5). On Windows, if you specify `PrioInherit` when you construct a mutex, the setting is ignored and the mutex is constructed as if you had specified `PrioNone`.

To change the value returned by `getDefaultMutexProtocol`, you can edit `IceUtil/Config.h` and modify the value of the preprocessor macro

ICE_DEFAULT_MUTEX_PROTOCOL (or set the macro's value as a compiler option), and then rebuild the IceUtil library.

31.10 Portable Signal Handling

The IceUtil::CtrlCHandler class provides a portable mechanism to handle Ctrl+C and similar signals sent to a C++ process. On Windows, IceUtil::CtrlCHandler is a wrapper for SetConsoleCtrlHandler; on POSIX platforms, it handles SIGHUP, SIGTERM and SIGINT with a dedicated thread that waits for these signals using sigwait. Signals are handled by a callback function implemented and registered by the user. The callback is a simple function that takes an int (the signal number) and returns void; it should not throw any exception:

```
namespace IceUtil {

    typedef void (*CtrlCHandlerCallback)(int);

    class CtrlCHandler {
    public:
        CtrlCHandler(CtrlCHandlerCallback = 0);
        ~CtrlCHandler();

        void setCallback(CtrlCHandlerCallback);
        CtrlCHandlerCallback getCallback() const;
    };
}
```

The member functions of CtrlCHandler behave as follows:

- constructor

  Constructs an instance with a callback function. Only one instance of CtrlCHandler can exist in a process at a given moment in time. On POSIX platforms, the constructor masks SIGHUP, SIGTERM and SIGINT, then starts a thread that waits for these signals using sigwait. For signal masking to work properly, it is imperative that the CtrlCHandler instance be created before starting any thread, and in particular before initializing an Ice communicator.

- destructor

  Destroys the instance, after which the default signal processing behavior is restored on Windows (`TerminateProcess`). On POSIX platforms, the "sigwait" thread is cancelled and joined, but the signal mask remains unchanged, so subsequent signals are ignored.
- `setCallback`

  Sets a new callback function.
- `getCallback`

  Gets the current callback function.

It is legal specify a value of zero (0) for the callback function, in which case signals are caught and ignored until a non-zero callback function is set.

A typical use for `CtrlCHandler` is to shutdown a communicator in an Ice server (see Section 8.3.1).

31.11 Summary

This chapter explained the threading abstractions provided by Ice: mutexes, monitors, and threads. Using these APIs allows to make your code thread safe and to create threads of your own without having to use non-portable APIs that differ in syntax or semantics across different platforms: Ice not only provides a portable API but also guarantees that the semantics of the various functions are the same across different platforms. This makes it easier to create thread-safe applications and allows you to move your code between platforms with simple recompilation.

Chapter 32
The Ice Run Time in Detail

32.1 Introduction

Now that we have seen the basics of implementing clients and servers, it is time to look at the Ice run time in more detail. This chapter presents the server-side APIs of the Ice run time for synchronous, oneway, and datagram invocations in detail.

Section 32.2 describes the functionality associated with Ice communicators, which are the main handle to the Ice run time. Sections 32.4 to 32.6 describe object adapters and the role they play for call dispatch, and show the relationship between proxies, Ice objects, servants, and object identities. Sections 32.7 and 32.8 describe servant locators and default servants, which are major mechanisms in Ice for controlling the trade-off between performance and memory consumption. Section 32.9 describes the most common implementation techniques that are used by servers. We suggest that you read this section in detail because knowledge of these techniques is crucial to building systems that perform and scale well. Section 32.12 describes implicit transmission of parameters from client to server and Section 32.13 discusses connection timeouts. Sections 32.14 to 32.16 describe oneway, datagram, and batched invocations, and Sections 32.17 to 32.21 deal with location services, administration, logging, statistics collection, and location transparency. Finally, Sections 32.22 to 32.25 discuss automatic retries, dispatch interceptors, string conversion, and how to write an Ice plug-in.

32.2 Communicators

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. An instance of `Ice::Communicator` is associated with a number of run-time resources:

- Client-side thread pool

  The client-side thread pool (see Section 32.10) is used to process replies to asynchronous method invocations (AMI), to avoid deadlocks in callbacks, and to process incoming requests on bidirectional connections (see Section 36.7).

- Server-side thread pool

  Threads in this pool accept incoming connections and handle requests from clients. See Section 32.10 for more information.

- Configuration properties

  Various aspects of the Ice run time can be configured via properties. Each communicator has its own set of such configuration properties (see Chapter 30).

- Object factories

  In order to instantiate classes that are derived from a known base type, the communicator maintains a set of object factories that can instantiate the class on behalf of the Ice run time (see Section 6.14.5 and Section 10.14.4).

- Logger object

  A logger object implements the `Ice::Logger` interface and determines how log messages that are produced by the Ice run time are handled (see Section 32.19).

- Statistics object

  A statistics object implements the `Ice::Stats` interface and is informed about the amount of traffic (bytes sent and received) that is handled by a communicator (see Section 32.20).

- Default router

  A router implements the `Ice::Router` interface. Routers are used by Glacier2 (see Chapter 42) to implement the firewall functionality of Ice.

- Default locator

  A locator is an object that resolves an object identity to a proxy. Locator objects are used to build location services, such as IceGrid (see Chapter 38).

- Plug-in manager

  Plug-ins are objects that add features to a communicator. For example, IceSSL (see Chapter 41) is implemented as a plug-in. Each communicator has a plug-in manager that implements the `Ice::PluginManager` interface and provides access to the set of plug-ins for a communicator.

- Object adapters

  Object adapters dispatch incoming requests and take care of passing each request to the correct servant.

Object adapters and objects that use different communicators are completely independent from each other. Specifically:

- Each communicator uses its own thread pool. This means that if, for example, one communicator runs out of threads for incoming requests, only objects using that communicator are affected. Objects using other communicators have their own thread pool and are therefore unaffected.
- Collocated invocations across different communicators are not optimized, whereas collocated invocations using the same communicator bypass much of the overhead of call dispatch.

Typically, servers use only a single communicator but, occasionally, multiple communicators can be useful. For example, IceBox (see Chapter 43) uses a separate communicator for each Ice service it loads to ensure that different services cannot interfere with each other. Multiple communicators are also useful to avoid thread starvation: if one service runs out of threads, this leaves the remaining services unaffected.

The interface of the communicator is defined in Slice. Part of this interface looks as follows:

```
module Ice {
    local interface Communicator {
        string proxyToString(Object* obj);
        Object* stringToProxy(string str);
        PropertyDict proxyToProperty(Object* proxy,
                                     string property);
        Object* propertyToProxy(string property);
        Identity stringToIdentity(string str);
        string identityToString(Identity id);
        ObjectAdapter createObjectAdapter(string name);
        ObjectAdapter createObjectAdapterWithEndpoints(
                                          string name,
                                          string endpoints);
        ObjectAdapter createObjectAdapterWithRouter(
```

```
                                                string name,
                                                Router* rtr);
        void shutdown();
        void waitForShutdown();
        bool isShutdown();
        void destroy();
        // ...
    };
    // ...
};
```

The communicator offers a number of operations:

- `proxyToString`
  `stringToProxy`

  These operations allow you to convert a proxy into its stringified representation and vice versa.

  Instead of calling `proxyToString` on the communicator, you can also use the `ice_toString` operation on a proxy to stringify it (see Section 32.11.2). However, you can only stringify non-null proxies that way—to stringify a null proxy, you must use `proxyToString`. (The stringified representation of a null proxy is the empty string.)

- `proxyToProperty`
  `propertyToProxy`

  `proxyToProperty` returns the set of proxy properties for the supplied proxy. The `property` parameter specifies the base name for the properties in the returned set (see Section 32.11.1).

  `propertyToProxy` retrieves the configuration property with the given name and converts its value into a proxy (see Section 32.11.1). A null proxy is returned if no property is found with the specified name.

- `identityToString`
  `stringToIdentity`

  These operations allow you to convert an identity to a string and vice versa (see Section 32.5).

- `createObjectAdapter`
  `createObjectAdapterWithEndpoints`
  `createObjectAdapterWithRouter`

  These operations create a new object adapter. Each object adapter is associated with zero or more transport endpoints.

  Typically, an object adapter has a single transport endpoint. However, an object adapter can also offer multiple endpoints. If so, these endpoints each lead to the same set of objects and represent alternative means of accessing these objects. This is useful, for example, if a server is behind a firewall but must offer access to its objects to both internal and external clients; by binding the adapter to both the internal and external interfaces, the objects implemented in the server can be accessed via either interface.

  An object adapter also can have no endpoint at all. In that case, the adapter can only be reached via collocated invocations originating from the same communicator as is used by the adapter.

  Whereas `createObjectAdapter` determines its endpoints from configuration information (see Section 32.4.6), `createObjectAdapterWithEndpoints` allows you to specify the transport endpoints for the new adapter. Typically, you should use `createObjectAdapter` in preference to `createObjectAdapterWithEndpoints`. Doing so keeps transport-specific information, such as host names and port numbers, out of the source code and allows you to reconfigure the application by changing a property (and so avoid recompilation when a transport endpoint needs to be changed).

  `createObjectAdapterWithRouter` creates a routed object adapter that allows clients to receive callbacks from servers that are behind a router (see Chapter 42).

  The newly-created adapter uses its name as a prefix for a collection of configuration properties that tailor the adapter's behavior (see Section D.4). By default, the adapter prints a warning if other properties are defined having the same prefix, but you can disable this warning using the property `Ice.Warn.UnknownProperties` (see Section D.3).

- `shutdown`

  This operation shuts down the server side of the Ice run time:

  - Operation invocations that are in progress at the time `shutdown` is called are allowed to complete normally. `shutdown` does *not* wait for these operations to complete; when shutdown returns, you know that no new incoming requests will be dispatched, but operations that were already in progress at

the time you called `shutdown` may still be running. You can wait for still executing operations to complete by calling `waitForShutdown`.

- Operation invocations that arrive after the server has called `shutdown` either fail with a `ConnectFailedException` or are transparently redirected to a new instance of the server (see Chapter 38).
- Note that `shutdown` initiates deactivation of all object adapters associated with the communicator, so attempts to use an adapter once shutdown has completed raise an `ObjectAdapterDeactivatedException`.

- `waitForShutdown`

  On the server side, this operation suspends the calling thread until the communicator has shut down (that is, until no more operations are executing in the server). This allows you to wait until the server is idle before you destroy the communicator.

  On the client side, `waitForShutdown` simply waits until another thread has called `shutdown` or `destroy`.

- `isShutdown`

  This operation returns true if `shutdown` has been invoked on the communicator. A return value of true does not necessarily indicate that the shutdown process has completed, only that it has been initiated. An application that needs to know whether shutdown is complete can call `waitForShutdown`. If the blocking nature of `waitForShutdown` is undesirable, the application can invoke it from a separate thread.

- `destroy`

  This operation destroys the communicator and all its associated resources, such as threads, communication endpoints, object adapters, and memory resources. Once you have destroyed the communicator (and therefore destroyed the run time for that communicator), you must not call any other Ice operation (other than to create another communicator).

  It is imperative that you call `destroy` before you leave the `main` function of your program. Failure to do so results in undefined behavior.

  Calling `destroy` before leaving `main` is necessary because `destroy` waits for all running threads to terminate before it returns. If you leave `main` without calling `destroy`, you will leave `main` with other threads still running; many threading packages do not allow you to do this and end up crashing your program.

If you call `destroy` without calling `shutdown`, the call waits for all executing operation invocations to complete before it returns (that is, the implementation of `destroy` implicitly calls `shutdown` followed by `waitForShutdown`). `shutdown` (and, therefore, `destroy`) deactivates all object adapters that are associated with the communicator. Since `destroy` blocks until all operation invocations complete, a servant will deadlock if it invokes `destroy` on its own communicator while executing a dispatched operation.

On the client side, calling `destroy` while operations are still executing causes those operations to terminate with a `CommunicatorDestroyedException`.

32.3 Communicator Initialization

During the creation of a communicator, the Ice run time initializes a number of features that affect the communicator's operation. Once set, these features remain in effect for the life time of the communicator, that is, you cannot change these features after you have created a communicator. Therefore, if you want to customize these features, you must do so when you create the communicator.

The following features can be customized at communicator creation time:

- the property set (see Chapter 30)
- the `Logger` interface (see Section 32.19)
- the `Stats` interface (see Section 32.20)
- the narrow and wide string converters (C++ only, see page 1105)
- the thread notification hook (see page 1015)
- the dispatcher (see Section 32.10.7)
- the class loader (Java only, see Section 32.26)

To establish these features, you initialize a structure or class of type `InitializationData` with the relevant settings. For C++ the structure is defined as follows:

```
namespace Ice {
    struct InitializationData {
        PropertiesPtr properties;
        LoggerPtr logger;
        StatsPtr stats;
        StringConverterPtr stringConverter;
        WstringConverterPtr wstringConverter;
```

```
        ThreadNotificationPtr threadHook;
        DispatcherPtr dispatcher;
    };
}
```

For languages other than C++, `InitializationData` is a class with all data members public. (The data members supported by this class vary with each language mapping.)

For C++, `Ice::initialize` is overloaded as follows:

```
namespace Ice {
    CommunicatorPtr initialize(int&, char*[],
                const InitializationData& = InitializationData(),
                Int = ICE_INT_VERSION);
    CommunicatorPtr initialize(StringSeq&,
                const InitializationData& = InitializationData(),
                Int = ICE_INT_VERSION);
    CommunicatorPtr initialize(
                const InitializationData& = InitializationData()
                Int = ICE_INT_VERSION);
}
```

The versions of `initialize` that accept an argument vector look for Ice-specific command-line options and removes them from the argument vector, as described on page 274. The version without an `argc`/`argv` pair is useful if you want to prevent property settings for a program from being changed by command-line arguments (see Section 30.9).

To set a feature, you set the corresponding field in the `InitializationData` structure and pass the structure to `initialize`. For example, to establish a custom logger of type `MyLogger`, you can use:

```
Ice::InitializationData id;
id.logger = new MyLoggerI;
Ice::CommunicatorPtr ic = Ice::initialize(argc, argv, id);
```

For Java, C#, and Objective-C, `Ice.Util.initialize` is overloaded similarly[1] (as is `Ice.initialize` for Python, `Ice::initialize` for Ruby, and `Ice_initialize` for PHP), so you can pass an `InitializationData` instance either with or without an argument vector. Note that you must supply an argument vector if you want `initialize` to look for a configuration file in the **`ICE_CONFIG`** environment variable. (See also Section 30.7.)

1. For Objective-C, the method name is `createCommunicator` (see Section 20.3).

32.4 Object Adapters

A communicator contains one or more object adapters. An object adapter sits at the boundary between the Ice run time and the server application code and has a number of responsibilities:

- It maps Ice objects to servants for incoming requests and dispatches the requests to the application code in each servant (that is, an object adapter implements an up-call interface that connects the Ice run time and the application code in the server).
- It assists in life cycle operations so Ice objects and servants can be created and existing destroyed without race conditions.
- It provides one or more transport endpoints. Clients access the Ice objects provided by the adapter via those endpoints. (It is also possible to create an object adapter without endpoints. In this case the adapter is used for bidirectional callbacks—see Section 36.7.)

Each object adapter has one or more servants that incarnate Ice objects, as well as one or more transport endpoints. If an object adapter has more than one endpoint, all servants registered with that adapter respond to incoming requests on any of the endpoints. In other words, if an object adapter has multiple transport endpoints, those endpoints represent alternative communication paths to the same set of objects (for example, via different transports).

Each object adapter belongs to exactly one communicator (but a single communicator can have many object adapters). Each object adapter has a name that distinguishes it from all other object adapters in the same communicator.

Each object adapter can optionally have its own thread pool, enabled via the `<adapter-name>.ThreadPool.Size` property (see Section 32.10.3). If so, client invocations for that adapter are dispatched in a thread taken from the adapter's thread pool instead of using a thread from the communicator's server thread pool.

32.4.1 The Active Servant Map

Each object adapter maintains a data structure known as the *active servant map*. The active servant map (or *ASM*, for short) is a lookup table that maps object identities to servants: for C++, the lookup value is a smart pointer to the corresponding servant's location in memory; for Java and C#, the lookup value is a reference to the servant. When a client sends an operation invocation to the server, the request

is targeted at a specific transport endpoint. Implicitly, the transport endpoint identifies the object adapter that is the target of the request (because no two object adapters can be bound to the same endpoint). The proxy via which the client sends its request contains the object identity for the corresponding object, and the client-side run time sends this object identity over the wire with the invocation. In turn, the object adapter uses that object identity to look in its ASM for the correct servant to dispatch the call to, as shown in Figure 32.1.

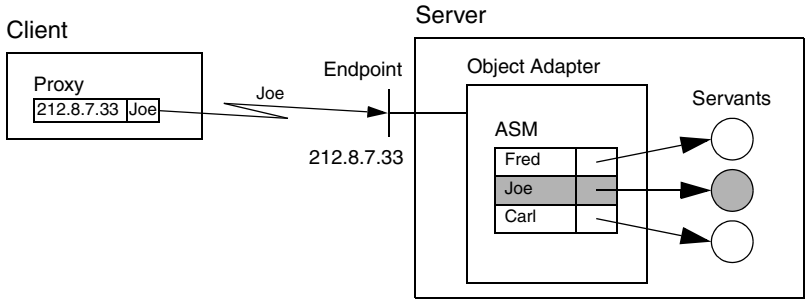


Figure 32.1. Binding a request to the correct servant.

The process of associating a request via a proxy to the correct servant is known as *binding*. The scenario depicted in Figure 32.1 shows direct binding, in which the transport endpoint is embedded in the proxy. Ice also supports an indirect binding mode, in which the correct transport endpoints are provided by the IceGrid service (see Chapter 38 for details).

If a client request contains an object identity for which there is no entry in the adapter's ASM, the adapter returns an `ObjectNotExistException` to the client (unless you use a servant locator—see Section 32.7).

32.4.2 Servants

As mentioned in Section 2.2.2, servants are the physical manifestation of an Ice object, that is, they are entities that are implemented in a concrete programming language and instantiated in the server's address space. Servants provide the server-side behavior for operation invocations sent by clients.

The same servant can be registered with one or more object adapters.

32.4.3 Object Adapter Interface

Object adapters are local interfaces:

```
module Ice {
    local interface ObjectAdapter {
        string getName();
        Communicator getCommunicator();

        // ...
    };
};
```

The operations behave as follows:

- The `getName` operation returns the name of the adapter as passed to one of the communicator operations `createObjectAdapter`, `createObjectAdapterWithEndpoints`, or `createObjectAdapterWith-Router`.
- The `getCommunicator` operation returns the communicator that was used to create the adapter.

Note that there are other operations in the `ObjectAdapter` interface; we will explore these throughout the remainder of this chapter.

32.4.4 Servant Activation and Deactivation

The term *servant activation* refers to making the presence of a servant for a particular Ice object known to the Ice run time. Activating a servant adds an entry to the active servant map shown in Figure 32.1. Another way of looking at servant activation is to think of it as creating a link between the identity of an Ice object and the corresponding programming-language servant that handles requests for that Ice object. Once the Ice run time has knowledge of this link, it can dispatch incoming requests to the correct servant. Without this link, that is, without a corresponding entry in the ASM, an incoming request for the identity results in an `ObjectNotExistException`. While a servant is activated, it is said to *incarnate* the corresponding Ice object.

The inverse operation is known as *servant deactivation*. Deactivating a servant removes an entry for a particular identity from the ASM. Thereafter, incoming requests for that identity are no longer dispatched to the servant and result in an `ObjectNotExistException`.

The object adapter offers a number of operations to manage servant activation and deactivation:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        Object* add(Object servant, Identity id);
        Object* addWithUUID(Object servant);
        Object  remove(Identity id);
        Object  find(Identity id);
        Object  findByProxy(Object* proxy);

        // ...
    };
};
```

The operations behave as follows:

- `add`

  The `add` operation adds a servant with the given identity to the ASM. Requests are dispatched to that servant as soon as `add` is called. The return value is the proxy for the Ice object incarnated by that servant. The proxy embeds the identity passed to `add`.

  You cannot call `add` with the same identity more than once: attempts to add an already existing identity to the ASM result in an `AlreadyRegisteredException`. (It does not make sense to add two servants with the same identity because that would make it ambiguous as to which servant should handle incoming requests for that identity.)

  Note that it is possible to activate the same servant multiple times with different identities. In that case, the same single servant incarnates multiple Ice objects. We explore the ramifications of this in more detail in Section 32.9.2.

- `addWithUUID`

  The `addWithUUID` operation behaves the same way as the `add` operation but does not require you to supply an identity for the servant. Instead, `addWithUUID` generates a UUID (see [14]) as the identity for the corresponding Ice object. You can retrieve the generated identity by calling the `ice_getIdentity` operation on the returned proxy. `addWithUUID` is useful to create identities for temporary objects, such as short-lived session objects. (You can also use `addWithUUID` for persistent objects that do not have a natural identity, as we have done for the file system application.)

- `remove`

  The `remove` operation breaks the association between an identity and its servant by removing the corresponding entry from the ASM; it returns a smart pointer to the removed servant.

  Once the servant is deactivated, new incoming requests for the removed identity result in an `ObjectNotExistException`. Requests that are executing inside the servant at the time `remove` is called are allowed to complete normally. Once the last request for the servant is complete, the object adapter drops its reference (or smart pointer, for C++) to the servant. At that point, the servant becomes available for garbage collection (or is destroyed, for C++), provided that you do not hold references or smart pointers to the servant elsewhere. The net effect is that a deactivated servant is destroyed once it becomes idle.

  Deactivating an object adapter (see Section 32.4.5) implicitly calls `remove` on its active servants.

- `find`

  The `find` operation performs a lookup in the ASM and returns the servant for the specified object identity. If no servant with that identity is registered, the operation returns null. Note that `find` does not consult any servant locators.

- `findByProxy`

  The `findByProxy` operation performs a lookup in the ASM and returns the servant with the object identity and facet that are embedded in the proxy. If no such servant is registered, the operation returns null. Note that `findByProxy` does not consult any servant locators.

32.4.5 Adapter States

An object adapter has a number of processing states:

- holding

  In this state, any incoming requests for the adapter are held, that is, not dispatched to servants.

  For TCP/IP (and other stream-oriented protocols), the server-side run time stops reading from the corresponding transport endpoint while the adapter is in the holding state. In addition, it also does not accept incoming connection requests from clients. This means that if a client sends a request to an adapter that is in the holding state, the client eventually receives a `TimeoutException`

or `ConnectTimeoutException` (unless the adapter is placed into the active state before the timer expires).

For UDP, client requests that arrive at an adapter that is in the holding state are thrown away.

Immediately after creation of an adapter, the adapter is in the holding state. This means that requests are not dispatched until you place the adapter into the active state.

Note that bidirectional adapters (see Section 36.7) cannot be placed into the holding state. If you call `hold` on a bidirectional adapter, the call does nothing.

- active

  In this state, the adapter accepts incoming requests and dispatches them to servants. A newly-created adapter is initially in the holding state. The adapter begins dispatching requests as soon as you place it into the active state.

  You can transition between the active and the holding state as many times as you wish.

  Note that bidirectional adapters (see Section 36.7) need not be activated. Further, calls to collocated servants (that is, to servants that are activated in the communicator that created the proxy) succeed even if the adapter is not activated, unless you have disabled collocation optimization (see Section 32.21).

- inactive

  In this state, the adapter has conceptually been destroyed (or is in the process of being destroyed). Deactivating an adapter destroys all transport endpoints that are associated with the adapter. Requests that are executing at the time the adapter is placed into the inactive state are allowed to complete, but no new requests are accepted. (New requests are rejected with an exception). Any attempt to use a deactivated object adapter results in an `ObjectAdapterDeactivatedException`.

The `ObjectAdapter` interface offers operations that allow you to change the adapter state, as well as to wait for a state change to be complete:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        void activate();
        void hold();
        void waitForHold();
        void deactivate();
```

```
        void waitForDeactivate();
        void isDeactivated();
        void destroy();

        // ...
    };
};
```

The operations behave as follows:

- `activate`

  The `activate` operation places the adapter into the active state. Activating an adapter that is already active has no effect. The Ice run time starts dispatching requests to servants for the adapter as soon as `activate` is called.

- `hold`

  The `hold` operation places the adapter into the holding state. Requests that arrive after calling `hold` are held as detailed on page 931. Requests that are in progress at the time `hold` is called are allowed to complete normally. Note that `hold` returns immediately without waiting for currently executing requests to complete.

- `waitForHold`

  The `waitForHold` operation suspends the calling thread until the adapter has completed its transition to the holding state, that is, until all currently executing requests have finished. You can call `waitForHold` from multiple threads, and you can call `waitForHold` while the adapter is in the active state. If you call `waitForHold` on an adapter that is already in the holding state, `waitForHold` returns immediately.

- `deactivate`

  The `deactivate` operation initiates deactivation of the adapter: requests that arrive after calling `deactivate` are rejected, but currently executing requests are allowed to complete. Once all requests have completed, the adapter enters the inactivate state. Note that `deactivate` returns immediately without waiting for the currently executing requests to complete. A deactivated adapter cannot be reactivated; you can create a new adapter with the same name, but only after calling `destroy` on the existing adapter. Any attempt to use a deactivated object adapter results in an `ObjectAdapterDeactivatedException`.

- `waitForDeactivate`

  The `waitForDeactivate` operation suspends the calling thread until the adapter has completed its transition to the inactive state, that is, until all currently executing requests have completed. You can call `waitForDeactivate` from multiple threads, and you can call `waitForDeactivate` while the adapter is in the active or holding state. Calling `waitForDeactivate` on an adapter that is in the inactive state does nothing and returns immediately.

- `isDeactivated`

  The `isDeactivated` operation returns true if `deactivate` has been invoked on the adapter. A return value of true does not necessarily indicate that the adapter has fully transitioned to the inactive state, only that it has begun this transition. Applications that need to know when deactivation is completed can use `waitForDeactivate`.

- `destroy`

  The `destroy` operation deactivates the adapter and releases all of its resources. Internally, `destroy` invokes `deactivate` followed by `waitForDeactivate`, therefore the operation blocks until all currently executing requests have completed. Furthermore, any servants associated with the adapter are destroyed, all transport endpoints are closed, and the adapter's name becomes available for reuse.

  Destroying a communicator implicitly destroys all of its object adapters. Invoking `destroy` on an adapter is only necessary when you need to ensure that its resources are released prior to the destruction of its communicator.

Placing an adapter into the holding state is useful, for example, if you need to make state changes in the server that require the server (or a group of servants) to be idle. For example, you could place the implementation of your servants into a dynamic library and upgrade the implementation by loading a newer version of the library at run time without having to shut down the server.

Similarly, waiting for an adapter to complete its transition to the inactive state is useful if your server needs to perform some final clean-up work that cannot be carried out until all executing requests have completed.

Note that you can create an object adapter with the same name as a previous object adapter, but only once `destroy` on the previous adapter has completed.

32.4.6 Endpoints

An object adapter maintains two sets of transport endpoints. One set identifies the network interfaces on which the adapter listens for new connections, and the other set is embedded in proxies created by the adapter and used by clients to communicate with it. We will refer to these sets of endpoints as the *physical endpoints* and the *published endpoints*, respectively. In most cases these sets are identical, but there are situations when they must be configured independently.

Physical Endpoints

An object adapter's physical endpoints identify the network interfaces on which it receives requests from clients. These endpoints are configured via the `name.Endpoints` property, or they can be specified explicitly when the adapter is created using the operation `createObjectAdapterWithEndpoints` (see Section 32.2). The endpoint syntax is described in detail in Appendix E; however, an endpoint generally consists of a transport protocol followed by an optional host name and port.

If a host name is specified, the object adapter listens only on the network interface associated with that host name. If no host name is specified but the property `Ice.Default.Host` is defined, the object adapter uses the property's value as the host name. Finally, if a host name is not specified, and the property `Ice.Default.Host` is undefined, the object adapter listens on all available network interfaces, including the loopback interface. You may also force the object adapter to listen on all interfaces by using one of the host names `0.0.0.0` or `*`. The adapter does *not* expand the list of interfaces when it is initialized. Instead, if no host is specified, or you use `-h *` or `- 0.0.0.0`, the adapter binds to `INADDR_ANY` to listen for incoming requests.

If you want an adapter to accept requests on certain network interfaces, you must specify a separate endpoint for each interface. For example, the following property configures a single endpoint for the adapter named `MyAdapter`:

```
MyAdapter.Endpoints=tcp -h 10.0.1.1 -p 9999
```

This endpoint causes the adapter to accept requests on the network interface associated with the IP address `10.0.1.1` at port `9999`. Note however that this adapter configuration does not accept requests on the loopback interface (the one associated with address `127.0.0.1`). If both addresses must be supported, then both must be specified explicitly, as shown below:

```
MyAdapter.Endpoints=\
    tcp -h 10.0.1.1 -p 9999:tcp -h 127.0.0.1 -p 9999
```

If these are the only two network interfaces available on the host, then a simpler configuration omits the host name altogether, causing the object adapter to listen on both interfaces automatically:

```
MyAdapter.Endpoints=tcp -p 9999
```

If you want to make your configuration more explicit, you can use one of the special host names:

```
MyAdapter.Endpoints=tcp -h * -p 9999
```

Another advantage to this configuration is that it ensures the object adapter always listens on all interfaces, even if a definition for `Ice.Default.Host` is later added to your configuration. Without an explicit host name, the addition of `Ice.Default.Host` could potentially change the interfaces on which the adapter is listening. For diagnostic purposes, you can determine the set of local addresses that Ice substitutes for the wildcard address by setting the property `Ice.Trace.Network=3` and reviewing the log output.

Careful consideration must also be given to the selection of a port for an endpoint. If no port is specified, the adapter uses a port that is selected (essentially at random) by the operating system, meaning the adapter will likely be using a different port each time the server is restarted. Whether that behavior is desirable depends on the application, but in many applications a client has a proxy containing the adapter's endpoint and expects that proxy to remain valid indefinitely. Therefore, an endpoint generally should contain a fixed port to ensure that the adapter is always listening at the same port.

However, there are certain situations where a fixed port is not required. For example, an adapter whose servants are transient does not need a fixed port, because the proxies for those objects are not expected to remain valid past the lifetime of the server process. Similarly, a server using indirect binding via IceGrid (see Chapter 38) does not need a fixed port because its port is never published.

Published Endpoints

An object adapter publishes its endpoints in the proxies it creates, but it is not always appropriate to publish the adapter's physical endpoints in a proxy. For example, suppose a server is running on a host in a private network, protected from the public network by a firewall that can forward network traffic to the server. The adapter's physical endpoints must use the private network's address scheme, but a client in the public network would be unable to use those endpoints if they were published in a proxy. In this scenario, the adapter must publish endpoints in its proxies that direct the client to the firewall instead.

The published endpoints are configured using the adapter property *name*.PublishedEndpoints. If this property is not defined, the adapter publishes its physical endpoints by default, with one exception: endpoints for the loopback address (127.0.0.1) are not published unless the loopback interface is the only interface, or 127.0.0.1 (or loopback) is explicitly listed as an endpoint with the **-h** option. Otherwise, to force the inclusion of loopback endpoints when they would normally be excluded, you must define *name*.PublishedEndpoints explicitly.

As an example, the properties below configure the adapter named MyAdapter with physical and published endpoints:

```
MyAdapter.Endpoints=tcp -h 10.0.1.1 -p 9999
MyAdapter.PublishedEndpoints=tcp -h corpfw -p 25000
```

This example assumes that clients connecting to host corpfw at port 25000 are forwarded to the adapter's endpoint in the private network.

Another use case of published endpoints is for replicated servers. Suppose we have two instances of a stateless server running on separate hosts in order to distribute the load between them. We can supply the client with a bootstrap proxy containing the endpoints of both servers, and the Ice run time in the client will select one of the servers at random when a connection is established. However, should the client invoke an operation on a server that returns a proxy for another object, that proxy would normally contain only the endpoint of the server that created it. Invocations on the new proxy are always directed at the same server, reducing the opportunity for load balancing.

We can alleviate this situation by configuring the adapters to publish the endpoints of both servers. For example, here is a configuration for the server on host Sun1:

```
MyAdapter.Endpoints=tcp -h Sun1 -p 9999
MyAdapter.PublishedEndpoints=tcp -h Sun1 -p 9999:tcp -h Sun2 -p 9999
```

Similarly, the configuration for host Sun2 retains the same published endpoints:

```
MyAdapter.Endpoints=tcp -h Sun2 -p 9999
MyAdapter.PublishedEndpoints=tcp -h Sun1 -p 9999:tcp -h Sun2 -p 9999
```

For troubleshooting purposes, you can examine the published endpoints for an object adapter by setting the property Ice.Trace.Network=3. Note however that this setting generates significant trace information about the Ice run time's network activity, therefore you may not want to use this setting by default.

Refreshing Endpoints

The list of interfaces of a host may change over time, for example, if a laptop moves in and out of range of a wireless network. The object adapter provides an operation to refresh its list of interfaces:

```
local interface ObjectAdapter {
    void refreshPublishedEndpoints();
    // ...
};
```

Calling `refreshPublishedEndpoints` causes the object adapter to update its internal list of available network interfaces and to use the most recent setting for the *`name`*`.PublishedEndpoints` property. This allows you to react to changing network interfaces while an object adapter is in use, but your application code must determine when it is necessary to call this operation.

Note that `refreshPublishedEndpoints` takes effect only for object adapters that specify published endpoints without a host or set the published endpoints to `-h *` or `-h 0.0.0.0`.

Timeouts

As a defense against hostile clients, we recommend that you specify a timeout for you physical object adapter endpoints. The timeout value you select affects tasks that the Ice run time normally does not expect to block for any significant amount of time, such as writing a reply message to a socket or waiting for SSL negotiation to complete. If you do not specify a timeout, the Ice run time waits indefinitely in these situations. As a result, malicious or misbehaving clients could consume excessive resources such as file descriptors.

Specifying a timeout in an object adapter endpoint is done exactly as in a proxy endpoint using the `-t` option:

```
MyAdapter.Endpoints=tcp -p 9999 -t 5000
```

In this example, we specify a timeout of five seconds.

Routers

If an object adapter is configured with a router, the adapter's published endpoints are augmented to reflect the router. See Chapter 42 for more information on configuring an adapter with a router.

Obtaining Endpoint Information

The object adapter provides two operations to retrieve the physical and published endpoints:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        EndpointSeq getEndpoints();
        EndpointSeq getPublishedEndpoints();

        // ...
    };
};
```

The sequences that are returned contain the adapter's physical and published endpoints, respectively. Section 36.5.2 provides details about the information provided by the sequence elements.

32.4.7 Creating Proxies

Although the servant activation operations described in Section 32.4.4 return proxies, the life cycle of proxies is completely independent from servants (see Chapter 34). The `ObjectAdapter` interface provides several operations for creating a proxy, regardless of whether a servant is currently activated for the object's identity:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        Object* createProxy(Identity id);
        Object* createDirectProxy(Identity id);
        Object* createIndirectProxy(Identity id);

        // ...
    };
};
```

These operations are described below:

- `createProxy`

  The `createProxy` operation returns a new proxy for the object with the given identity. The adapter's configuration determines whether the return value is a

direct proxy or an indirect proxy (see Section 2.2.2). If the adapter is configured with an adapter id (see Section 32.17.5), the operation returns an indirect proxy that refers to the adapter id. If the adapter is also configured with a replica group id, the operation returns an indirect proxy that refers to the replica group id. Otherwise, if an adapter id is not defined, `createProxy` returns a direct proxy containing the adapter's published endpoints (see Section 32.4.6).

- `createDirectProxy`

  The `createDirectProxy` operation returns a direct proxy containing the adapter's published endpoints (see Section 32.4.6).

- `createIndirectProxy`

  The `createIndirectProxy` operation returns an indirect proxy. If the adapter is configured with an adapter id, the returned proxy refers to that adapter id. Otherwise, the proxy refers only to the object's identity (see page 11).

  In contrast to `createProxy`, `createIndirectProxy` does not use the replica group id. Therefore, the returned proxy always refers to a specific replica.

Configuring Proxies

After using one of the operations discussed above to create a proxy, you will receive a proxy that is configured by default for twoway invocations. If you require the proxy to have a different configuration, you can use the proxy factory methods described in Section 32.11.2. As an example, the code below demonstrates how to configure the proxy for oneway invocations:

```
// C++
Ice::ObjectAdapterPtr adapter = ...;
Ice::Identity id = ...;
Ice::ObjectPrx proxy = adapter->createProxy(id)->ice_oneway();
```

You can also instruct the object adapter to use a different default proxy configuration by setting the property *name*`.ProxyOptions`. For example, the following property causes the object adapter to return proxies that are configured for oneway invocations by default:

```
MyAdapter.ProxyOptions=-o
```

See Appendix D for more information on this property.

32.4.8 Using Multiple Object Adapters

A typical server rarely needs to use more than one object adapter. If you are considering using multiple object adapters, we suggest that you check whether any of the items in the list below apply to your situation:

- You need fine-grained control over which objects are accessible. For example, you could have an object adapter with only secure endpoints to restrict access to some administrative objects, and another object adapter with non-secure endpoints for other objects. Because an object adapter is associated with one or more transport endpoints, you can firewall a particular port, so objects associated with the corresponding endpoint cannot be reached unless the firewall rules are satisfied.
- You need control over the number of threads in the pools for different sets of objects in your application. For example, you may not need concurrency on the objects connected to a particular object adapter, and multiple object adapters, each with its own thread pool, can be useful to solve deadlocks. See Section 32.10.5 for more information on dealing with deadlocks.
- You want to be able to temporarily disable processing new requests for a set of objects. This can be accomplished by placing an object adapter in the holding state.
- You want to set up different request routing when using an Ice router with Glacier2.

If none of the preceding items apply, chances are that you do not need more than one object adapter.

32.5 Object Identity

Each Ice object has an object identity defined as follows:

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
};
```

As you can see, an object identity consists of a pair of strings, a `name` and a `category`. The complete object identity is the combination of `name` and `category`, that

is, for two identities to be equal, both `name` and `category` must be the same. The `category` member is usually the empty string, unless you are using servant locators (see Section 32.7).[2]

If `name` is an empty string, `category` must be the empty string as well. (An identity with an empty `name` and a non-empty `category` is illegal.) If a proxy contains an identity in which `name` is empty, Ice interprets that proxy as a null proxy.

Object identities can be represented as strings; the `category` part appears first and is followed by the `name`; the two components are separated by a `/` character, for example:

```
Factory/File
```

In this example, `Factory` is the `category`, and `File` is the `name`. If the `name` or `category` member themselves contain a `/` character, the stringified representation escapes the `/` character with a `\`, for example:

```
Factories\/Factory/Node\/File
```

In this example, the `category` member is `Factories/Factory` and the `name` member is `Node/File`.

32.5.1 Syntax for Stringified Identities

You rarely need to write identities as strings because, typically, your code will be using `identityToString` and `stringToIdentity` (see Section 32.5.2), or simply deal with proxies instead of identities. However, on occasion, you will need to use stringified identities in configuration files. If the identities happen to contain meta-characters (such as a slash or backslash), or characters outside the printable ASCII range, these characters must be escaped in the stringified representation. Here are rules that the Ice run time applies when parsing a stringified identity:

1. The parser scans the stringified identity for an un-escaped slash character (`/`). If such a slash character can be found, the substrings to the left and right of the slash are parsed as the `category` and `name` members of the identity, respectively; if no such slash character can be found, the entire string is parsed as the `name` member of the identity, and the `category` member is the empty string.

2. Glacier2 (see Chapter 42) also uses the `category` member for filtering.

2. Each of the `category` (if present) and `name` substrings is parsed according to the following rules:
   1. All characters in the string must be in the ASCII range 32 (space) to 126 (~); characters outside this range cause the parse to fail.
   2. Any character that is not part of an escape sequence is treated as that character.
   3. The parser recognizes the following escape sequences and replaces them with their equivalent character:

      `\\` (backslash)

      `\'` (single quote)

      `\"` (double quote)

      `\b` (space)

      `\f` (form feed)

      `\n` (new line)

      `\r` (carriage return)

      `\t` (tab)
   4. An escape sequence of the form `\o`, `\oo`, or `\ooo` (where *o* is a digit in the range 0 to 7) is replaced with the ASCII character with the corresponding octal value. Parsing for octal digits allows for at most three consecutive digits, so the string `\0763` is interpreted as the character with octal value 76 (>) followed by the character `3`. Parsing for octal digits terminates as soon as it encounters a character that is not in the range 0 to 7, so `\7x` is the character with octal value 7 (bell) followed by the character `x`. Octal escape sequences must be in the range 0 to 255 (octal 000 to 377); escape sequences outside this range cause a parsing error. For example, `\539` is an illegal escape sequence.
   5. If a character follows a backslash, but is not part of a recognized escape sequence, the backslash is ignored, so `\x` is the character `x`.

32.5.2 Helper Functions

To make conversion of identities to and from strings easier, the `Communicator` interface provides appropriate conversion functions:

```
local interface Communicator {
    string identityToString(Identity id);
    Identity stringToIdentity(string id);
};
```

For C++, the operations on the communicator are the only way to convert between identities and strings. For other languages, the conversion functions are provided as operations on the communicator as well but, in addition, the language mappings provide static utility functions. (The utility functions have the advantage that you can call them without holding a reference to the communicator.)[3]

For Java, the utility functions are in the `Ice.Util` class and are defined as:

```
package Ice;

public final class Util {
    public static String   identityToString(Identity id);
    public static Identity stringToIdentity(String s);
}
```

For C#, the utility functions are in the `Ice.Util` class and are defined as:

```
namespace Ice
{
    public sealed class Util
    {
        public static string   identityToString(Identity id);
        public static Identity stringToIdentity(string s);
    }
}
```

These functions correctly encode and decode characters that might otherwise cause problems (such as control characters).

As mentioned on page 930, each entry in the ASM for an object adapter must be unique: you cannot add two servants with the same identity to the ASM.

3. For C++, the static utility functions are not provided due to the need to apply string conversions, and the string converters are registered on the communicator (see Section 32.24).

32.6 The Ice::Current Object

Up to now, we have tacitly ignored the trailing parameter of type Ice::Current that is passed to each skeleton operation on the server side. The Current object is defined as follows:

```
module Ice {
    local dictionary<string, string> Context;

    enum OperationMode { Normal, \Nonmutating, \Idempotent };

    local struct Current {
        ObjectAdapter   adapter;
        Connection      con;
        Identity        id;
        string          facet;
        string          operation;
        OperationMode   mode;
        Context         ctx;
        int             requestId;
    };
};
```

Note that the Current object provides access to information about the currently executing request to the implementation of an operation in the server:

- adapter

  The adapter member provides access to the object adapter via which the request is being dispatched. In turn, the adapter provides access to its communicator (via the getCommunicator operation).

- con

  The con member provides information about the connection over which this request was received (see Section 36.5.1).

- id

  The id member provides the object identity for the current request (see Section 32.5).

- facet

  The facet member provides access to the facet for the request (see Chapter 33).

- `operation`

  The `operation` member contains the name of the operation that is being invoked. Note that the operation name may indicate one of the operations on `Ice::Object`, such as `ice_ping` or `ice_isA`. (`ice_isA` is invoked if a client performs a `checkedCast`.)
- `mode`

  The `mode` member contains the invocation mode for the operation (`Normal` or `Idempotent`).
- `ctx`

  The `ctx` member contains the current context for the invocation (see Section 32.12).
- `requestId`

  The Ice protocol (see Chapter 37) uses request IDs to associate replies with their corresponding requests. The `requestId` member provides this ID. For oneway requests (which do not have replies), the request ID is `0`. For collocated requests (which do not use the Ice protocol), the request ID is `-1`.

If you implement your server such that it uses a separate servant for each Ice object, the contents of the `Current` object are not particularly interesting. (You would most likely access the `Current` object to read the `adapter` member, for example, to activate or deactivate a servant.) However, as we will see in Section 32.7, the `Current` object is essential for more sophisticated (and more scalable) servant implementations.

32.7 Servant Locators

Using an adapter's ASM to map Ice objects to servants has a number of design implications:

- Each Ice object is represented by a different servant.[4]
- All servants for all Ice objects are permanently in memory.

Using a separate servant for each Ice object in this fashion is common to many server implementations: the technique is simple to implement and provides a

4. It is possible to register a single servant with multiple identities. However, there is little point in doing so because a default servant (see Section 32.8) achieves the same thing.

natural mapping from Ice objects to servants. Typically, on start-up, the server instantiates a separate servant for each Ice object, activates each servant, and then calls `activate` on the object adapter to start the flow of requests.

There is nothing wrong with the above design, provided that two criteria are met:

1. The server has sufficient memory available to keep a separate servant instantiated for each Ice object at all times.
2. The time required to initialize all the servants on start-up is acceptable.

For many servers, neither criterion presents a problem: provided that the number of servants is small enough and that the servants can be initialized quickly, this is a perfectly acceptable design. However, the design does not scale well: the memory requirements of the server grow linearly with the number of Ice objects so, if the number of objects gets too large (or if each servant stores too much state), the server runs out of memory.

Ice offers two APIs that help you scale servers to larger numbers of objects: *servant locators* and *default servants*. A default servant is essentially a simplified version of a servant locator that satisfies the majority of use cases, whereas a servant locator provides more flexibility for those applications that require it. Refer to Section 32.8 for more information on default servants.

32.7.1 Overview

In a nutshell, a servant locator is a local object that you implement and attach to an object adapter. Once an adapter has a servant locator, it consults its ASM to locate a servant for an incoming request as usual. If a servant for the request can be found in the ASM, the request is dispatched to that servant. However, if the ASM does not have an entry for the object identity of the request, the object adapter calls back into the servant locator to ask it to provide a servant for the request. The servant locator either

- instantiates a servant and passes it to the Ice run time, in which case the request is dispatched to that newly instantiated servant, or
- the servant locator indicates failure to locate a servant to the Ice run time, in which case the client receives an `ObjectNotExistException`.

This simple mechanism allows us to scale servers to provide access to an unlimited number of Ice objects: instead of instantiating a separate servant for each and every Ice object in existence, the server can instantiate servants for only a subset of Ice objects, namely those that are actually used by clients.

Servant locators are most commonly used by servers that provide access to databases: typically, the number of entries in the database is far larger than what the server can hold in memory. Servant locators allow the server to only instantiate servants for those Ice objects that are actually used by clients.

Another common use for servant locators is in servers that are used for process control or network management: in that case, there is no database but, instead, there is a potentially very large number of devices or network elements that must be controlled via the server. Otherwise, this scenario is the same as for large databases: the number of Ice objects exceeds the number of servants that the server can hold in memory and, therefore, requires an approach that allows the number of instantiated servants to be less than the number of Ice objects.

32.7.2 Servant Locator Interface

A servant locator has the following interface:

```
module Ice {
    local interface ServantLocator {
        ["UserException"]
        Object locate(    Current     curr,
                      out LocalObject cookie);

        ["UserException"]
        void finished(    Current     curr,
                          Object      servant,
                          LocalObject cookie);

        void deactivate(string category);
    };
};
```

Note that `ServantLocator` is a local interface. To create an actual implementation of a servant locator, you must define a class that is derived from `Ice::ServantLocator` and provide implementations of the `locate`, `finished`, and `deactivate` operations. The Ice run time invokes the operations on your derived class as follows:

- `locate`

  Whenever a request arrives for which no entry exists in the ASM, the Ice run time calls `locate`. The implementation of `locate` (which you provide as part of the derived class) is supposed to return a servant that can process the

incoming request. Your implementation of `locate` can behave in three possible ways:

1. Instantiate and return a servant for the current request. In this case, the Ice run time dispatches the request to the newly instantiated servant.
2. Return null. In this case, the Ice run time raises an `ObjectNotExistException` in the client.
3. Throw a run-time exception. In this case, the Ice run time propagates the thrown exception back to the client. Keep in mind that all run-time exceptions, apart from `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`, are presented as `UnknownLocalException` to the client.

   You can also throw user exceptions from `locate`. If the user exception is in the corresponding operation's exception specification, that user exception is returned to the client. User exceptions thrown by `locate` that are not listed in the exception specification of the corresponding operation are returned to the client as `UnknownUserException`. Non-Ice exceptions are returned to the client as `UnknownException` (see page 119).

The `cookie` out-parameter to `locate` allows you return a local object to the object adapter. The object adapter does not care about the contents of that object (and it is legal to return a null cookie). Instead, the Ice run time passes whatever cookie you return from `locate` back to you when it calls `finished`. This allows you to pass an arbitrary amount of state from `locate` to the corresponding call to `finished`.

- `finished`

  If a call to `locate` has returned a servant to the Ice run time, the Ice run time dispatches the incoming request to the servant. Once the request is complete (that is, the operation being invoked has completed), the Ice run time calls `finished`, passing the servant whose operation has completed, the `Current` object for the request, and the cookie that was initially created by `locate`. This means that every call to `locate` is balanced by a corresponding call to `finished` (provided that `locate` actually returned a servant).

  If you throw an exception from `finished`, the Ice run time propagates the thrown exception back to the client. As for `locate`, you can throw user exceptions from `finished`. If a user exception is in the corresponding operation's exception specification, that user exception is returned to the client. User

exceptions that are not in the corresponding operation's exception specification are returned to the client as `UnknownUserException`.

`finished` can also throw run-time exceptions. However, only `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException` are propagated without change to the client; other run-time exceptions are returned to the client as `UnknownLocalException`.

Non-Ice exceptions thrown from `finished` are returned to the client as `UnknownException` (see page 119).

If both the operation implementation and `finished` throw a user exception, the exception thrown by `finished` overrides the exception thrown by the operation.

- `deactivate`

  The `deactivate` operation allows a servant locator to clean up once it is no longer needed. (For example, the locator might close a database connection.) The Ice run time passes the category of the servant locator being deactivated to the `deactivate` operation.

  The run time calls `deactivate` when the object adapter to which the servant locator is attached is destroyed. More precisely, `deactivate` is called when you call `destroy` on the object adapter, or when you call `destroy` on the communicator (which implicitly calls `destroy` on the object adapter).

  Once the run time has called `deactivate`, it is guaranteed that no further calls to `locate` or `finished` can happen, that is, `deactivate` is called exactly once, after all operations dispatched via this servant locator have completed.

  This also explains why `deactivate` is not called as part of `ObjectAdapter::deactivate`: `ObjectAdapter::deactivate` initiates deactivation and returns immediately, so it cannot call `ServantLocator::deactivate` directly, because there might still be outstanding requests dispatched via this servant locator that have to complete first—in turn, this would mean that either `ObjectAdapter::deactivate` could block (which it must not do) or that a call to `ServantLocator::deactivate` could be followed by one or more calls to `finished` (which must not happen either).

It is important to realize that the Ice run time does not "remember" the servant that is returned by a particular call to `locate`. Instead, the Ice run time simply dispatches an incoming request to the servant returned by `locate` and, once the request is complete, calls `finished`. In particular, if two requests for the same servant arrive more or less simultaneously, the Ice run time calls `locate` and `finished` once for each request. In other words, `locate` establishes the associa-

tion between an object identity and a servant; that association is valid only for a single request and is never used by the Ice run time to dispatch a different request.

32.7.3 Threading Guarantees for Servant Locators

The Ice run time guarantees that every operation invocation that involves a servant locator is bracketed by calls to `locate` and `finished`, that is, every call to `locate` is balanced by a corresponding call to `finished` (assuming that the call to `locate` actually returned a servant, of course).

In addition, the Ice run time guarantees that `locate`, the operation, and `finished` are called by the same thread. This guarantee is important because it allows you to use `locate` and `finished` to implement thread-specific pre- and post-processing around operation invocations. (For example, you can start a transaction in `locate` and commit or roll back that transaction in `finished`, or you can acquire a lock in `locate` and release the lock in `finished`.[5])

Note that, if you are using asynchronous method dispatch, the thread that starts a call is not necessarily the thread that finishes it. In that case, `finished` is called by whatever thread executes the operation implementation, which may be a different thread than the one that called `locate`.

The Ice run time also guarantees that `deactivate` is called when you deactivate the object adapter to which the servant locator is attached. The `deactivate` call is made only once all operations that involved the servant locator are finished, that is, `deactivate` is guaranteed not to run concurrently with `locate` or `finished`, and is guaranteed to be the *last* call made to a servant locator.

Beyond this, the Ice run time provides no threading guarantees for servant locators. In particular:

- It is possible for invocations of `locate` to proceed concurrently (for the same object identity or for different object identities).
- It is possible for invocations of `finished` to proceed concurrently (for the same object identity or for different object identities).
- It is possible for invocations of `locate` and `finished` to proceed concurrently (for the same object identity or for different object identities).

5. Both transactions and locks usually are thread-specific, that is, only the thread that started a transaction can commit it or roll it back, and only the thread that acquired a lock can release the lock.

These semantics allow you to extract the maximum amount of parallelism from your application code (because the Ice run time does not serialize invocations when serialization may not be necessary). Of course, this means that you must protect access to shared data from `locate` and `finished` with mutual exclusion primitives as necessary.

32.7.4 Servant Locator Registration

An object adapter does not automatically know when you create a servant locator. Instead, you must explicitly register servant locators with the object adapter:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        void addServantLocator(ServantLocator locator,
                               string         category);

        ServantLocator removeServantLocator(string category);

        ServantLocator findServantLocator(string category);

        // ...
    };
};
```

As you can see, the object adapter allows you to add, remove, and find servant locators. Note that, when you register a servant locator, you must provide an argument for the `category` parameter. The value of the `category` parameter controls which object identities the servant locator is responsible for: only object identities with a matching `category` member (see page 941) trigger a corresponding call to `locate`. An incoming request for which no explicit entry exists in the ASM and with a category for which no servant locator is registered returns an `ObjectNotExistException` to the client.

`addServantLocator` has the following semantics:

- You can register exactly one servant locator for a specific category. Attempts to call `addServantLocator` for the same category more than once raise an `AlreadyRegisteredException`.
- You can register different servant locators for different categories, or you can register the same single servant locator multiple times (each time for a different category). In the former case, the category is implicit in the servant

locator instance that is called by the Ice run time; in the latter case, the implementation of `locate` can find out which category the incoming request is for by examining the object identity member of the `Current` object that is passed to `locate`.

- It is legal to register a servant locator for the empty category. Such a servant locator is known as a *default servant locator*: if a request comes in for which no entry exists in the ASM, and whose category does not match the category of any other registered servant locator, the Ice run time calls `locate` on the default servant locator.

`removeServantLocator` removes and returns the servant locator for a specific category (including the empty category) with the following semantics:

- If no servant locator is registered for the specified category, the operation raises `NotRegisteredException`.
- Once a servant locator is successfully removed for the specified category, the Ice run time guarantees that no new incoming requests for that category are dispatched to the servant locator.
- A call to `removeServantLocator` returns immediately without waiting for the completion of any pending requests on that servant locator; such requests still complete normally by calling `finished` on the servant locator.
- Removing a servant locator does not cause Ice to invoke `deactivate` on that servant locator, as `deactivate` is only called when a registered servant locator's object adapter is destroyed.

`findServantLocator` allows you to retrieve the servant locator for a specific category (including the empty category). If no match is found, the operation returns null.

Call Dispatch Semantics

The preceding rules may seem complicated, so here is a summary of the actions taken by the Ice run time to locate a servant for an incoming request.

Every incoming request implicitly identifies a specific object adapter for the request (because the request arrives at a specific transport endpoint and, therefore, identifies a particular object adapter). The incoming request carries an object identity that must be mapped to a servant. To locate a servant, the Ice run time goes through the following steps, in the order shown:

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request to the corresponding servant.

2. If the category of the incoming object identity is non-empty, look for a default servant (see Section 32.8) that is registered for that category. If such a default servant is registered, dispatch the request to that servant.
3. If the category of the incoming object identity is empty, or no default servant could be found for the category in step 2, look for a default servant that is registered for the empty category. If such a default servant is registered, dispatch the request to that servant.
4. If the category of the incoming object identity is non-empty and no servant could be found in the preceding steps, look for a servant locator that is registered for that category. If such a servant locator is registered, call `locate` on the servant locator and, if `locate` returns a servant, dispatch the request to that servant, followed by a call to `finished`; otherwise, if the call to `locate` returns null, raise `ObjectNotExistException` or `FacetNotExistException` in the client.
5. If the category of the incoming object identity is empty, or no servant locator could be found for the category in step 4, look for a default servant locator (that is, a servant locator that is registered for the empty category). If a default servant locator is registered, dispatch the request as for step 2.
6. Raise `ObjectNotExistException` or `FacetNotExistException` in the client. (`ObjectNotExistException` is raised if the ASM does not contain a servant with the given identity at all, `FacetNotExistException` is raised if the ASM contains a servant with a matching identity, but a non-matching facet.)

It is important to keep these call dispatch semantics in mind because they enable a number of powerful implementation techniques. Each technique allows you to streamline your server implementation and to precisely control the trade-off between performance, memory consumption, and scalability. To illustrate the possibilities, we outline a number of the most common implementation techniques in the following section.

32.7.5 Implementing a Simple Servant Locator

To illustrate the concepts outlined in the previous sections, let us examine a (very simple) implementation of a servant locator. Consider that we want to create an electronic phone book for the entire world's telephone system (which, clearly, involves a very large number of entries, certainly too many to hold the entire phone book in memory). The actual phone book entries are kept in a large database. Also assume that we have a search operation that returns the details of a

phone book entry. The Slice definitions for this application might look something like the following:

```
struct Details {
    // Lots of details about the entry here...
};

interface PhoneEntry {
    idempotent Details getDetails();
    idempotent void updateDetails(Details d);
    // ...
};

struct SearchCriteria {
    // Fields to permit searching...
};

interface PhoneBook {
    idempotent PhoneEntry* search(SearchCriteria c);
    // ...
};
```

The details of the application do not really matter here; the important point to note is that each phone book entry is represented as an interface for which we need to create a servant eventually, but we cannot afford to keep servants for all entries permanently in memory.

Each entry in the phone database has a unique identifier. This identifier might be an internal database identifier, or a combination of field values, depending on exactly how the database is constructed. The important point is that we can use this database identifier to link the proxy for an Ice object to its persistent state: we simply use the database identifier as the object identity. This means that each proxy contains the primary access key that is required to locate the persistent state of each Ice object and, therefore, instantiate a servant for that Ice object.

What follows is an outline implementation in C++. The class definition of our servant locator looks as follows:

```
class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current&  c,
                                  Ice::LocalObjectPtr& cookie);

    virtual void finished(const Ice::Current&        c,
                          const Ice::ObjectPtr&      servant,
```

```
                                const Ice::LocalObjectPtr& cookie);

    virtual void deactivate(const std::string& category);
};
```

Note that `MyServantLocator` inherits from `Ice::ServantLocator` and implements the pure virtual functions that are generated by the **`slice2cpp`** compiler for the `Ice::ServantLocator` interface. Of course, as always, you can add additional member functions, such as a constructor and destructor, and you can add private data members as necessary to support your implementation.

In C++, you can implement the `locate` member function along the following lines:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current&  c,
                         Ice::LocalObjectPtr& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try {
        d = DB_lookup(name);
    } catch (const DB_error&)
        return 0;
    }

    // We have the state, instantiate a servant and return it.
    //
    return new PhoneEntryI(d);
}
```

For the time being, the implementations of `finished` and `deactivate` are empty and do nothing.

The `DB_lookup` call in the preceding example is assumed to access the database. If the lookup fails (presumably, because no matching record could be found), `DB_lookup` throws a `DB_error` exception. The code catches that exception and returns zero instead; this raises `ObjectNotExistException` in the client to indicate that the client used a proxy to a no-longer existent Ice object.

Note that `locate` instantiates the servant on the heap and returns it to the Ice run time. This raises the question of when the servant will be destroyed. The answer is that the Ice run time holds onto the servant for as long as necessary, that is, long enough to invoke the operation on the returned servant and to call `finished` once the operation has completed. Thereafter, the servant is no longer needed and the Ice run time destroys the smart pointer that was returned by `locate`. In turn, because no other smart pointers exist for the same servant, this causes the destructor of the `PhoneEntryI` instance to be called, and the servant to be destroyed.

The upshot of this design is that, for every incoming request, we instantiate a servant and allow the Ice run time to destroy the servant once the request is complete. Depending on your application, this may be exactly what is needed, or it may be prohibitively expensive—we will explore designs that avoid creation and destruction of a servant for every request shortly.

In Java, the implementation of our servant locator looks very similar:[6]

```
public class MyServantLocator implements Ice.ServantLocator {

    public Ice.Object locate(Ice.Current c,
                             Ice.LocalObjectHolder cookie)
    {
        // Get the object identity. (We use the name member
        // as the database key.
        String name = c.id.name;

        // Use the identity to retrieve the state
        // from the database.
        //
        ServantDetails d;
        try {
            d = DB.lookup(name);
        } catch (DB.error e) {
            return null;
        }

        // We have the state, instantiate a servant and return it.
        //
        return new PhoneEntryI(d);
```

6. The C# implementation is virtually identical to the Java implementation, so we do not show it here.

```
    }

    public void finished(Ice.Current c,
                         Ice.Object servant,
                         java.lang.Object cookie)
    {
    }

    public void deactivate(String category)
    {
    }
}
```

All implementations of `locate` follow the pattern illustrated by the previous pseudo-code:

1. Use the `id` member of the passed `Current` object to obtain the object identity. Typically, only the `name` member of the identity is used to retrieve servant state. The `category` member is normally used to select a servant locator. (We will explore use of the category member shortly.)
2. Retrieve the state of the Ice object from secondary storage (or the network) using the object identity as a key.
   - If the lookup succeeds, you have retrieved the state of the Ice object.
   - If the lookup fails, return null. In that case, the Ice object for the client's request truly does not exist, presumably, because that Ice object was deleted earlier, but the client still has a proxy to the now-deleted object.
3. Instantiate a servant and use the state retrieved from the database to initialize the servant. (In this example, we pass the retrieved state to the servant constructor.)
4. Return the servant.

Of course, before we can use our servant locator, we must inform the adapter of its existence prior to activating the adapter, for example (in Java or C#):

```
MyServantLocator sl = new MyServantLocator();
adapter.addServantLocator(sl, "");
```

Note that, in this example, we have installed the servant locator for the empty category. This means that `locate` on our servant locator will be called for invocations to any of our Ice objects (because the empty category acts as the default). In effect, with this design, we are not using the `category` member of the object identity. This is fine, as long as all our servants all have the same, single interface.

However, if we need to support several different interfaces in the same server, this simple strategy is no longer sufficient.

32.7.6 Using the `category` Member of the Object Identity

The simple example in the preceding section always instantiates a servant of type `PhoneEntryI`. In other words, the servant locator implicitly is aware of the type of servant the incoming request is for. This is not a very realistic assumption for most servers because, usually, a server provides access to objects with several different interfaces. This poses a problem for our `locate` implementation: somehow, we need to decide inside `locate` what type of servant to instantiate. You have several options for solving this problem:

- Use a separate object adapter for each interface type and use a separate servant locator for each object adapter.

  This technique works fine, but has the down-side that each object adapter requires a separate transport endpoint, which is wasteful.

- Mangle a type identifier into the `name` component of the object identity.

  This technique uses part of the object identity to denote what type of object to instantiate. For example, in our file system application, we have directory and file objects. By convention, we could prepend a '`d`' to the identity of every directory and prepend an '`f`' to the identity of every file. The servant locator then can use the first letter of the identity to decide what type of servant to instantiate:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current&  c,
                         Ice::LocalObjectPtr& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;
    std::string realId = c.id.name.substr(1);
    try {
        if (name[0] == 'd') {
            // The request is for a directory.
            //
            DirectoryDetails d = DB_lookup(realId);
            return new DirectoryI(d);
        } else {
            // The request is for a file.
```

```
            //
            FileDetails d = DB_lookup(realId);
            return new FileI(d);
        }
    } catch (DatabaseNotFoundException&) {
        return 0;
    }
}
```

While this works, it is awkward: not only do we need to parse the `name` member to work out what type of object to instantiate, but we also need to modify the implementation of `locate` whenever we add a new type to our application.

- Use the `category` member of the object identity to denote the type of servant to instantiate.

  This is the recommended approach: for every interface type, we assign a separate identifier as the value of the `category` member of the object identity. (For example, we can use '`d`' for directories and '`f`' for files.) Instead of registering a single servant locator, we create two different servant locator implementations, one for directories and one for files, and then register each locator for the appropriate category:

```
class DirectoryLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current&  c,
                                  Ice::LocalObjectPtr& cookie)
    {
        // Code to locate and instantiate a directory here...
    }


    virtual void finished(const Ice::Current&        c,
                          const Ice::ObjectPtr&      servant,
                          const Ice::LocalObjectPtr& cookie)
    {
    }

    virtual void deactivate(const std::string& category)
    {
    }
};

class FileLocator : public virtual Ice::ServantLocator {
```

```
public:

    virtual Ice::ObjectPtr locate(const Ice::Current&  c,
                                  Ice::LocalObjectPtr& cookie)
    {
        // Code to locate and instantiate a file here...
    }


    virtual void finished(const Ice::Current&        c,
                          const Ice::ObjectPtr&      servant,
                          const Ice::LocalObjectPtr& cookie)
    {
    }

    virtual void deactivate(const std::string& category)
    {
    }
};

// ...

// Register two locators, one for directories and
// one for files.
//
adapter->addServantLocator(new DirectoryLocator(), "d");
adapter->addServantLocator(new FileLocator(), "f");
```

Yet another option is to use the `category` member of the object identity, but to use a single default servant locator (that is, a locator for the empty category). With this approach, all invocations go to the single default servant locator, and you can switch on the `category` value inside the implementation of the `locate` operation to determine which type of servant to instantiate. However, this approach is harder to maintain than the previous one; the `category` member of the Ice object identity exists specifically to support servant locators, so you might as well use it as intended.

32.7.7 Using Cookies

Occasionally, it can be useful to be able to pass information between `locate` and `finished`. For example, the implementation of `locate` could choose among a number of alternative database backends, depending on load or availability and, to properly finalize state, the implementation of `finish` might need to know which

database was used by `locate`. To support such scenarios, you can create a cookie in your `locate` implementation; the Ice run time passes the value of the cookie to `finished` after the operation invocation has completed. The cookie must derive from `Ice::LocalObject` and can contain whatever state and member functions are useful to your implementation:

```
class MyCookie : public virtual Ice::LocalObject {
public:
    // Whatever is useful here...
};

typedef IceUtil::Handle<MyCookie> MyCookiePtr;

class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current&  c,
                                  Ice::LocalObjectPtr& cookie)
    {
        // Code as before...

        // Allocate and initialize a cookie.
        //
        cookie = new MyCookie(...);

        return new PhoneEntryI;
    }

    virtual void finished(const Ice::Current&        c,
                          const Ice::ObjectPtr&      servant,
                          const Ice::LocalObjectPtr& cookie)
    {
        // Down-cast cookie to actual type.
        //
        MyCookiePtr mc = MyCookiePtr::dynamicCast(cookie);

        // Use information in cookie to clean up...
        //
        // ...
    }

    virtual void deactivate(const std::string& category);
};
```

32.8 Default Servants

As explained in Section 32.4.1, the Active Servant Map (ASM) is a simple lookup table that maintains a one-to-one mapping between object identities and servants. Although the ASM is easy to understand and offers efficient indexing, it does not scale well when the number of objects is very large. Scalability is a common problem with object-oriented middleware: servers frequently are used as front ends to large databases that are accessed remotely by clients. The server's job is to present an object-oriented view to clients of a very large number of records in the database. Typically, the number of records is far too large to instantiate servants for even a fraction of the database records.

A common technique for solving this problem is to use *default servants*. A default servant is a servant that, for each request, takes on the persona of a different Ice object. In other words, the servant changes its behavior according to the object identity that is accessed by a request, on a per-request basis. In this way, it is possible to allow clients access to an unlimited number of Ice objects with only a single servant in memory. A default servant is essentially a specialized version of a *servant locator* (see Section 32.7) that satisfies the majority of use cases with a simpler API, whereas a servant locator provides more flexibility for those applications that require it.

Default servant implementations are attractive not only because of the memory savings they offer, but also because of the simplicity of implementation: in essence, a default servant is a facade [2] to the persistent state of an object in the database. This means that the programming required to implement a default servant is typically minimal: it simply consists of the code required to read and write the corresponding database records.

32.8.1 Overview

A default servant is a regular servant that you implement and register with an object adapter. For each incoming request, the object adapter first attempts to locate a servant in its Active Servant Map (ASM). If no servant is found, the object adapter dispatches the request to a default servant. With this design, a default servant is the object adapter's servant of last resort if no match was found in the ASM.

Implementing a default servant requires a somewhat different mindset than the typical "one servant per Ice object" strategy used in less advanced applications. The most important quality of a default servant is its statelessness: it must be

prepared to dispatch multiple requests simultaneously for different objects. The price we have to pay for the unlimited scalability and reduced memory footprint is performance: default servants typically make a database access for every invoked operation, which is obviously slower than caching state in memory as part of a servant that has been added to the ASM. However, this does not mean that default servants carry an unacceptable performance penalty: databases often provide sophisticated caching, so even though the operation implementations read and write the database, as long as they access cached state, performance may be entirely acceptable.

32.8.2 Default Servant API

The default servant API consists of the following operations in the object adapter interface:

```
module Ice {
    local interface ObjectAdapter {
        void addDefaultServant(Object servant, string category);
        Object removeDefaultServant(string category);
        Object findDefaultServant(string category);

        // ...
    };
};
```

As you can see, the object adapter allows you to add, remove, and find default servants. Note that, when you register a default servant, you must provide an argument for the `category` parameter. The value of the `category` parameter controls which object identities the default servant is responsible for: only object identities with a matching `category` member trigger a dispatch to this default servant. An incoming request for which no explicit entry exists in the ASM and with a category for which no default servant is registered returns an `ObjectNotExistException` to the client.

`addDefaultServant` has the following semantics:

- You can register exactly one default servant for a specific category. Attempts to call `addDefaultServant` for the same category more than once raise an `AlreadyRegisteredException`.
- You can register different default servants for different categories, or you can register the same single default servant multiple times (each time for a different category). In the former case, the category is implicit in the default

servant instance that is called by the Ice run time; in the latter case, the servant can find out which category the incoming request is for by examining the object identity member of the `Current` object that is passed to the dispatched operation.

- It is legal to register a default servant for the empty category. Such a servant is used if a request comes in for which no entry exists in the ASM, and whose category does not match the category of any other registered default servant.

`removeDefaultServant` removes the default servant for the specified category. Attempts to remove a non-existent default servant raise `NotRegisteredException`. The operation returns the removed default servant. Once a default servant is successfully removed for the specified category, the Ice run time guarantees that no new incoming requests for that category are dispatched to the servant.

The `findDefaultServant` operation allows you to retrieve the default servant for a specific category (including the empty category). If no default servant is registered for the specified category, `findDefaultServant` returns null.

32.8.3 Threading Guarantees for Default Servants

The threading semantics for a default servant are no different than for a servant registered in the ASM: operations may be dispatched on a default servant concurrently, for the same object identity or for different object identities. If you have configured the communicator with multiple dispatch threads, your default servant must protect access to shared data with appropriate locks.

32.8.4 Call Dispatch Semantics

This section summarizes the actions taken by the Ice run time to locate a servant for an incoming request.

Every incoming request implicitly identifies a specific object adapter for the request (because the request arrives at a specific transport endpoint and, therefore, identifies a particular object adapter). The incoming request carries an object identity that must be mapped to a servant. To locate a servant, the Ice run time goes through the following steps, in the order shown:

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request to the corresponding servant.
2. If the category of the incoming object identity is non-empty, look for a default servant that is registered for that category. If such a default servant is registered, dispatch the request to that servant.

3. If the category of the incoming object identity is empty, or no default servant could be found for the category in step 2, look for a default servant that is registered for the empty category. If such a default servant is registered, dispatch the request to that servant.
4. If the category of the incoming object identity is non-empty and no servant could be found in the preceding steps, look for a servant locator (see Section 32.7) that is registered for that category. If such a servant locator is registered, call `locate` on the servant locator and, if `locate` returns a servant, dispatch the request to that servant, followed by a call to `finished`; otherwise, if the call to `locate` returns null, raise `ObjectNotExistException` or `FacetNotExistException` in the client. (`ObjectNotExistException` is raised if the ASM does not contain a servant with the given identity at all, `FacetNotExistException` is raised if the ASM contains a servant with a matching identity, but a non-matching facet.)
5. If the category of the incoming object identity is empty, or no servant locator could be found for the category in step 4, look for a default servant locator (that is, a servant locator that is registered for the empty category). If a default servant locator is registered, dispatch the request as for step 2.
6. Raise `ObjectNotExistException` or `FacetNotExistException` in the client. (`ObjectNotExistException` is raised if the ASM does not contain a servant with the given identity at all, `FacetNotExistException` is raised if the ASM contains a servant with a matching identity, but a non-matching facet.)

It is important to keep these call dispatch semantics in mind because they enable a number of powerful implementation techniques. Each technique allows you to streamline your server implementation and to precisely control the trade-off between performance, memory consumption, and scalability. We discuss some guidelines for using default servants in the following section.

32.8.5 Guidelines

This section provides some guidelines to assist you in implementing default servants effectively.

Object identity is the Key

When an incoming request is dispatched to the default servant, the target object identity is provided in the `Current` argument. The `name` field of the identity typically supplies everything the default servant requires in order to satisfy the

request. For instance, it may serve as the key in a database query, or even hold an encoded structure in some proprietary format that your application uses to convey more than just a string.

Naturally, the client can also pass arguments to the operation that assist the default servant in retrieving whatever state it requires. However, this approach can easily introduce implementation artifacts into your Slice interfaces, and in most cases the client should not need to know that the server is implemented with a default servant. If at all possible, use only the object identity.

Minimize Contention

For better scalability, the default servant's implementation should strive to eliminate contention among the dispatch threads. As an example, when a database holds the default servant's state, each of the servant's operations usually begins with a query. Assuming that the database API is thread-safe, the servant needs to perform no explicit locking of its own. With a copy of the state in hand, the implementation can work with function-local data to satisfy the request.

Combine Strategies

The ASM still plays a useful role even in applications that are ideally suited for default servants. For example, there is no need to implement a singleton object as a default servant: if there can only be one instance of the object, implementing it as a default servant does nothing to improve your application's scalability.

Applications often install a handful of servants in the ASM while servicing the majority of requests in a default servant. For example, a database application might install a singleton query object in the ASM while using a default servant to process all invocations on the database records.

Categories Denote Interfaces

In general, all of the objects serviced by a default servant must have the same interface. If you only need a default servant for one interface, you can register the default servant with an empty category string. However, to implement several interfaces, you will need a default servant implementation for each one. Furthermore, you must take steps to ensure that the object adapter dispatches an incoming request to the appropriate default servant. The `category` field of the object identity is intended to serve this purpose.

For example, a process control system might have interfaces named `Sensor` and `Switch`. To direct requests to the proper default servant, the application uses the symbol `Sensor` or `Switch` as the category of each object's identity, and

registers corresponding default servants having those same categories with the object adapter.

Plan for the Future

If you suspect that you might eventually need to implement more than one interface with default servants, we recommend using a non-empty category even if you start out having only one default servant. Adding another default servant later becomes much easier if the application is already designed to operate correctly with categories.

Throw exceptions

If a request arrives for an object that no longer exists, it is the default servant's responsibility to raise `ObjectNotExistException`.

Handle `ice_ping`

One issue you need to be aware of with default servants is the need to override `ice_ping`: the default implementation of `ice_ping` that the servant inherits from its skeleton class always succeeds. For servants that are registered with the ASM, this is exactly what we want; however, for default servants, `ice_ping` must fail if a client uses a proxy to a no-longer existent Ice object. To avoid getting successful `ice_ping` invocations for non-existent Ice objects, you must override `ice_ping` in the default servant. The implementation must check whether the object identity for the request denotes a still-existing Ice object and, if not, raise `ObjectNotExistException`.

It is good practice to override `ice_ping` if you are using default servants. Because you cannot override operations on `Ice::Object` using a Java or C# tie servant (or an Objective-C delegate servant), you must implement default servants by deriving from the generated skeleton class if you choose to override `ice_ping`.

Consider Interceptors

Dispatch interceptors are an advanced Ice feature that you can read more about in Section 32.23. A dispatch interceptor is often installed as a default servant.

Use a `Blobject` Default Servant to Forward Messages

Message forwarding services, such as Glacier2 (see Chapter 42), can be implemented simply and efficiently with a `Blobject` default servant (see

Section 35.3.2). Such a servant simply chooses a destination to forward a request to, without decoding any of the parameters.

32.9 Server Implementation Techniques

As we mentioned on page 947, instantiating a servant for each Ice object on server start-up is a viable design, provided that you can afford the amount of memory required by the servants, as well as the delay in start-up of the server. However, Ice supports more flexible mappings between Ice objects and servants; these alternate mappings allow you to precisely control the trade-off between memory consumption, scalability, and performance. We outline a few of the more common implementation techniques in this section.

32.9.1 Incremental Initialization

If you use a servant locator, the servant returned by `locate` is used only for the current request, that is, the Ice run time does not add the servant to the active servant map. Of course, this means that if another request comes in for the same Ice object, `locate` must again retrieve the object state and instantiate a servant. A common implementation technique is to add each servant to the ASM as part of `locate`. This means that only the first request for each Ice object triggers a call to `locate`; thereafter, the servant for the corresponding Ice object can be found in the ASM and the Ice run time can immediately dispatch another incoming request for the same Ice object without having to call the servant locator.

An implementation of `locate` to do this would look something like the following:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current&  c,
                         Ice::LocalObjectPtr& cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try {
```

```
        d = DB_lookup(name);
    } catch (const DB_error&)
        return 0;
    }

    // We have the state, instantiate a servant.
    //
    Ice::ObjectPtr servant = new PhoneEntryI(d);

    // Add the servant to the ASM.
    //
    c.adapter->add(servant, c.id);      // NOTE: Incorrect!

    return servant;
}
```

This is almost identical to the implementation on page 956—the only difference is that we also add the servant to the ASM by calling `ObjectAdapter::add`. Unfortunately, this implementation is wrong because it suffers from a race condition. Consider the situation where we do not have a servant for a particular Ice object in the ASM, and two clients more or less simultaneously send a request for the same Ice object. It is entirely possible for the thread scheduler to schedule the two incoming requests such that the Ice run time completes the lookup in the ASM for both requests and, for each request, concludes that no servant is in memory. The net effect is that `locate` will be called twice for the same Ice object, and our servant locator will instantiate two servants instead of a single servant. Because the second call to `ObjectAdapter::add` will raise an `AlreadyRegisteredException`, only one of the two servants will be added to the ASM.

Of course, this is hardly the behavior we expect. To avoid the race condition, our implementation of `locate` must check whether a concurrent invocation has already instantiated a servant for the incoming request and, if so, return that servant instead of instantiating a new one. The Ice run time provides the `ObjectAdapter::find` operation to allow us to test whether an entry for a specific identity already exists in the ASM:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        Object find(Identity id);
```

```
        // ...
    };
};
```

find returns the servant if it exists in the ASM and null, otherwise. Using this lookup function, together with a mutex, allows us to correctly implement locate. The class definition of our servant locator now has a private mutex so we can establish a critical region inside locate:

```
class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current&  c,
                                  Ice::LocalObjectPtr&);

    // Declaration of finished() and deactivate() here...

private:
    IceUtil::Mutex _m;
};
```

The locate member function locks the mutex and tests whether a servant is already in the ASM: if so, it returns that servant; otherwise, it instantiates a new servant and adds it to the ASM as before:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current&  c,
                         Ice::LocalObjectPtr&)
{
    IceUtil::Mutex::Lock lock(_m);

    // Check if we have instantiated a servant already.
    //
    Ice::ObjectPtr servant = c.adapter.find(c.id);

    if (!servant) {     // We don't have a servant already

        // Instantiate a servant.
        //
        ServantDetails d;
        try {
           d = DB_lookup(c.id.name);
        } catch (const DB_error&) {
           return 0;
        }
```

```
        servant = new PhoneEntryI(d);

        // Add the servant to the ASM.
        //
        c.adapter->add(servant, c.id);
    }

    return servant;
}
```

The Java version of this locator is almost identical, but we use the `synchronized` qualifier instead of a mutex to make `locate` a critical region:[7]

```
synchronized public Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    // Check if we have instantiated a servant already.
    //
    Ice.Object servant = c.adapter.find(c.id);

    if (servant == null) { // We don't have a servant already

        // Instantiate a servant
        //
        ServantDetails d;
        try {
            d = DB.lookup(c.id.name);
        } catch (DB.error&) {
            return null;
        }
        servant = new PhoneEntryI(d);

        // Add the servant to the ASM.
        //
        c.adapter.add(servant, c.id);
    }

    return servant;
}
```

Using a servant locator that adds the servant to the ASM has a number of advantages:

7. In C#, you can place the body of locate into a `lock(this)` statement.

- Servants are instantiated on demand, so the cost of initializing the servants is spread out over many invocations instead of being incurred all at once during server start-up.
- The memory requirements for the server are reduced because servants are instantiated only for those Ice objects that are actually accessed by clients. If clients only access a subset of the total number of Ice objects, the memory savings can be substantial.

In general, incremental initialization is beneficial if instantiating servants during start-up is too slow. The memory savings can be worthwhile as well but, as a rule, are realized only for comparatively short-lived servers: for long-running servers, chances are that, sooner or later, every Ice object will be accessed by some client or another; in that case, there are no memory savings because we end up with an instantiated servant for every Ice object regardless.

32.9.2 Default Servants

As explained in Section 32.8, default servants are a very effective tool for conserving memory when a server hosts a large number of Ice objects.

To create a default servant implementation, we need as many default servants as there are non-abstract interfaces in the system. For example, for our file system application, we require two default servants, one for directories and one for files. In addition, the object identities we create use the `category` member of the object identity to encode the type of interface of the corresponding Ice object. The value of the category field can be anything that identifies the interface, such as the '`d`' and '`f`' convention we used on page 960. Alternatively, you could use `"Directory"` and `"File"`, or use the type ID of the corresponding interface, such as `"::Filesystem::Directory"` and `"::Filesystem::File"`. The `name` member of the object identity must be set to whatever identifier we can use to retrieve the persistent state of each directory and file from secondary storage. (For our file system application, we used a UUID as a unique identifier.)

Registration of the default servants is as follows:

```
adapter->addDefaultServant(new DirectoryI, "d");
adapter->addDefaultServant(new FileI, "f");
```

All the action happens in the implementation of the operations, using the following steps for each operation:

1. Use the passed `Current` object to get the identity for the current request.

2. Use the `name` member of the identity to locate the persistent state of the servant on secondary storage. If no record can be found for the identity, throw an `ObjectNotExistException`.
3. Implement the operation to operate on that retrieved state (returning the state or updating the state as appropriate for the operation).

In pseudo-code, this might look something like the following:

```
Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current& c) const
{
    // Use the identity of the directory to retrieve
    // its contents.
    DirectoryContents dc;
    try {
        dc = DB_getDirectory(c.id.name);
    } catch(const DB_error&) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }

    // Use the records retrieved from the database to
    // initialize return value.
    //
    FileSystem::NodeSeq ns;
    // ...

    return ns;
}
```

Note that the servant implementation is completely stateless: the only state it operates on is the identity of the Ice object for the current request (and that identity is passed as part of the `Current` parameter).

Overriding `ice_ping`

Section 32.8.5 recommends that a default servant implementation take steps to preserve the semantics of the `ice_ping` operation, which is used to test whether an Ice object exists. If a default servant fails to override `ice_ping`, clients may mistakenly believe that a non-existent Ice object still exists. The code below demonstrates how we can override the operation in our file system application:

```
void
Filesystem::DirectoryI::ice_ping(const Ice::Current& c) const
{
    try {
```

```
        d = DB_lookup(c.id.name);
    } catch (const DB_error&) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
}
```

It is good practice to override `ice_ping` if you are using default servants.

32.9.3 Hybrid Approaches and Caching

Depending on the nature of your application, you may be able to steer a middle path that provides better performance while keeping memory requirements low: if your application has a number of frequently-accessed objects that are performance-critical, you can add servants for those objects to the ASM. If you store the state of these objects in data members inside the servants, you effectively have a cache of these objects.

The remaining, less-frequently accessed objects can be implemented with a default servant. For example, in our file system implementation, we could choose to instantiate directory servants permanently, but to have file objects implemented with a default servant. This provides efficient navigation through the directory tree and incurs slower performance only for the (presumably less frequent) file accesses.

This technique could be augmented with a cache of recently-accessed files, along similar lines to the buffer pool used by the Unix kernel [10]. The point is that you can combine use of the ASM with servant locators and default servants to precisely control the trade-offs among scalability, memory consumption, and performance to suit the needs of your application.

32.9.4 Servant Evictors

A variation on the previous theme and particularly interesting use of a servant locator is as an *evictor* [4]. An evictor is a servant locator that maintains a cache of servants:

- Whenever a request arrives (that is, `locate` is called by the Ice run time), the evictor checks to see whether it can find a servant for the request in its cache. If so, it returns the servant that is already instantiated in the cache; otherwise, it instantiates a servant and adds it to the cache.
- The cache is a queue that is maintained in least-recently used (LRU) order: the least-recently used servant is at the tail of the queue, and the most-recently

used servant is at the head of the queue. Whenever a servant is returned from or added to the cache, it is moved from its current queue position to the head of the queue, that is, the "newest" servant is always at the head, and the "oldest" servant is always at the tail.

- The queue has a configurable length that corresponds to how many servants will be held in the cache; if a request arrives for an Ice object that does not have a servant in memory and the cache is full, the evictor removes the least-recently used servant at the tail of the queue from the cache in order to make room for the servant about to be instantiated at the head of the queue.

Figure 32.2 illustrates an evictor with a cache size of five after five invocations have been made, for object identities 1 to 5, in that order.

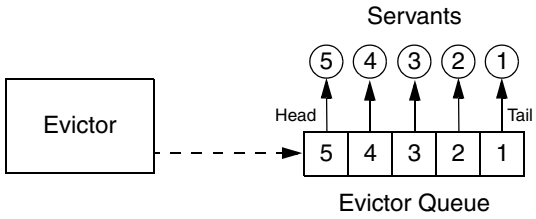


Figure 32.2. An evictor after five invocations for object identities 1 to 5.

At this point, the evictor has instantiated five servants, and has placed each servant onto the evictor queue. Because requests were sent by the client for object identities 1 to 5 (in that order), servant 5 ends up at the head of the queue (at the most-recently used position), and servant 1 ends up at the tail of the queue (at the least-recently used position).

Assume that the client now sends a request for servant 3. In this case, the servant is found on the evictor queue and moved to the head position. The resulting ordering is shown in Figure 32.3.

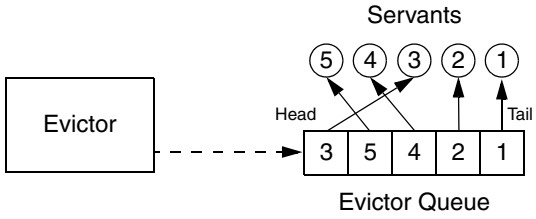


Figure 32.3. The evictor from Figure 32.2 after accessing servant 3.

Assume that the next client request is for object identity 6. The evictor queue is fully populated, so the evictor creates a servant for object identity 6, places that servant at the head of the queue, and evicts the servant with identity 1 (the least-recently used servant) at the tail of the queue, as shown in Figure 32.4.

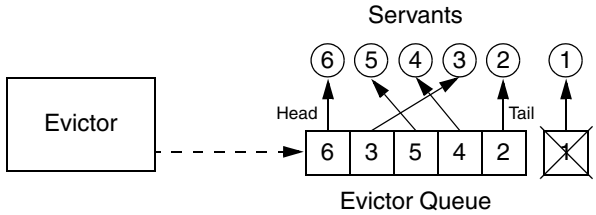


Figure 32.4. The evictor from Figure 32.3 after evicting servant 1.

The evictor pattern combines the advantages of the ASM with the advantages of a default servant: provided that the cache size is sufficient to hold the working set of servants in memory, most requests are served by an already instantiated servant, without incurring the overhead of creating a servant and accessing the database to initialize servant state. By setting the cache size, you can control the trade-off between performance and memory consumption as appropriate for your application.

The following sections show how to implement an evictor in both C++ and Java. (You can also find the source code for the evictor with the code examples for this book in the Ice distribution.)

Creating an Evictor Implementation in C++

The evictor we show here is designed as an abstract base class: in order to use it, you derive an object from the `EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definitions as follows:

```
class EvictorBase : public Ice::ServantLocator {
public:
    EvictorBase(int size = 1000);

    virtual Ice::ObjectPtr locate(const Ice::Current& c,
                                  Ice::LocalObjectPtr& cookie);
    virtual void finished(const Ice::Current& c,
                          const Ice::ObjectPtr&,
                          const Ice::LocalObjectPtr& cookie);
    virtual void deactivate(const std::string&);
```

```
protected:
    virtual Ice::ObjectPtr add(const Ice::Current&,
                               Ice::LocalObjectPtr&) = 0;
    virtual void evict(const Ice::ObjectPtr&,
                       const Ice::LocalObjectPtr&) = 0;

private:
    // ...
};

typedef IceUtil::Handle<EvictorBase> EvictorBasePtr;
```

Note that the evictor has a constructor that sets the size of the queue, with a default argument to set the size to 1000.

The `locate`, `finished`, and `deactivate` functions are inherited from the `ServantLocator` base class; these functions implement the logic to maintain the queue in LRU order and to add and evict servants as needed.

The `add` and `evict` functions are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are pure virtual, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant. This allows `evict` to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map does not only store servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue. Storing the queue position is not strictly necessary—we store the position for efficiency reasons because it allows us to locate a servant's position in the queue in constant time instead of having to search through the queue in order to maintain its LRU property.

3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The need for the use count deserves some extra explanation: suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still executing, the servant for *I* could be evicted while it is still executing the original request.

By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*. However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

The evictor queue does not store the identity of the servant. Instead, the entries on the queue are iterators into the evictor map. This is useful when the time comes to evict a servant: instead of having to search the map for the identity of the servant to be evicted, we can simply delete the map entry that is pointed at by the iterator at the tail of the queue. We can get away with storing an iterator into the evictor queue as part of the map, and storing an iterator into the evictor map as part of the queue because both `std::list` and `std::map` do not invalidate forward iterators when we add or delete entries[8] (except for invalidating iterators that point at a deleted entry, of course).

8. Reverse iterators *can* be invalidated by modification of list entries: if a reverse iterator points at `rend` and the element at the head of the list is erased, the iterator pointing at `rend` is invalidated.

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

This leads to the following definitions in the private section of our evictor:

```
class EvictorBase : public Ice::ServantLocator {
    // ...

private:

    struct EvictorEntry;
    typedef IceUtil::Handle<EvictorEntry> EvictorEntryPtr;

    typedef std::map<Ice::Identity, EvictorEntryPtr> EvictorMap;
    typedef std::list<EvictorMap::iterator> EvictorQueue;

    struct EvictorEntry : public Ice::LocalObject
    {
        Ice::ObjectPtr servant;
        Ice::LocalObjectPtr userCookie;
        EvictorQueue::iterator queuePos;
        int useCount;
    };

    EvictorMap _map;
    EvictorQueue _queue;
    Ice::Int _size;

    IceUtil::Mutex _mutex;

    void evictServants();
};
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. In addition, we use a private `_mutex` data member so we can correctly serialize access to the evictor's data structures.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit—we will discuss this function in more detail shortly.

The `EvictorEntry` structure serves as the cookie that we pass from `locate` to `finished`; it stores the servant, the servant's position in the evictor queue, the servant's use count, and the cookie that we pass from `add` to `evict`.

The implementation of the constructor is trivial. The only point of note is that we ignore negative sizes:[9]

```
EvictorBase::EvictorBase(Ice::Int size) : _size(size)
{
    if (_size < 0)
        _size = 1000;
}
```

Almost all the action of the evictor takes place in the implementation of `locate`:

```
Ice::ObjectPtr
EvictorBase::locate(const Ice::Current& c,
                    Ice::LocalObjectPtr& cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    //
    // Check if we have a servant in the map already.
    //
    EvictorEntryPtr entry;
    EvictorMap::iterator i = _map.find(c.id);
    if (i != _map.end()) {
        //
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        entry = i->second;
        _queue.erase(entry->queuePos);
    } else {
        //
        // We do not have an entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map.
        //
        entry = new EvictorEntry;
        entry->servant = add(c, entry->userCookie); // Down-call
        if (!entry->servant) {
            return 0;
        }
        entry->useCount = 0;
        i = _map.insert(std::make_pair(c.id, entry)).first;
    }
```

9. We could have stored the size as a `size_t` instead. However, for consistency with the Java implementation, which cannot use unsigned integers, we use `Ice::Int` to store the size.

```
    //
    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(entry->useCount);
    entry->queuePos = _queue.insert(_queue.begin(), i);

    cookie = entry;

    return entry->servant;
}
```

The first step in `locate` is to lock the `_mutex` data member. This protects the evictor's data structures from concurrent access. The next step is to instantiate a smart pointer to an `EvictorEntry`. That smart pointer acts as the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`. That same smart pointer is also the value type of our map entries, so we do not store two copies of the same information redundantly—instead, smart pointers ensure that a single copy of each `EvictorEntry` structure is shared by both the cookie and the map.

The next step is to look in the evictor map to see whether we already have an entry for this object identity. If so, we remove the entry from its current queue position.

Otherwise, we do not have an entry for this object identity yet, so we have to create one. The code creates a new evictor entry, and then calls `add` to get a new servant. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we return zero to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The last few lines of `locate` add the entry for the current request to the head of the evictor queue to maintain its LRU property, increment the use count of the entry, set the cookie that is returned from locate to point at the `EvictorEntry`, and finally return the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

```
void
EvictorBase::finished(const Ice::Current&,
                      const Ice::ObjectPtr&,
                      const Ice::LocalObjectPtr& cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    EvictorCookiePtr ec = EvictorCookiePtr::dynamicCast(cookie);

    // Decrement use count and check if
    // there is something to evict.
    //
    --(ec->entry->useCount);
    evictServants();
}
```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

```
void
EvictorBase::evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    EvictorQueue::reverse_iterator p = _queue.rbegin();
    int excessEntries = static_cast<int>(_map.size() - _size);

    for (int i = 0; i < excessEntries; ++i) {
        EvictorMap::iterator mapPos = *p;
        if (mapPos->second->useCount == 0) {
            evict(mapPos->second->servant,
                  mapPos->second->userCookie);
            p = EvictorQueue::reverse_iterator(
                    _queue.erase(mapPos->second->queuePos));
            _map.erase(mapPos);
        } else
            ++p;
    }
}
```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function

in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

```
void
EvictorBase::deactivate(const std::string& category)
{
    IceUtil::Mutex::Lock lock(_mutex);

    _size = 0;
    evictServants();
}
```

Note that, with this implementation of `evictServants`, we only scan the tail section of the evictor queue for servants to evict. If we have long-running operations, this allows the number of servants in the queue to remain above the evictor size if the servants in the tail section have a non-zero use count. This means that, even immediately after calling `evictServants`, the queue length can still exceed the evictor size.

We can adopt a more aggressive strategy for eviction: instead of scanning only the excess entries in the queue, if, after looking in the tail section of the queue, we still have more servants in the queue than the queue size, we keep scanning for servants with a zero use count until the queue size drops below the limit. This alternative version of `evictServants` looks as follows:

```
void
EvictorBase::evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // try to evict servants until the length drops
    // below the limit.
    //
    EvictorQueue::reverse_iterator p = _queue.rbegin();
    int numEntries = static_cast<int>_map.size();

    for (int i = 0; i < numEntries && _map.size() > _size; ++i) {
        EvictorMap::iterator mapPos = *p;
        if (mapPos->second->useCount == 0) {
            evict(mapPos->second->servant,
```

```
                    mapPos->second->userCookie);
            p = EvictorQueue::reverse_iterator(
                    _queue.erase(mapPos->second->queuePos));
            _map.erase(mapPos);
        } else
            ++p;
    }
}
```

The only difference in this version is that terminating condition for the `for`-loop has changed: instead of scanning only the excess entries for servants with a use count, this version keeps scanning until the evictor size drops below the limit.

Which version is more appropriate depends on your application: if locating and evicting servants is expensive, and memory is not at a premium, the first version (which only scans the tail section) is more appropriate; if you want to keep memory consumption to a minimum, the second version in more appropriate. Also keep in mind that the difference between the two versions is significant only if you have long-running operations and many concurrent invocations from clients; otherwise, there is no point in more aggressively scanning for servants to remove because they are going to become idle again very quickly and get evicted as soon as the next request arrives.

Creating an Evictor Implementation in Java

The evictor we show here is designed as an abstract base class: in order to use it, you derive an object from the `Evictor.EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definition as follows:

```
package Evictor;

public abstract class EvictorBase implements Ice.ServantLocator
{
    public
    EvictorBase()
    {
        _size = 1000;
    }

    public
    EvictorBase(int size)
    {
        _size = size < 0 ? 1000 : size;
    }
```

```
    public abstract Ice.Object
    add(Ice.Current c, Ice.LocalObjectHolder cookie);

    public abstract void
    evict(Ice.Object servant, java.lang.Object cookie);

    synchronized public final Ice.Object
    locate(Ice.Current c, Ice.LocalObjectHolder cookie)
    {
        // ...
    }

    synchronized public final void
    finished(Ice.Current c, Ice.Object o, java.lang.Object cookie)
    {
        // ...
    }

    synchronized public final void
    deactivate(String category)
    {
        // ...
    }

    // ...

    private int _size;
}
```

Note that the evictor has constructors to set the size of the queue, with a default size of 1000.

The `locate`, `finished`, and `deactivate` methods are inherited from the `ServantLocator` base class; these methods implement the logic to maintain the queue in LRU order and to add and evict servants as needed. The methods are synchronized, so the evictor's internal data structures are protected from concurrent access.

The `add` and `evict` methods are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are abstract, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant. This allows

`evict` to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map does not only store servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue.
3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The last two points deserve some extra explanation.

- The evictor queue must be maintained in least-recently used order, that is, every time an invocation arrives and we find an entry for the identity in the evictor map, we also must locate the servant's identity on the evictor queue and move it to the front of the queue. However, scanning for that entry is inefficient because it requires $O(n)$ time. To get around this, we store an iterator in the evictor map that marks the corresponding entry's position in the evictor queue. This allows us to dequeue the entry from its current position and enqueue it at the head of the queue in $O(1)$ time.

  Unfortunately, the various lists provided by `java.util` do not allow us to keep an iterator to a list position without invalidating that iterator as the list is updated. To deal with this, we use a special-purpose linked list implementation, `Evictor.LinkedList`, that does not have this limitation. `LinkedList` has an interface similar to `java.util.LinkedList` but does not invalidate iterators other than iterators that point at an element that is removed. For brevity, we do not show the implementation of this list here—you can find the implementation in the code examples for this book in the Ice distribution.

- We maintain a use count as part of the map in order to avoid incorrect eviction of servants. Suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still executing, the servant for *I* could be evicted while it is still executing the original request.

  By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*. However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

  The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

This leads to the following definitions in the private section of our evictor:

```
package Evictor;

public abstract class EvictorBase implements Ice.ServantLocator
{
    // ...

    private class EvictorEntry
    {
        Ice.Object servant;
        java.lang.Object userCookie;
        java.util.Iterator<Ice.Identity> queuePos;
        int useCount;
    }
```

```
    private void evictServants()
    {
        // ...
    }

    private java.util.Map<Ice.Identity, EvictorEntry> _map =
        new java.util.HashMap<Ice.Identity, EvictorEntry>();
    private Evictor.LinkedList<Ice.Identity> _queue =
        new Evictor.LinkedList<Ice.Identity>();
    private int _size;
}
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. The map key is the identity of the Ice object, and the lookup value is of type `EvictorEntry`. The queue simply stores identities, of type `Ice::Identity`.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit—we will discuss this function in more detail shortly.

Almost all the action of the evictor takes place in the implementation of `locate`:

```
synchronized public final Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    //
    // Check if we have a servant in the map already.
    //
    EvictorEntry entry = _map.get(c.id);
    if (entry != null) {
        //
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        entry.queuePos.remove();
    } else {
        //
        // We do not have entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map.
        //
        entry = new EvictorEntry();
        Ice.LocalObjectHolder cookieHolder =
            new Ice.LocalObjectHolder();
        entry.servant = add(c, cookieHolder); // Down-call
        if (entry.servant == null) {
```

```
            return null;
        }
        entry.userCookie = cookieHolder.value;
        entry.useCount = 0;
        _map.put(c.id, entry);
    }

    //
    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(entry.useCount);
    _queue.addFirst(c.id);
    entry.queuePos = _queue.iterator();
    entry.queuePos.next(); // Position iterator on the element.

    cookie.value = entry;
    return entry.servant;
}
```

The code uses an `EvictorEntry` as the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`.

We first look for an existing entry in the evictor map, using the object identity as the key. If we have an entry in the map already, we dequeue the corresponding identity from the evictor queue. (The `queuePos` member of `EvictorEntry` is an iterator that marks that entry's position in the evictor queue.)

Otherwise, we do not have an entry in the map, so we create a new one and call the `add` method. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we return null to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The final few lines of code increment the entry's use count, add the entry at the head of the evictor queue, store the entry's position in the queue, and assign the entry to the cookie that is returned from `locate`, before returning the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

```
synchronized public final void
finished(Ice.Current c, Ice.Object o, java.lang.Object cookie)
{
    EvictorEntry entry = (EvictorEntry)cookie;

    // Decrement use count and check if
    // there is something to evict.
    //
    --(entry.useCount);
    evictServants();
}
```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

```
private void evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    java.util.Iterator<Ice.Identity> p = _queue.riterator();
    int excessEntries = _map.size() - _size;
    for (int i = 0; i < excessEntries; ++i) {
        Ice.Identity id = p.next();
        EvictorEntry e = _map.get(id);
        if (e.useCount == 0) {
            evict(e.servant, e.userCookie); // Down-call
            e.queuePos.remove();
            _map.remove(id);
        }
    }
}
```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

```
synchronized public final void
deactivate(String category)
{
    _size = 0;
    evictServants();
}
```

Note that, with this implementation of `evictServants`, we only scan the tail section of the evictor queue for servants to evict. If we have long-running operations, this allows the number of servants in the queue to remain above the evictor size if the servants in the tail section have a non-zero use count. This means that, even immediately after calling `evictServants`, the queue length can still exceed the evictor size.

We can adopt a more aggressive strategy for eviction: instead of scanning only the excess entries in the queue, if, after looking in the tail section of the queue, we still have more servants in the queue than the queue size, we keep scanning for servants with a zero use count until the queue size drops below the limit. This alternative version of `evictServants` looks as follows:

```
private void evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    java.util.Iterator<Ice.Identity> p = _queue.riterator();
    int numEntries = _map.size();
    for (int i = 0; i < excessEntries && _map.size() > _size;
         ++i) {
        Ice.Identity id = p.next();
        EvictorEntry e = _map.get(id);
        if (e.useCount == 0) {
            evict(e.servant, e.userCookie); // Down-call
            e.queuePos.remove();
            _map.remove(id);
        }
    }
}
```

The only difference in this version is that terminating condition for the `for`-loop has changed: instead of scanning only the excess entries for servants with a use count, this version keeps scanning until the evictor size drops below the limit.

Which version is more appropriate depends on your application: if locating and evicting servants is expensive, and memory is not at a premium, the first version (which only scans the tail section) is more appropriate; if you want to keep memory consumption to a minimum, the second version in more appropriate. Also keep in mind that the difference between the two versions is significant only if you have long-running operations and many concurrent invocations from clients; otherwise, there is no point in more aggressively scanning for servants to remove because they are going be become idle again very quickly and get evicted as soon as the next request arrives.

Creating an Evictor Implementation in C#

The `System.Collections` classes do not provide a container that does not invalidate iterators when we modify the contents of the container but, to efficiently implement an evictor, we need such a container. To deal with this, we use a special-purpose linked list implementation, `Evictor.LinkedList`, that does not invalidate iterators when we delete or add an element. For brevity, we only show the interface of `LinkedList` here—you can find the implementation in the code examples for this book in the Ice for .NET distribution.

```
namespace Evictor
{
    public class LinkedList<T> : ICollection<T>, ICollection,
                                 ICloneable
    {
        public LinkedList();

        public int Count { get; }

        public void Add(T value);
        public void AddFirst(T value);
        public void Clear();
        public bool Contains(T value);
        public bool Remove(T value);

        public IEnumerator GetEnumerator();

        public class Enumerator : IEnumerator<T>, IEnumerator,
                                  IDisposable
        {
            public void Reset();

            public T Current { get; }
```

```
            public bool MoveNext();
            public bool MovePrev();
            public void Remove();
            public void Dispose();
        }

        public void CopyTo(T[] array, int index);
        public void CopyTo(Array array, int index);

        public object Clone();

        public bool IsReadOnly { get; }
        public bool IsSynchronized { get; }
        public object SyncRoot { get; }
    }
}
```

The `Add` method appends an element to the list, and the `AddFirst` method prepends an element to the list. `GetEnumerator` returns an enumerator for the list elements; immediately after calling `GetEnumerator`, the enumerator does not point at any element until you call either `MoveNext` or `MovePrev`, which position the enumerator at the first and last element, respectively. `Current` returns the element at the enumerator position, and `Remove` deletes the element at the current position and leaves the enumerator pointing at no element. Calling `MoveNext` or `MovePrev` after calling `Remove` positions the enumerator at the element following or preceding the deleted element, respectively. `MoveNext` and `MovePrev` return true if they have positioned the enumerator on an element; otherwise, they return false and leave the enumerator position on the last and first element, respectively.

Given this `LinkedList`, we can implement the evictor. The evictor we show here is designed as an abstract base class: in order to use it, you derive an object from the `Evictor.EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definitions as follows:

```
namespace Evictor
{
    public abstract class EvictorBase
        : Ice.ServantLocator
    {
        public EvictorBase()
        {
            _size = 1000;
```

```
        }

        public EvictorBase(int size)
        {
            _size = size < 0 ? 1000 : size;
        }

        protected abstract Ice.Object add(Ice.Current c,
                                          out object cookie);

        protected abstract void evict(Ice.Object servant,
                                      object cookie);

        public Ice.Object locate(Ice.Current c,
                                 out object cookie)
        {
            lock(this)
            {
                // ...
            }
        }

        public void finished(Ice.Current c, Ice.Object o,
                             object cookie)
        {
            lock(this)
            {
                // ...
            }
        }

        public void deactivate(string category)
        {
            lock(this)
            {
                // ...
            }
        }

        private int _size;
    }
}
```

Note that the evictor has constructors to set the size of the queue, with a default size of 1000.

The `locate`, `finished`, and `deactivate` methods are inherited from the `ServantLocator` base class; these methods implement the logic to maintain the queue in LRU order and to add and evict servants as needed. The methods use a `lock(this)` statement for their body, so the evictor's internal data structures are protected from concurrent access.

The `add` and `evict` methods are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are abstract, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant. This allows `evict` to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map does not only store servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue.
3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The last two points deserve some extra explanation.

- The evictor queue must be maintained in least-recently used order, that is, every time an invocation arrives and we find an entry for the identity in the evictor map, we also must locate the servant's identity on the evictor queue and move it to the front of the queue. However, scanning for that entry is inefficient because it requires $O(n)$ time. To get around this, we store an iterator in the evictor map that marks the corresponding entry's position in the evictor queue. This allows us to dequeue the entry from its current position and enqueue it at the head of the queue in $O(1)$ time, using the `Evictor.LinkedList` implementation we saw on page 993.

- We maintain a use count as part of the map in order to avoid incorrect eviction of servants. Suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still executing, the servant for *I* could be evicted while it is still executing the original request.

  By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*. However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

  The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

This leads to the following definitions in the private section of our evictor:

```
namespace Evictor
{
    using System.Collections.Generic;

    public abstract class EvictorBase
        : Ice.ServantLocator
    {
        // ...

        private class EvictorEntry
        {
            internal Ice.Object servant;
            internal object userCookie;
            internal LinkedList<Ice.Identity>.Enumerator queuePos;
```

```
            internal int useCount;
        }

        private void evictServants()
        {
            // ...
        }

        private Dictionary<Ice.Identity, EvictorEntry> _map
            = new Dictionary<Ice.Identity, EvictorEntry>();
        private LinkedList<Ice.Identity> _queue =
            new LinkedList<Ice.Identity>();
        private int _size;
    }
}
```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. The map key is the identity of the Ice object, and the lookup value is of type `EvictorEntry`. The queue simply stores identities, of type `Ice.Identity`.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit—we will discuss this function in more detail shortly.

Almost all the action of the evictor takes place in the implementation of `locate`:

```
public Ice.Object locate(Ice.Current c,
                         out object cookie)
{
    lock(this)
    {
        //
        // Check if we a servant in the map already.
        //
        EvictorEntry entry = _map[c.id];
        if (entry != null) {
            //
            // Got an entry already, dequeue the entry from
            // its current position.
            //
            entry.queuePos.Remove();
        } else {
            //
            // We do not have an entry. Ask the derived class to
            // instantiate a servant and add an entry to the map.
```

```
            //
            entry = new EvictorEntry();
            entry.servant = add(c, out entry.userCookie);
            if (entry.servant == null) {
                cookie = null;
                return null;
            }
            entry.useCount = 0;
            _map[c.id] = entry;
        }

        //
        // Increment the use count of the servant and enqueue
        // the entry at the front, so we get LRU order.
        //
        ++(entry.useCount);
        _queue.AddFirst(c.id);
        entry.queuePos = (LinkedList<Ice.Identity>.Enumerator)
            _queue.GetEnumerator();
        entry.queuePos.MoveNext();

        cookie = entry;

        return entry.servant;
    }
}
```

The code uses an `EvictorEntry` as the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`.

We first look for an existing entry in the evictor map, using the object identity as the key. If we have an entry in the map already, we dequeue the corresponding identity from the evictor queue. (The `queuePos` member of `EvictorEntry` is an iterator that marks that entry's position in the evictor queue.)

Otherwise, we do not have an entry in the map, so we create a new one and call the `add` method. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we return null to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The final few lines of code increment the entry's use count, add the entry at the head of the evictor queue, store the entry's position in the queue, and initialize the

cookie that is returned from `locate`, before returning the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

```
public void finished(Ice.Current c, Ice.Object o,
                     object cookie)
{
    lock(this)
    {
        EvictorEntry entry = (EvictorEntry)cookie;

        //
        // Decrement use count and check if
        // there is something to evict.
        //
        --(entry.useCount);
        evictServants();
    }
}
```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

```
private void evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    LinkedList<Ice.Identity>.Enumerator p =
        (LinkedList<Ice.Identity>.Enumerator)
        _queue.GetEnumerator();
    int excessEntries = _map.Count - _size;
    for (int i = 0; i < excessEntries; ++i) {
        p.MovePrev();
        Ice.Identity id = p.Current;
        EvictorEntry e = _map[id];
        if (e.useCount == 0) {
            evict(e.servant, e.userCookie); // Down-call
            p.Remove();
```

```
            _map.Remove(id);
        }
    }
}
```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

```
public void deactivate(string category)
{
    lock(this)
    {
        _size = 0;
        evictServants();
    }
}
```

Note that, with this implementation of `evictServants`, we only scan the tail section of the evictor queue for servants to evict. If we have long-running operations, this allows the number of servants in the queue to remain above the evictor size if the servants in the tail section have a non-zero use count. This means that, even immediately after calling `evictServants`, the queue length can still exceed the evictor size.

We can adopt a more aggressive strategy for eviction: instead of scanning only the excess entries in the queue, if, after looking in the tail section of the queue, we still have more servants in the queue than the queue size, we keep scanning for servants with a zero use count until the queue size drops below the limit. This alternative version of `evictServants` looks as follows:

```
private void evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    LinkedList<Ice.Identity>.Enumerator p =
```

```
        (LinkedList<Ice.Identity>.Enumerator)
        _queue.GetEnumerator();
    int numEntries = _map.Count;
    for (int i = 0; i < numEntries && _map.Count > _size; ++i) {
        p.MovePrev();
        Ice.Identity id = p.Current;
        EvictorEntry e = _map[id];
        if (e.useCount == 0) {
            evict(e.servant, e.userCookie); // Down-call
            p.Remove();
            _map.Remove(id);
        }
    }
}
```

The only difference in this version is that terminating condition for the `for`-loop has changed: instead of scanning only the excess entries for servants with a use count, this version keeps scanning until the evictor size drops below the limit.

Which version is more appropriate depends on your application: if locating and evicting servants is expensive, and memory is not at a premium, the first version (which only scans the tail section) is more appropriate; if you want to keep memory consumption to a minimum, the second version in more appropriate. Also keep in mind that the difference between the two versions is significant only if you have long-running operations and many concurrent invocations from clients; otherwise, there is no point in more aggressively scanning for servants to remove because they are going be become idle again very quickly and get evicted as soon as the next request arrives.

Using Servant Evictors

Using a servant evictor is simply a matter of deriving a class from `EvictorBase` and implementing the `add` and `evict` methods. You can turn a servant locator into an evictor by simply taking the code that you wrote for `locate` and placing it into `add`—`EvictorBase` then takes care of maintaining the cache in least-recently used order and evicting servants as necessary. Unless you have clean-up requirements for your servants (such as closing network connections or database handles), the implementation of `evict` can be left empty.

One of the nice aspects of evictors is that you do not need to change anything in your servant implementation: the servants are ignorant of the fact that an evictor is in use. This makes it very easy to add an evictor to an already existing code base with little disturbance of the source code.

Evictors can provide substantial performance improvements over default servants: especially if initialization of servants is expensive (for example, because servant state must be initialized by reading from a network), an evictor performs much better than a default servant, while keeping memory requirements low.

32.10 The Ice Threading Model

Ice is inherently a multi-threaded platform. There is no such thing as a single-threaded server in Ice. As a result, you must concern yourself with concurrency issues: if a thread reads a data structure while another thread updates the same data structure, havoc will ensue unless you protect the data structure with appropriate locks. In order to build Ice applications that behave correctly, it is important that you understand the threading semantics of the Ice run time. This section discusses Ice's *thread pool* concurrency model and provides guidelines for writing thread-safe Ice applications.

32.10.1 Introduction to Thread Pools

A thread pool is a collection of threads that the Ice run time draws upon to perform specific tasks. Each communicator creates two thread pools:

- The *client thread pool* services outgoing connections, which primarily involves handling the replies to outgoing requests and includes notifying AMI callback objects. If a connection is used in bidirectional mode (see Section 36.7), the client thread pool also dispatches incoming callback requests.
- The *server thread pool* services incoming connections. It dispatches incoming requests and, for bidirectional connections, processes replies to outgoing requests.

By default, these two thread pools are shared by all of the communicator's object adapters. If necessary, you can configure individual object adapters to use a private thread pool instead.

If a thread pool is exhausted because all threads are currently dispatching a request, additional incoming requests are transparently delayed until a request completes and relinquishes its thread; that thread is then used to dispatch the next pending request. Ice minimizes thread context switches in a thread pool by using a leader-follower implementation (see [17]).

32.10.2 Configuring Thread Pools

Each thread pool has a unique name that serves as the prefix for its configuration properties:

- *name*`.Size`

  This property specifies the initial size of the thread pool. If not defined, the default value is one.

- *name*`.SizeMax`

  This property specifies the maximum size of the thread pool. If not defined, the default value is one. If the value of this property is less than that of *name*`.Size`, this property is adjusted to be equal to *name*`.Size`.

- *name*`.SizeWarn`

  This property sets a high water mark; when the number of threads in a pool reaches this value, the Ice run time logs a warning message. If you see this warning message frequently, it could indicate that you need to increase the value of *name*`.SizeMax`. The default value is zero, which disables the warning.

- *name*`.StackSize`

  This property specifies the number of bytes to use as the stack size of threads in the thread pool. The operating system's default is used if this property is not defined or is set to zero.

- *name*`.Serialize`

  Setting this property to a value greater than zero forces the thread pool to serialize all messages received over a connection. It is unnecessary to enable serialization for a thread pool whose maximum size is one because such a thread pool is already limited to processing one message at a time. For thread pools with more than one thread, serialization has a negative impact on latency and throughput. If not defined, the default value is zero.

  We discuss this feature in more detail in Section 32.10.4.

- *name*`.ThreadIdleTime`

  This property specifies the number of seconds that a thread in the thread pool must be idle before it terminates. The default value is 60 seconds if this property is not defined. Setting it to zero disables the termination of idle threads.

For configuration purposes, the names of the client and server thread pools are `Ice.ThreadPool.Client` and `Ice.ThreadPool.Server`, respectively.

As an example, the following properties establish the initial and maximum sizes for these thread pools:

```
Ice.ThreadPool.Client.Size=1
Ice.ThreadPool.Client.SizeMax=10
Ice.ThreadPool.Server.Size=1
Ice.ThreadPool.Server.SizeMax=10
```

To monitor the thread pool activities of the Ice run time, you can enable the `Ice.Trace.ThreadPool` property. Setting this property to a non-zero value causes the Ice run time to log a message when it creates a thread pool, as well as each time the size of a thread pool increases or decreases.

Dynamic Thread Pools

A *dynamic* thread pool can grow and shrink when necessary in response to changes in an application's work load. All thread pools have at least one thread, but a dynamic thread pool can grow as the demand for threads increases, up to the pool's maximum size. Threads may also be terminated automatically when they have been idle for some time.

The dynamic nature of a thread pool is determined by the configuration properties *name*`.Size`, *name*`.SizeMax`, and *name*`.ThreadIdleTime`. A thread pool is not dynamic in its default configuration because *name*`.Size` and *name*`.SizeMax` are both set to one, meaning the pool can never grow to contain more than a single thread. To configure a dynamic thread pool, you must set at least one of *name*`.Size` or *name*`.SizeMax` to a value greater than one. We can use several configuration scenarios to explore the semantics of dynamic thread pools in greater detail:

- *name*`.SizeMax=5`

  This thread pool initially contains a single thread because *name*`.Size` has a default value of one, and Ice can grow the pool up to the maximum of five threads. During periods of inactivity, idle threads terminate after 60 seconds (the default value for *name*`.ThreadIdleTime`) until the pool contains just one thread again.

- *name*`.Size=3`
  name`.SizeMax=5`

  This thread pool starts with three active threads but otherwise behaves the same as in the previous configuration. The pool can still shrink to a size of one as threads become idle.

- *name*.Size=3
  name.ThreadIdleTime=10

  This thread pool starts with three active threads and shrinks quickly to one thread during periods of inactivity. As demand increases again, the thread pool can return to its maximum size of three threads (*name*.SizeMax defaults to the value of *name*.Size).

- *name*.SizeMax=5
  name.ThreadIdleTime=0

  This thread pool can grow from its initial size of one thread to contain up to five threads, but it will never shrink because *name*.ThreadIdleTime is set to zero.

- *name*.Size=5
  name.ThreadIdleTime=0

  This thread pool starts with five threads and can neither grow nor shrink.

To summarize, the value of *name*.ThreadIdleTime determines whether (and how quickly) a thread pool can shrink to a size of one. A thread pool that shrinks can also grow to its maximum size. Finally, setting *name*.SizeMax to a value larger than *name*.Size allows a thread pool to grow beyond its initial capacity.

32.10.3 Adapter Thread Pools

The default behavior of an object adapter is to share the thread pools of its communicator and, for many applications, this behavior is entirely sufficient. However, the ability to configure an object adapter with its own thread pool is useful in certain situations:

- When the concurrency requirements of an object adapter does not match those of its communicator.

  In a server with multiple object adapters, the configuration of the communicator's client and server thread pools may be a good match for some object adapters, but others may have different requirements. For example, the servants hosted by one object adapter may not support concurrent access, in which case limiting that object adapter to a single-threaded pool eliminates the need for synchronization in those servants. On the other hand, another object adapter might need a multi-threaded pool for better performance.

- To ensure that a minimum number of threads is available for dispatching requests to an adapter's servants. This is especially important for eliminating

the possibility of deadlocks when using nested invocations (see Section 32.10.5).

An object adapter's thread pool supports all of the properties described in Section 32.10.2. For configuration purposes, the name of an adapter's thread pool is *adapter*`.ThreadPool`, where *adapter* is the name of the adapter.

An adapter creates its own thread pool when at least one of the following properties has a value greater than zero:

- *adapter*`.ThreadPool.Size`
- *adapter*`.ThreadPool.SizeMax`

These properties have the same semantics as those described earlier except they both have a default value of zero, meaning that an adapter uses the communicator's thread pools by default.

As an example, the properties shown below configure a thread pool for the object adapter named `PrinterAdapter`:

```
PrinterAdapter.ThreadPool.Size=3
PrinterAdapter.ThreadPool.SizeMax=15
PrinterAdapter.ThreadPool.SizeWarn=14
```

32.10.4 Design Considerations

Improper configuration of a thread pool can have a serious impact on the performance of your application. This section discusses some issues that you should consider when designing and configuring your applications.

Single-Threaded Pool

There are several implications of using a thread pool with a maximum size of one thread:

- Only one message can be dispatched at a time.

  This can be convenient because it lets you avoid (or postpone) dealing with thread-safety issues in your application (see Section 32.10.6). However, it also eliminates the possibility of dispatching requests concurrently, which can be a bottleneck for applications running on multi-CPU systems or that perform blocking operations. Another option is to enable serialization in a multi-threaded pool, as discussed on page 1008.

- Only one AMI reply can be processed at a time.

  An application must increase the size of the client thread pool in order to process multiple AMI callbacks in parallel.

- Nested twoway invocations are limited.

  At most one level of nested twoway invocations is possible. (See Section 32.10.5.)

It is important to remember that a communicator's client and server thread pools have a default maximum size of one thread, therefore these limitations also apply to any object adapter that shares the communicator's thread pools.

Multi-Threaded Pool

Configuring a thread pool to support multiple threads implies that the application is prepared for the Ice run time to dispatch operation invocations or AMI callbacks concurrently. Although greater effort is required to design a thread-safe application, you are rewarded with the ability to improve the application's scalability and throughput.

Choosing an appropriate maximum size for a thread pool requires careful analysis of your application. For example, in compute-bound applications it is best to limit the number of threads to the number of physical processors in the host machine; adding any more threads only increases context switches and reduces performance. Increasing the size of the pool beyond the number of processors can improve responsiveness when threads can become blocked while waiting for the operating system to complete a task, such as a network or file operation. On the other hand, a thread pool configured with too many threads can have the opposite effect and negatively impact performance. Testing your application in a realistic environment is the recommended way of determining the optimum size for a thread pool.

If your application uses nested invocations, it is very important that you evaluate whether it is possible for thread starvation to cause a deadlock. Increasing the size of a thread pool can lessen the chance of a deadlock, but other design solutions are usually preferred. Section 32.10.5 discusses nested invocations in more detail.

Serialization

When using a multi-threaded pool, the nondeterministic nature of thread scheduling means that requests from the same connection may not be dispatched in the order they were received. Some applications cannot tolerate this behavior, such as a transaction processing server that must guarantee that requests are executed in order. There are two ways of satisfying this requirement:

1. Use a single-threaded pool.

2. Configure a multi-threaded pool to serialize requests using its `Serialize` property (see Section 32.10.2).

At first glance these two options may seem equivalent, but there is a significant difference: a single-threaded pool can only dispatch one request at a time and therefore serializes requests from *all* connections, whereas a multi-threaded pool configured for serialization can dispatch requests from different connections concurrently while serializing requests from the same connection.

You can obtain the same behavior from a multi-threaded pool without enabling serialization, but only if you design the clients so that they do not send requests from multiple threads, do not send requests over more than one connection, and only use synchronous twoway invocations. In general, however, it is better to avoid such tight coupling between the implementations of the client and server.

Enabling serialization can improve responsiveness and performance compared to a single-threaded pool, but there is an associated cost. The extra synchronization that the pool must perform to serialize requests adds significant overhead and results in higher latency and reduced throughput.

As you can see, thread pool serialization is not a feature that you should enable without analyzing whether the benefits are worthwhile. For example, it might be an inappropriate choice for a server with long-running operations when the client needs the ability to have several operations in progress simultaneously. If serialization was enabled in this situation, the client would be forced to work around it by opening several connections to the server (see Section 36.3), which again tightly couples the client and server implementations. If the server must keep track of the order of client requests, a better solution would be to use serialization in conjunction with asynchronous dispatch to queue the incoming requests for execution by other threads.

32.10.5 Nested Invocations

A *nested invocation* is one that is made within the context of another Ice operation. For instance, the implementation of an operation in a servant might need to make a nested invocation on some other object, or an AMI callback object might invoke an operation in the course of processing a reply to an asynchronous request. It is also possible for one of these invocations to result in a nested callback to the originating process. The maximum depth of such invocations is determined by the size of the thread pools used by the communicating parties.

Deadlocks

Applications that use nested invocations must be carefully designed to avoid the potential for deadlock, which can easily occur when invocations take a circular path. For example, Figure 32.5 presents a deadlock scenario when using the default thread pool configuration.

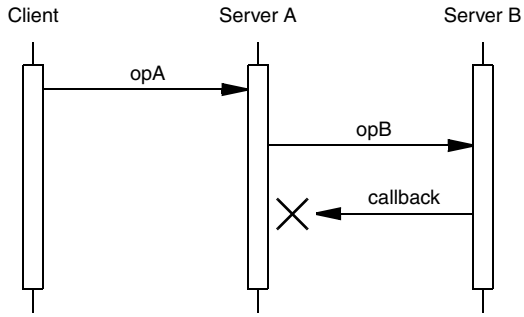


Figure 32.5. Nested invocation deadlock.

In this diagram, the implementation of `opA` makes a nested twoway invocation of `opB`, but the implementation of `opB` causes a deadlock when it tries to make a nested callback. As mentioned in Section 32.10.1, the communicator's thread pools have a maximum size of one thread unless explicitly configured otherwise. In Server A, the only thread in the server thread pool is busy waiting for its invocation of `opB` to complete, and therefore no threads remain to handle the callback from Server B. The client is now blocked because Server A is blocked, and they remain blocked indefinitely unless timeouts are used.

There are several ways to avoid a deadlock in this scenario:

- Increase the maximum size of the server thread pool in Server A.

  Configuring the server thread pool in Server A to support more than one thread allows the nested callback to proceed. This is the simplest solution, but it requires that you know in advance how deeply nested the invocations may occur, or that you set the maximum size to a sufficiently large value that exhausting the pool becomes unlikely. For example, setting the maximum size to two avoids a deadlock when a single client is involved, but a deadlock could easily occur again if multiple clients invoke `opA` simultaneously. Furthermore, setting the maximum size too large can cause its own set of problems (see Section 32.10.4).

- Use a oneway invocation.

  If Server A called `opB` using a oneway invocation, it would no longer need to wait for a response and therefore `opA` could complete, making a thread available to handle the callback from Server B. However, we have made a significant change in the semantics of `opA` because now there is no guarantee that `opB` has completed before `opA` returns, and it is still possible for the oneway invocation of `opB` to block (see Section 32.14).

- Create another object adapter for the callbacks.

  No deadlock occurs if the callback from Server B is directed to a different object adapter that is configured with its own thread pool (see Section 32.10.3).

- Implement `opA` using asynchronous dispatch and invocation.

  By declaring `opA` as an AMD operation and invoking `opB` using AMI, Server A can avoid blocking the thread pool's thread while it waits for `opB` to complete. This technique, known as *asynchronous request chaining*, is used extensively in Ice services such as IceGrid and Glacier2 to eliminate the possibility of deadlocks.

As another example, consider a client that makes a nested invocation from an AMI callback object using the default thread pool configuration. The (one and only) thread in the client thread pool receives the reply to the asynchronous request and invokes its callback object. If the callback object in turn makes a nested twoway invocation, a deadlock occurs because no more threads are available in the client thread pool to process the reply to the nested invocation. The solutions are similar to some of those presented for Figure 32.5: increase the maximum size of the client thread pool, use a oneway invocation, or call the nested invocation using AMI.

Analyzing an Application

A number of factors must be considered when evaluating whether an application is properly designed and configured for nested invocations:

- The thread pool configurations in use by all communicating parties have a significant impact on an application's ability to use nested invocations. While analyzing the path of circular invocations, you must pay careful attention to the threads involved to determine whether sufficient threads are available to avoid deadlock. This includes not just the threads that dispatch requests, but also the threads that make the requests and process the replies. Enabling the

`Ice.Trace.ThreadPool` can give you a better understanding of the thread pool behavior in your application.

- Bidirectional connections are another complication, since you must be aware of which threads are used on either end of the connection.
- Finally, the synchronization activities of the communicating parties must also be scrutinized. For example, a deadlock is much more likely when a lock is held while making an invocation.

As you can imagine, tracing the call flow of a distributed application to ensure there is no possibility of deadlock can quickly become a complex and tedious process. In general, it is best to avoid circular invocations if at all possible.

32.10.6 Thread Safety

The Ice run time itself is fully thread safe, meaning multiple application threads can safely call methods on objects such as communicators, object adapters, and proxies without synchronization problems. As a developer, you must also be concerned with thread safety because the Ice run time can dispatch multiple invocations concurrently in a server. In fact, it is possible for multiple requests to proceed in parallel within the same servant and within the same operation on that servant. It follows that, if the operation implementation manipulates non-stack storage (such as member variables of the servant or global or static data), you must interlock access to this data to avoid data corruption.

The need for thread safety in an application depends on its configuration. Using the default thread pool configuration typically makes synchronization unnecessary because at most one operation can be dispatched at a time. Thread safety becomes an issue once you increase the maximum size of a thread pool.

Ice uses the native synchronization and threading primitives of each platform. For C++ users, Ice provides a collection of convenient and portable wrapper classes for use by Ice applications (see Chapter 31).

Marshaling Issues

The marshaling semantics of the Ice run time present a subtle thread safety issue that arises when an operation returns data by reference. In C++, the only relevant case is returning an instance of a Slice class, either directly or nested as a member of another type. In Java, .NET, and Python, Slice structures, sequences, and dictionaries are also affected.

The potential for corruption occurs whenever a servant returns data by reference, yet continues to hold a reference to that data. For example, consider the following Java implementation:

```
public class GridI extends _GridDisp
{
    GridI()
    {
        _grid = // ...
    }

    public int[][]
    getGrid(Ice.Current curr)
    {
        return _grid;
    }

    public void
    setValue(int x, int y, int val, Ice.Current curr)
    {
        _grid[x][y] = val;
    }

    private int[][] _grid;
}
```

Suppose that a client invoked the `getGrid` operation. While the Ice run time marshals the returned array in preparation to send a reply message, it is possible for another thread to dispatch the `setValue` operation on the same servant. This race condition can result in several unexpected outcomes, including a failure during marshaling or inconsistent data in the reply to `getGrid`. Synchronizing the `getGrid` and `setValue` operations would not fix the race condition because the Ice run time performs its marshaling outside of this synchronization.

One solution is to implement accessor operations, such as `getGrid`, so that they return copies of any data that might change. There are several drawbacks to this approach:

- Excessive copying can have an adverse affect on performance.
- The operations must return deep copies in order to avoid similar problems with nested values.
- The code to create deep copies is tedious and error-prone to write.

Another solution is to make copies of the affected data only when it is modified. In the revised code shown below, `setValue` replaces `_grid` with a copy that contains the new element, leaving the previous contents of `_grid` unchanged:

```
public class GridI extends _GridDisp
{
    ...

    public synchronized int[][]
    getGrid(Ice.Current curr)
    {
        return _grid;
    }

    public synchronized void
    setValue(int x, int y, int val, Ice.Current curr)
    {
        int[][] newGrid = // shallow copy...
        newGrid[x][y] = val;
        _grid = newGrid;
    }

    ...
}
```

This allows the Ice run time top safely marshal the return value of `getGrid` because the array is never modified again. For applications where data is read more often than it is written, this solution is more efficient than the previous one because accessor operations do not need to make copies. Furthermore, intelligent use of shallow copying can minimize the overhead in mutating operations.

Finally, a third approach changes accessor operations to use AMD in order to regain control over marshaling. After annotating the `getGrid` operation with `amd` metadata, we can revise the servant as follows:

```
public class GridI extends _GridDisp
{
    ...

    public synchronized void
    getGrid_async(AMD_Grid_getGrid cb, Ice.Current curr)
    {
        cb.ice_response(_grid);
    }

    public synchronized void
```

```
    setValue(int x, int y, int val, Ice.Current curr)
    {
        _grid[x][y] = val;
    }

    ...
}
```

Normally, AMD is used in situations where the servant needs to delay its response to the client without blocking the calling thread. For getGrid, that is not the goal; instead, as a side-effect, AMD provides the desired marshaling behavior. Specifically, the Ice run time marshals the reply to an asynchronous request at the time the servant invokes `ice_response` on the AMD callback object. Because `getGrid` and `setValue` are synchronized, this guarantees that the data remains in a consistent state during marshaling.

Thread Creation and Destruction Hooks

On occasion, it is necessary to intercept the creation and destruction of threads created by the Ice run time, for example, to interoperate with libraries that require applications to make thread-specific initialization and finalization calls (such as COM's `CoInitializeEx` and `CoUninitialize`). Ice provides callbacks to inform an application when each run-time thread is created and destroyed. For C++, the callback class looks as follows:[10]

```
class ThreadNotification : public IceUtil::Shared {
public:
    virtual void start() = 0;
    virtual void stop() = 0;
};
typedef IceUtil::Handle<ThreadNotification> ThreadNotificationPtr;
```

To receive notification of thread creation and destruction, you must derive a class from `ThreadNotification` and implement the `start` and `stop` member functions. These functions will be called by the Ice run by each thread as soon as it is created, and just before it exits. You must install your callback class in the Ice run time when you create a communicator by setting the `threadHook` member of the `InitializationData` structure (see Section 32.3).

For example, you could define a callback class and register it with the Ice run time as follows:

10.See below for other languages.

```
class MyHook : public virtual Ice::ThreadNotification {
public:
    void start()
    {
        cout << "start: id = " << ThreadControl().id() << endl;
    }
    void stop()
    {
        cout << "stop: id = " << ThreadControl().id() << endl;
    }
};

int
main(int argc, char* argv[])
{
    // ...

    Ice::InitializationData id;
    id.threadHook = new MyHook;
    communicator = Ice::initialize(argc, argv, id);

    // ...
}
```

The implementation of your `start` and `stop` methods can make whatever thread-specific calls are required by your application.

For Java and C#, `Ice.ThreadNotification` is an interface:

```
public interface ThreadNotification {
    void start();
    void stop();
}
```

To receive the thread creation and destruction callbacks, you must derive a class from this interface that implements the `start` and `stop` methods, and register an instance of that class when you create the communicator. (The code to do this is analogous to the C++ version.)

For Python, the interface is:

```
class ThreadNotification(object):
    def __init__(self):
        pass

    # def start():
    # def stop():
```

The Ice run time calls the `start` and `stop` methods of the class instance you provide to `Ice.initialize` (see Section 24.3) when it creates and destroys threads.

Installing Thread Hooks via a Plug-in

The thread hook facility described on page 1015 requires that you modify a program's source code in order to receive callbacks when threads in the Ice run time are created and destroyed. It is also possible to install thread hooks using the Ice plug-in facility (see Section 32.25), which is useful for adding thread hooks to an existing program that you cannot (or prefer not to) modify.

Ice provides a base class named `ThreadHookPlugin` for C++, Java, and C# that supplies the necessary functionality. The C++ class definition is shown below:

```
namespace Ice {
class ThreadHookPlugin : public Ice::Plugin {
public:

    ThreadHookPlugin(const CommunicatorPtr& communicator,
                     const ThreadNotificationPtr&);

    virtual void initialize();

    virtual void destroy();
};
}
```

The equivalent definitions for Java and C# are quite similar and therefore not presented here.

The `ThreadHookPlugin` constructor installs the given `ThreadNotification` object into the specified communicator. The `initialize` and `destroy` methods are empty, but you can subclass `ThreadHookPlugin` and override these methods if necessary.

In order to create a thread hook plug-in, you must do the following:

- Define and export a factory class (for Java and C#) or factory function (for C++) that returns an instance of `ThreadHookPlugin` (see Section 32.25.1).
- Implement the `ThreadNotification` object that you will pass to the `ThreadHookPlugin` constructor.
- Package your code into a format that is suitable for dynamic loading, such as a shared library or DLL for C++ or an assembly for C#.

To install your plug-in, use a configuration property like the one shown below:

```
Ice.Plugin.MyThreadHookPlugin=MyHooks:createPlugin ...
```

The first component of the property value represents the plug-in's entry point. For C++, this value includes the abbreviated name of the shared library or DLL (`MyHooks`) and the name of a factory function (`createPlugin`).

If your property value is language-specific and the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice run time for a certain language. For example, here is the C++-specific version:

```
Ice.Plugin.MyThreadHookPlugin.cpp=MyHooks:createPlugin ...
```

Refer to Appendix D for more information on the `Ice.Plugin` properties.

32.10.7 Dispatching Invocations to User Threads

By default, operation invocations and AMI callbacks are executed by a thread from a thread pool. This behavior is simple and convenient for applications because they need not concern themselves with thread creation and destruction. However, there are situations where it is necessary to respond to operation invocations or AMI callbacks in a particular thread. For example, in a server, you might need to update a database that does not permit concurrent access from different threads or, in a client, you might need to update a user interface with the results of an invocation. (Many UI frameworks require all UI updates to be made by a specific thread.)

In Ice for C++, Java, and .NET, you can control which thread receives operation invocations and AMI callbacks, so you can ensure that all updates are made by a thread you choose. The implementation techniques vary slightly for each language and are explained in the sections that follow.

C++ Dispatcher API

To install a dispatcher, you must instantiate a class that derives from `Ice::Dispatcher` and initialize a communicator with that instance in the `InitializationData` structure. All invocations that arrive for this communicator are made via the specified dispatcher. For example:

```
class MyDispatcher : public Ice::Dispatcher /*, ... */
    // ...
};
```

```
int
main(int argc, char* argv[])
{
    Ice::CommunicatorPtr communicator;

    try {
        Ice::InitializationData initData;
        initData.properties = Ice::createProperties(argc, argv);
        initData.dispatcher = new MyDispatcher();
        communicator = Ice::initialize(argc, argv, initData);

        // ...
    } catch (const Ice::Exception& ex) {
        // ...
    }

    // ...
}
```

The `Ice::Dispatcher` abstract base class has the following interface:

```
class Dispatcher : virtual public IceUtil::Shared
{
public:
    virtual void dispatch(const DispatcherCallPtr&,
                          const ConnectionPtr&) = 0;
};

typedef IceUtil::Handle<Dispatcher> DispatcherPtr;
```

The Ice run time invokes the `dispatch` method whenever an operation invocation arrives or an AMI invocation completes, passing an instance of `DispatcherCall` and the connection via which the invocation arrived. The job of `dispatch` is to pass the incoming invocation to an operation implementation. (The connection parameter allows you to decide how to dispatch the operation based on the connection via which it was received.)

You can write `dispatch` such that it blocks and waits for completion of the invocation because `dispatch` is called by a thread in the server-side thread pool (for incoming operation invocations) or the client-side thread pool (for AMI callbacks).

The `DispatcherCall` instance encapsulates all the details of the incoming call. It is another abstract base class with the following interface:

```
class DispatcherCall : virtual public IceUtil::Shared
{
public:
    virtual ~DispatcherCall() { }

    virtual void run() = 0;
};

typedef IceUtil::Handle<DispatcherCall> DispatcherCallPtr;
```

Your implementation of `dispatch` is expected to call `run` on the `DispatcherCall` instance (or, more commonly, to cause `run` to be called some time later). When you call `run`, the Ice run time processes the invocation in the thread that calls `run`.

A very simple way to implement `dispatch` would be as follows:

```
class MyDispatcher : public Ice::Dispatcher
public:
    virtual void dispatch(const Ice::DispatcherCallPtr& d,
                          const Ice::ConnectionPtr)
    {
        d->run(); // Does not throw, blocks until op completes.
    }
};
```

Whenever the Ice run time receives an incoming operation invocation or when an AMI invocation completes, it calls `dispatch` which, in turn, calls `run` on the `DispatcherCall` instance.

With this simple example, `dispatch` immediately calls `run`, and `run` does not return until the corresponding operation invocation is complete. As a result, this implementation ties up a thread in the thread pool for the duration of the call.

So far, we really have not gained anything because all we have is a callback method that is called by the Ice run time. However, this simple mechanism is sufficient to ensure that we can update a UI from the correct thread.

A common technique to avoid blocking is to use asynchronous method invocation (see Section 6.15). In response to a UI event (such as the user pressing a "Submit" button), the application initiates an operation invocation from the corresponding UI callback by calling the operation's `begin_` method. This is guaranteed not to block the caller, so the UI remains responsive. Some time later, when the operation completes, the Ice run time invokes an AMI callback from one of the threads in its thread pool. That callback now has to update the UI, but that can only be done from the UI thread. By using a dispatcher, you can easily delegate

the update to the correct thread. For example, here is how you can arrange for AMI callbacks to be passed to the UI thread with MFC:

```
class MyDialog : public CDialog { ... };

class MyDispatcher : public Ice::Dispatcher {
public:
    MyDispatcher(MyDialog* dialog) : _dialog(dialog)
    {
    }

    virtual void
    dispatch(const Ice::DispatcherCallPtr& call,
             const Ice::ConnectionPtr&)
    {
        _dialog->PostMessage(
            WM_AMI_CALLBACK, 0,
            reinterpret_cast<LPARAM>(
                new Ice::DispatcherCallPtr(call)));
    }

private:
    MyDialog* _dialog;
};
```

The `MyDispatcher` class simply stores the `CDialog` handle for the UI and calls `PostMessage`, passing the `DispatcherCall` instance. In turn, this causes the UI thread to receive an event and invoke the UI callback method that was registered to respond to `WM_AMI_CALLBACK` events.

In turn, the implementation of the callback method calls `run`:

```
LRESULT
MyDialog::OnAMICallback(WPARAM, LPARAM lParam)
{
    try {
        Ice::DispatcherCallPtr* call =
            reinterpret_cast<Ice::DispatcherCallPtr*>(lParam);
        (*call)->run();
        delete call;
    } catch (const Ice::Exception& ex) {
        // ...
    }
    return 0;
}
```

The Ice run time calls `dispatch` once the asynchronous operation invocation is complete. In turn, this causes the `OnAMICallback` to trigger, which calls `run`. Because the operation has completed already, `run` does not block, so the UI remains responsive.

Please see the `MFC` demo in your Ice distribution for a fully-functional UI client that uses this technique.

Java Dispatcher API

To install a dispatcher, you must instantiate a class that implements `Ice.Dispatcher` and initialize a communicator with that instance in the `InitializationData` structure. All invocations that arrive for this communicator are made via the specified dispatcher. For example:

```
public class MyDispatcher implements Ice.Dispatcher
{
    // ...
}

public class Server
{
    public static void
    main(String[] args)
    {
        Ice.Communicator communicator;

        try {
            Ice.InitializationData initData =
                    new Ice.InitializationData();
            initData.properties = Ice.Util.createProperties(args);
            initData.dispatcher = new MyDispatcher();
            communicator = Ice.Util.initialize(args, initData);

            // ...
        } catch (Ice.LocalException & ex) { {
            // ...
        }

        // ...
    }

    // ...
}
```

The `Ice.Dispatcher` interface looks as follows:

```
public interface Dispatcher
{
    void dispatch(Runnable runnable, Ice.Connection con);
}
```

The Ice run time invokes the `dispatch` method whenever an operation invocation arrives, passing a `Runnable` and the connection via which the invocation arrived. The job of `dispatch` is to pass the incoming invocation to an operation implementation. (The `con` parameter allows you to decide how to dispatch the operation based on the connection via which it was received.)

You can write `dispatch` such that it blocks and waits for completion of the invocation because `dispatch` is called by a thread in the server-side thread pool (for incoming operation invocations) or the client-side thread pool (for AMI callbacks).

Your implementation of `dispatch` is expected to call `run` on the `Runnable` instance (or, more commonly, to cause `run` to be called some time later). When you call `run`, the Ice run time processes the invocation in the thread that calls `run`.

A very simple way to implement `dispatch` would be as follows:

```
public class MyDispatcher implements Ice.Dispatcher
{
    public void
    dispatch(Runnable runnable, Ice.Connection connection)
    {
        // Does not throw, blocks until op completes.
        runnable.run();
    }
}
```

Whenever the Ice run time receives an incoming operation invocation or when an AMI invocation completes, it calls `dispatch` which, in turn, calls `run` on the `Runnable` instance.

With this simple example, `dispatch` immediately calls `run`, and `run` does not return until the corresponding operation invocation is complete. As a result, this implementation ties up a thread in the thread pool for the duration of the call.

So far, we really have not gained anything because all we have is a callback method that is called by the Ice run time. However, this simple mechanism is sufficient to ensure that we can update a UI from the correct thread.

A common technique to avoid blocking is to use asynchronous method invocation (see Section 10.17). In response to a UI event (such as the user pressing a "Submit" button), the application initiates an operation invocation from the corre-

sponding UI callback by calling the operation's `begin_` method. This is guaranteed not to block the caller, so the UI remains responsive. Some time later, when the operation completes, the Ice run time invokes an AMI `response` callback from one of the threads in its thread pool. That callback now has to update the UI, but that can only be done from the UI thread. By using a dispatcher, you can easily delegate the update to the correct thread. For example, here is how you can arrange for AMI callbacks to be passed to the UI thread with Swing:

```
public class Client extends JFrame
{
    public static void main(final String[] args)
    {
        SwingUtilities.invokeLater(new Runnable()
        {
            public void run()
            {
                try {
                    new Client(args);
                } catch (Ice.LocalException e) {
                    JOptionPane.showMessageDialog(
                        null, e.toString(),
                        "Initialization failed",
                        JOptionPane.ERROR_MESSAGE);
                }
            }
        });
    }

    Client(String[] args)
    {
        Ice.Communicator communicator;

        try {
            Ice.InitializationData initData =
                new Ice.InitializationData();
            initData.dispatcher = new Ice.Dispatcher()
            {
                public void
                dispatch(Runnable runnable,
                         Ice.Connection connection)
                {
                    SwingUtilities.invokeLater(runnable);
                }
            };
            communicator = Ice.Util.initialize(args, initData);
```

```
        }
        catch(Throwable ex)
        {
            // ...
        }
        // ...
    }

    // ...
}
```

The `dispatch` method simply delays the call to `run` by calling `invokeLater`, passing it the `Runnable` that is provided by the Ice run time. This causes the Swing UI thread to eventually make the call to `run`. Because the Ice run time does not call `dispatch` until the asynchronous invocation is complete, that call to `run` does not block and the UI remains responsive.

Please see the `swing` demo in your Ice distribution for a fully-functional UI client that uses this technique.

C# Dispatcher API

To install a dispatcher, you must initialize a communicator with a delegate of type `Ice.Dispatcher` in the `InitializationData` structure. All invocations that arrive for this communicator are made via the specified dispatcher. For example:

```
public class Server
{
    public static void Main(string[] args)
    {
        Ice.Communicator communicator = null;

        try {
            Ice.InitializationData initData =
                    new Ice.InitializationData();
            initData.dispatcher = new MyDispatcher().dispatch;
            communicator =
                Ice.Util.initialize(ref args, initData);
            // ...
        } catch (System.Exception ex) {
            // ...
        }

        // ...
```

```
    }

    // ...
}
```

The `Ice.Dispatcher` delegate is defined as follows:

```
public delegate void Dispatcher(System.Action call,
                                Connection con);
```

The Ice run time calls your delegate whenever an operation invocation arrives, passing a `System.Action` delegate and the connection via which the invocation arrived. The job of your delegate is to pass the incoming invocation to an operation implementation. (The `con` parameter allows you to decide how to dispatch the operation based on the connection via which it was received.)

In this example, the delegate calls a method `dispatch` on an instance of a `MyDispatcher` class. You can write `dispatch` such that it blocks and waits for completion of the invocation because `dispatch` is called by a thread in the server-side thread pool (for incoming operation invocations) or the client-side thread pool (for AMI callbacks).

Your implementation of `dispatch` is expected to invoke the `call` delegate (or, more commonly, to cause it to be invoked some time later). When you invoke the `call` delegate, the Ice run time processes the invocation in the thread that invokes the delegate.

A very simple way to implement `dispatch` would be as follows:

```
public class MyDispatcher
{
    public void
    dispatch(System.Action call, Ice.Connection con)
    {
        // Does not throw, blocks until op completes.
        call();
    }
};
```

Whenever the Ice run time receives an incoming operation invocation or when an AMI invocation completes, it calls `dispatch` which, in turn, invokes the `call` delegate.

With this simple example, `dispatch` immediately invokes the delegate, and that call does not return until the corresponding operation invocation is complete. As a result, this implementation ties up a thread in the thread pool for the duration of the call.

So far, we really have not gained anything because all we have is a callback method that is called by the Ice run time. However, this simple mechanism is sufficient to ensure that we can update a UI from the correct thread.

A common technique to avoid blocking is to use asynchronous method invocation (see Section 14.16). In response to a UI event (such as the user pressing a "Submit" button), the application initiates an operation invocation from the corresponding UI callback by calling the operation's `begin_` method. This is guaranteed not to block the caller, so the UI remains responsive. Some time later, when the operation completes, the Ice run time invokes an AMI callback from one of the threads in its thread pool. That callback now has to update the UI, but that can only be done from the UI thread. By using a dispatcher, you can easily delegate the update to the correct thread. For example, here is how you can arrange for AMI callbacks to be passed to the UI thread with WPF:

```
public partial class MyWindow : Window
{
    private void Window_Loaded(object sender, EventArgs e)
    {
        Ice.Communicator communicator = null;

        try
        {
            Ice.InitializationData initData =
                    new Ice.InitializationData();
            initData.dispatcher =
                    delegate(System.Action action,
                             Ice.Connection connection)
            {
                Dispatcher.BeginInvoke(
                    DispatcherPriority.Normal, action);
            };
            communicator = Ice.Util.initialize(initData);
        }
        catch(Ice.LocalException ex)
        {
            // ...
        }
    }

    // ...
}
```

The delegate calls `Dispatcher.BeginInvoke` on the `action` delegate. This causes WPF to queue the actual asynchronous invocation of `action` for

later execution by the UI thread. Because the Ice run time does not invoke your delegate until an asynchronous operation invocation is complete, when the UI thread executes the corresponding call to the `EndInvoke` method, that call does not block and the UI remains responsive.

The net effect is that you can invoke an operation asynchronously from a UI callback method without the risk of blocking the UI thread. For example:

```
public partical class MyWindow : Window
{
    private void someOp_Click(object sender, RoutedEventArgs e)
    {
        MyIntfPrx p = ...;

        // Call remote operation asynchronously.
        // Response is processed in UI thread.
        p.begin_someOp().whenCompleted(this.opResponse,
                                       this.opException);
    }

    public void opResponse()
    {
        // Update UI...
    }

    public void opException(Ice.Exception ex)
    {
        // Update UI...
    }
}
```

Please see the `wpf` demo in your Ice distribution for a fully-functional UI client that uses this technique.

Implementation Notes

A call to `Communicator::destroy` will hang indefinitely if there are pending requests that have not yet been dispatched, therefore it is very important for a dispatcher implementation to ensure that all requests are dispatched.

32.11 Proxies

The introduction to proxies provided in Section 2.2.2 describes a proxy as a local artifact that makes a remote invocation as easy to use as a regular function call. In

fact, processing remote invocations is just one of a proxy's many responsibilities. A proxy also encapsulates the information necessary to contact the object, including its identity (see Section 32.5) and addressing details such as endpoints (see Section 32.11.3). Proxy methods provide access to configuration and connection information, and act as factories for creating new proxies (see Section 32.11.2). Finally, a proxy initiates the establishment of a new connection when necessary (see Section 36.3).

32.11.1 Obtaining Proxies

An application can obtain a proxy in a number of ways.

Stringified Proxies

The communicator operation `stringToProxy` creates a proxy from its stringified representation, as shown in the following C++ example:

```
Ice::ObjectPrx p = communicator->stringToProxy("ident:tcp -p 5000");
```

See Appendix E for a description of stringified proxies and Section 32.2 for more information on the `stringToProxy` operation.

Proxy Properties

Rather than hard-coding a stringified proxy as the previous example demonstrated, an application can gain more flexibility by externalizing the proxy in a configuration property. For example, we can define a property that contains our stringified proxy as follows:

```
MyApp.Proxy=ident:tcp -p 5000
```

We can use the communicator operation `propertyToProxy` to convert the property's value into a proxy, as shown below in Java:

```
Ice.ObjectPrx p = communicator.propertyToProxy("MyApp.Proxy");
```

As an added convenience, `propertyToProxy` allows you to define subordinate properties that configure the proxy's local settings. The properties below demonstrate this feature:

```
MyApp.Proxy=ident:tcp -p 5000
MyApp.Proxy.PreferSecure=1
MyApp.Proxy.EndpointSelection=Ordered
```

These additional properties simplify the task of customizing a proxy without the need to change the application's code. The properties shown above are equivalent to the following statements:

```
Ice.ObjectPrx p = communicator.stringToProxy("ident:tcp -p 5000");
p = p.ice_preferSecure(true);
p = p.ice_endpointSelection(Ice.EndpointSelectionType.Ordered);
```

The list of supported proxy properties is presented in Section D.9. Note that the communicator prints a warning by default if it does not recognize a subordinate property. You can disable this warning using the property `Ice.Warn.UnknownProperties` (see Section D.3).

Note that proxy properties can themselves have proxy properties. For example, the following sets the `PreferSecure` property on the default locator's router:

```
Ice.Default.Locator.Router.PreferSecure=1
```

Factory Methods

As we discuss in Section 32.11.2, proxy factory methods allow you to modify aspects of an existing proxy. Since proxies are immutable, factory methods always return a new proxy if the desired modification differs from the proxy's current configuration. Consider the following C# example:

```
Ice.ObjectPrx p = communicator.stringToProxy("...");
p = p.ice_oneway();
```

`ice_oneway` is considered a factory method because it returns a proxy configured to use oneway invocations. If the original proxy uses a different invocation mode, the return value of `ice_oneway` is a new proxy object.

The `checkedCast` and `uncheckedCast` methods can also be considered factory methods because they return new proxies that are narrowed to a particular Slice interface. A call to `checkedCast` or `uncheckedCast` typically follows the use of other factory methods, as shown below:

```
Ice.ObjectPrx p = communicator.stringToProxy("...");
Ice.LocatorPrx loc =
    Ice.LocatorPrxHelper.checkedCast(p.ice_secure(true));
```

Invocations

An application can also obtain a proxy as the result of an Ice invocation. Consider the following Slice definitions:

```
interface Account { ... };
interface Bank {
    Account* findAccount(string id);
};
```

Invoking the `findAccount` operation returns a proxy for an `Account` object. There is no need to use `checkedCast` or `uncheckedCast` on this proxy because it has already been narrowed to the `Account` interface. The C++ code below demonstrates how to invoke `findAccount`:

```
BankPrx bank = ...
AccountPrx acct = bank->findAccount(id);
```

Of course, the application must have already obtained a proxy for the bank object using one of the techniques shown above.

32.11.2 Proxy Methods

Although the core proxy functionality is supplied by a language-specific base class, we can describe the proxy methods in terms of Slice operations as shown below:

```
bool ice_isA(string id);
void ice_ping();
StringSeq ice_ids();
string ice_id();
int ice_getHash();
Communicator ice_getCommunicator();
string ice_toString();
Object* ice_identity(Identity id);
Identity ice_getIdentity();
Object* ice_adapterId(string id);
string ice_getAdapterId();
Object* ice_endpoints(EndpointSeq endpoints);
EndpointSeq ice_getEndpoints();
Object* ice_endpointSelection(EndpointSelectionType t);
EndpointSelectionType ice_getEndpointSelection();
Object* ice_context(Context ctx);
Context ice_getContext();
Object* ice_defaultContext();
Object* ice_facet(string facet);
string ice_getFacet();
Object* ice_twoway();
bool ice_isTwoway();
Object* ice_oneway();
```

```
bool ice_isOneway();
Object* ice_batchOneway();
bool ice_isBatchOneway();
Object* ice_datagram();
bool ice_isDatagram();
Object* ice_batchDatagram();
bool ice_isBatchDatagram();
Object* ice_secure(bool b);
bool ice_isSecure();
Object* ice_preferSecure(bool b);
bool ice_isPreferSecure();
Object* ice_compress(bool b);
Object* ice_timeout(int timeout);
Object* ice_router(Router* rtr);
Router* ice_getRouter();
Object* ice_locator(Locator* loc);
Locator* ice_getLocator();
Object* ice_locatorCacheTimeout(int seconds);
int ice_getLocatorCacheTimeout();
Object* ice_collocationOptimized(bool b);
bool ice_isCollocationOptimized();
Object* ice_connectionId(string id);
string ice_getConnectionId();
Connection ice_getConnection();
Connection ice_getCachedConnection();
Object* ice_connectionCached(bool b);
bool ice_isConnectionCached();
void ice_flushBatchRequests();
bool ice_invoke(string operation, OperationMode mode,
                ByteSeq inParams, out ByteSeq outParams);
```

These methods can be categorized as follows:

- Remote inspection: methods that return information about the remote object. These methods make remote invocations and therefore accept an optional trailing argument of type `Ice::Context` (see Section 32.12).
- Local inspection: methods that return information about the proxy's local configuration.
- Factory: methods that return new proxy instances configured with different features.
- Request processing: methods that flush batch requests and send "dynamic" Ice invocations.

Proxies are immutable, so factory methods allow an application to obtain a new proxy with the desired configuration. Factory methods essentially clone the original proxy and modify one or more features of the new proxy.

Many of the factory methods are not supported by fixed proxies. Attempting to invoke one of these methods causes the Ice run time to raise `FixedProxyException`. See page 12 for a description of fixed proxies and Section 36.7 for additional details.

The core proxy methods are explained in greater detail in Table 32.1.

Table 32.1. The semantics of core proxy methods.

Method	Description	Remote
`ice_isA`	Returns `true` if the remote object supports the type indicated by the `id` argument, otherwise `false`. This method can only be invoked on a twoway proxy.	Yes
`ice_ping`	Determines whether the remote object is reachable. Does not return a value.	Yes
`ice_ids`	Returns the type ids of the types supported by the remote object. The return value is an array of strings. This method can only be invoked on a twoway proxy.	Yes
`ice_id`	Returns the type id of the most-derived type supported by the remote object. This method can only be invoked on a twoway proxy.	Yes
`ice_getHash`	Returns a hash value for the proxy for C++. For other language mappings, use the built-in hash method.	No
`ice_getCommunicator`	Returns the communicator that was used to create this proxy.	No
`ice_toString`	Returns the string representation of the proxy.	No
`ice_identity`	Returns a new proxy having the given identity.	No

Table 32.1. The semantics of core proxy methods.

Method	Description	Remote
`ice_getIdentity`	Returns the identity of the Ice object represented by the proxy.	No
`ice_adapterId`	Returns a new proxy having the given adapter id.	No
`ice_getAdapterId`	Returns the proxy's adapter id, or an empty string if no adapter id is configured.	No
`ice_endpoints`	Returns a new proxy having the given endpoints.	No
`ice_getEndpoints`	Returns a sequence of `Endpoint` objects representing the proxy's endpoints. See Section 36.5.2 for more information.	No
`ice_endpointSelection`	Returns a new proxy having the given selection policy (random or ordered). See Section 36.3.1 for more information.	No
`ice_getEndpointSelection`	Returns the endpoint selection policy for the proxy.	No
`ice_context`	Returns a new proxy having the given request context. See Section 32.12 for more information on request contexts.	No
`ice_getContext`	Returns the request context associated with the proxy. See Section 32.12 for more information on request contexts.	No
`ice_facet`	Returns a new proxy having the given facet name. See Chapter 33 for more information on facets.	No
`ice_getFacet`	Returns the name of the facet associated with the proxy, or an empty string if no facet has been set. See Chapter 33 for more information on facets.	No
`ice_twoway`	Returns a new proxy for making twoway invocations.	No

Table 32.1. The semantics of core proxy methods.

Method	Description	Remote
`ice_isTwoway`	Returns `true` if the proxy uses twoway invocations, otherwise `false`.	No
`ice_oneway`	Returns a new proxy for making oneway invocations (see Section 32.14).	No
`ice_isOneway`	Returns `true` if the proxy uses oneway invocations, otherwise `false`.	No
`ice_batchOneway`	Returns a new proxy for making batch oneway invocations (see Section 32.16).	No
`ice_isBatchOneway`	Returns `true` if the proxy uses batch oneway invocations, otherwise `false`.	No
`ice_datagram`	Returns a new proxy for making datagram invocations (see Section 32.15).	No
`ice_isDatagram`	Returns `true` if the proxy uses datagram invocations, otherwise `false`.	No
`ice_batchDatagram`	Returns a new proxy for making batch datagram invocations (see Section 32.16).	No
`ice_isBatchDatagram`	Returns `true` if the proxy uses batch datagram invocations, otherwise `false`.	No
`ice_secure`	Returns a new proxy whose endpoints may be filtered depending on the boolean argument. If `true`, only endpoints using secure transports are allowed, otherwise all endpoints are allowed.	No
`ice_isSecure`	Returns `true` if the proxy uses only secure endpoints, otherwise `false`.	No

Table 32.1. The semantics of core proxy methods.

Method	Description	Remote
`ice_preferSecure`	Returns a new proxy whose endpoints are filtered depending on the boolean argument. If `true`, endpoints using secure transports are given precedence over endpoints using non-secure transports. If `false`, the default behavior gives precedence to endpoints using non-secure transports.	No
`ice_isPreferSecure`	Returns `true` if the proxy prefers secure endpoints, otherwise `false`.	No
`ice_compress`	Returns a new proxy whose protocol compression capability is determined by the boolean argument. If `true`, the proxy uses protocol compression if it is supported by the endpoint. If `false`, protocol compression is never used.	No
`ice_timeout`	Returns a new proxy with the given timeout value in milliseconds. A value of `-1` disables timeouts. See Section 32.13 for more information on timeouts.	No
`ice_router`	Returns a new proxy configured with the given router proxy. See Chapter 42 for more information on routers.	No
`ice_getRouter`	Returns the router that is configured for the proxy (null if no router is configured).	No
`ice_locator`	Returns a new proxy with the specified locator. See Chapter 38 for more information on locators.	No
`ice_getLocator`	Returns the locator that is configured for the proxy (null if no locator is configured).	No

Table 32.1. The semantics of core proxy methods.

Method	Description	Remote
`ice_locatorCacheTimeout`	Returns a new proxy with the specified locator cache timeout. When binding a proxy to an endpoint, the run time caches the proxy returned by the locator and uses the cached proxy while the cached proxy has been in the cache for less than the timeout. Proxies older than the timeout cause the run time to rebind via the locator. A value of 0 disables caching entirely, and a value of -1 means that cached proxies never expire. The default value is -1.	No
`ice_getLocatorCacheTimeout`	Returns the locator cache timeout value in seconds.	No
`ice_collocationOptimized`	Returns a new proxy configured for collocation optimization. If `true`, collocated optimizations are enabled. The default value is `true`.	No
`ice_isCollocationOptimized`	Returns `true` if the proxy uses collocation optimization, otherwise `false`.	No
`ice_connectionId`	Returns a new proxy having the given connection identifier. See Section 36.3.3 for more information.	No
`ice_getConnectionId`	Returns the connection id, or an empty string if no connection id has been configured.	No
`ice_getConnection`	Returns an object representing the connection used by the proxy. If the proxy is not currently associated with a connection, the Ice run time attempts to establish a connection first. See Section 36.5 for more information.	No

Table 32.1. The semantics of core proxy methods.

Method	Description	Remote
`ice_getCachedConnection`	Returns an object representing the connection used by the proxy, or null if the proxy is not currently associated with a connection. See Section 36.5 for more information.	No
`ice_connectionCached`	Enables or disables connection caching for the proxy. See Section 36.3.4 for more information	No
`ice_isConnectionCached`	Returns `true` if the proxy uses connection caching, otherwise `false`.	No
`ice_flushBatchRequests` `begin_ice_flushBatchRequests`	Sends a batch of operation invocations synchronously or asynchronously (see Section 32.16).	Yes
`ice_invoke` `begin_ice_invoke`	Allows dynamic invocation of an operation without the need for compiled Slice definitions. Requests can be sent synchronously (see Section 35.3.1) or asynchronously (see Section 35.4).	Yes

32.11.3 Endpoints

Proxy endpoints are the client-side equivalent of object adapter endpoints (see Section 32.4.6). A proxy endpoint identifies the protocol information used to contact a remote object, as shown in the following example:

```
tcp -h www.zeroc.com -p 10000
```

This endpoint states that an object is reachable via TCP on the host `www.zeroc.com` and the port `10000`.

A proxy must have, or be able to obtain, at least one endpoint in order to be useful. As defined in Section 2.2.2, a *direct proxy* contains one or more endpoints:

```
MyObject:tcp -h www.zeroc.com -p 10000:ssl -h www.zeroc.com -p 10001
```

In this example the object with the identity `MyObject` is available at two separate endpoints, one using TCP and the other using SSL.

If a direct proxy does not contain the **`-h`** option (that is, no host is specified), the Ice run time uses the value of the `Ice.Default.Host` property (see Appendix D). If `Ice.Default.Host` is not defined, the localhost interface is used.

An *indirect proxy* uses a locator (see Chapter 38) to retrieve the endpoints dynamically. One style of indirect proxy contains an adapter identifier:

```
MyObject @ MyAdapter
```

When this proxy requires the endpoints associated with `MyAdapter`, it requests them from the locator.

32.11.4 Endpoint Filtering

A proxy's configuration determines how its endpoints are used. For example, a proxy configured for secure communication will only use endpoints having a secure protocol, such as SSL.

The factory functions described in Table 32.2 allow applications to manipulate endpoints indirectly. Calling one of these functions returns a new proxy whose endpoints are used in accordance with the proxy's configuration.

Table 32.2. Proxy factory functions and their effects on endpoints.

Option	Description
`ice_secure`	Selects only endpoints using a secure protocol (e.g., SSL).
`ice_datagram`	Selects only endpoints using a datagram protocol (e.g., UDP).
`ice_batchDatagram`	Selects only endpoints using a datagram protocol (e.g., UDP).
`ice_twoway`	Selects only endpoints capable of making twoway invocations (e.g., TCP, SSL). For example, this disables datagram endpoints.

Table 32.2. Proxy factory functions and their effects on endpoints.

Option	Description
`ice_oneway`	Selects only endpoints capable of making reliable oneway invocations (e.g., TCP, SSL). For example, this disables datagram endpoints.
`ice_batchOneway`	Selects only endpoints capable of making reliable oneway batch invocations (e.g., TCP, SSL). For example, this disables datagram endpoints.

Upon return, the set of endpoints in the new proxy is unchanged from the old one. However, the new proxy's configuration drives a filtering process that the Ice run time performs during connection establishment, as described in Section 36.3.1.

The factory functions do not raise an exception if they produce a proxy with no viable endpoints. For example, the C++ statement below creates such a proxy:

```
proxy = comm->stringToProxy("id:tcp -p 10000")->ice_datagram();
```

It is always possible that a proxy could become viable after additional factory functions are invoked, therefore the Ice run time does not raise an exception until connection establishment is attempted. At that point, the application can expect to receive `NoEndpointException` if the filtering process eliminates all endpoints.

An application can also create a proxy with a specific set of endpoints using the `ice_endpoints` factory function, whose only argument is a sequence of `Ice::Endpoint` objects. At present, an application is not able to create new instances of `Ice::Endpoint`, but rather can only incorporate instances obtained by calling `ice_getEndpoints` on a proxy. Note that `ice_getEndpoints` may return an empty sequence if the proxy has no endpoints, as is the case with an indirect proxy.

32.11.5 Defaults and Overrides

It is important to understand how proxies are influenced by Ice configuration properties and settings. The relevant properties can be classified into two categories: defaults and overrides.

Default Properties

Default properties affect proxies created as the result of an Ice invocation, or by calls to `stringToProxy` or `propertyToProxy`. These properties do not influence proxies created by factory methods.

For example, suppose we define the following default property:

```
Ice.Default.PreferSecure=1
```

We can verify that the property has the desired affect using the following C++ code:

```
Ice::ObjectPrx p = communicator->stringToProxy(...);
assert(p->ice_isPreferSecure());
```

Furthermore, we can verify that the property does not affect proxies returned by factory methods:

```
Ice::ObjectPrx p2 = p->ice_preferSecure(false);
assert(!p2->ice_isPreferSecure());
Ice::ObjectPrx p3 = p2->ice_oneway();
assert(!p3->ice_isPreferSecure());
```

The default properties are described in Section D.8.

Override Properties

Defining an override property causes the Ice run time to ignore any equivalent proxy setting and use the override property value instead. For example, consider the following property definition:

```
Ice.Override.Secure=1
```

This property instructs the Ice run time to use only secure endpoints, producing the same semantics as calling `ice_secure(true)` on every proxy. However, the property does not alter the settings of an existing proxy, but rather directs the Ice run time to use secure endpoints regardless of the proxy's security setting. We can verify that this is the case using the following C++ code:

```
Ice::ObjectPrx p = communicator->stringToProxy(...);
p = p->ice_secure(false);
assert(!p->ice_isSecure()); // The security setting is retained.
```

The override properties are described in Section D.8.

32.12 The `Ice::Context` Parameter

Methods on a proxy are overloaded with a trailing parameter of type `const Ice::Context &` (C++), `java.util.Map` (Java), or `Dictionary<string, string>` (C#). The Slice definition of this parameter is as follows:

```
module Ice {
    local dictionary<string, string> Context;
};
```

As you can see, a context is a dictionary that maps strings to strings or, conceptually, a context is a collection of name–value pairs. The contents of this dictionary (if any) are implicitly marshaled with every request to the server, that is, if the client populates a context with a number of name–value pairs and uses that context for an invocation, the name–value pairs that are sent by the client are available to the server.

On the server side, the operation implementation can access the received `Context` via the `ctx` member of the `Ice::Current` parameter (see Section 32.6) and extract the name–value pairs that were sent by the client.

Context names beginning with an underscore () are reserved for use by Ice.

32.12.1 Passing a Context Explicitly

Contexts provide a means of sending an unlimited number of parameters from client to server without having to mention these parameters in the signature of an operation. For example, consider the following definition:

```
struct Address {
    // ...
};

interface Person {
    string setAddress(Address a);
    // ...
};
```

Assuming that the client has a proxy to a `Person` object, it could do something along the following lines:

```
PersonPrx p = ...;
Address a = ...;

Ice::Context ctx;
ctx["write policy"] = "immediate";

p->setAddress(a, ctx);
```

In Java, the same code would looks as follows:

```
PersonPrx p = ...;
Address a = ...;

java.util.Map ctx = new java.util.HashMap();
ctx.put("write policy", "immediate");

p.setAddress(a, ctx);
```

In C#, the code is almost identical, except that the context dictionary is type safe:

```
using System.Collections.Generic;

PersonPrx p = ...;
Address a = ...;

Dictionary<string, string> ctx = new Dictionary<string, string>();
ctx["write policy"] = "immediate";

p.setAddress(a, ctx);
```

On the server side, we can extract the policy value set by the client to influence how the implementation of `setAddress` works. A C++ implementation might look like:

```
void
PersonI::setAddress(const Address& a, const Ice::Current& c)
{
    Ice::Context::const_iterator i = c.ctx.find("write policy");
    if (i != c.ctx.end() && i->second == "immediate") {

        // Update the address details and write through to the
        // data base immediately...

    } else {
```

```
        // Write policy was not set (or had a bad value), use
        // some other database write strategy.
    }
}
```

For this example, the server examines the value of the context with the key `"write policy"` and, if that value is `"immediate"`, writes the update sent by the client straight away; if the write policy is not set or contains a value that is not recognized, the server presumably applies a more lenient write policy (such as caching the update in memory and writing it later). The Java version of the operation implementation is essentially identical, so we do not show it here.

32.12.2 Passing a Per-Proxy Context

Instead of passing a context explicitly with an invocation, you can also use a *per-proxy* context. Per-proxy contexts allow you to set a context on a particular proxy once and, thereafter, whenever you use that proxy to invoke an operation, the previously-set context is sent with each invocation. The proxy base class provides a member function, `ice_context`, to do this:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx
                ice_context(const Ice::Context&) const;
            // ...
        };
    }
}
```

For Java, the corresponding function is:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_context(java.util.Map newContext);
    // ...
}
```

For C#, the corresponding method is:

```
namespace Ice
{
    public interface ObjectPrx
    {
```

```
        ObjectPrx ice_context(Dictionary<string, string>
                                    newContext);
        // ...
    }
}
```

ice_context creates a new proxy that stores the passed context. Note that the return type of ice_context is ObjectPrx. Therefore, before you can use the newly-created proxy, you must down-cast it to the correct type. For example, in C++:

```
Ice::Context ctx;
ctx["write policy"] = "immediate";

PersonPrx p1 = ...;
PersonPrx p2 = PersonPrx::uncheckedCast(p1->ice_context(ctx));

Address a = ...;

p1->setAddress(a);       // Sends no context

p2->setAddress(a);       // Sends ctx implicitly

Ice::Context ctx2;
ctx2["write policy"] = "delayed";

p2->setAddress(a, ctx2); // Sends ctx2
```

As the example illustrates, once we have created the p2 proxy, any invocation via p2 implicitly sends the previously-set context. The final line of the example illustrates that it is possible to explicitly send a context for an invocation even if the proxy has an implicit context—an explicit context always overrides any implicit context.

Note that, once you have set a per-proxy context, that context becomes immutable: if you change the context you have passed to ice_context later, that does not affect the per-proxy context of any proxies you previously created with that context because each proxy on which you set a per-proxy context makes a copy of the dictionary and stores that copy.

32.12.3 Retrieving the Per-Proxy Context

You can retrieve the per-proxy context by calling `ice_getContext` on the proxy. If a proxy has no implicit context, the returned dictionary is empty. For C++, the signature is:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::Context ice_getContext() const;
            // ...
        };
    }
}
```

For Java, the signature is:

```
package Ice;

public interface ObjectPrx {
    java.util.Map ice_getContext();
    // ...
}
```

For C#, the signature is:

```
namespace Ice
{
    public interface ObjectPrx
    {
        Dictionary<string, string> ice_getContext();
    }
}
```

32.12.4 Implicit Contexts

In addition to the explicit and the implicit per-proxy contexts we described in the preceding sections, you can also establish an implicit context on a communicator. This implicit context is sent with all invocations made via proxies created by that communicator, provided that you do not supply an explicit context with the call.

Access to this implicit context is provided by the `Communicator` interface:

```
module Ice {
    local interface Communicator
    {
        ImplicitContext getImplicitContext();
```

```
        // ...
    };
};
```

getImplicitContext returns the implicit context. If a communicator has no implicit context, the operation returns a null proxy.

You can manipulate the contents of the implicit context via the ImplicitContext interface:

```
local interface ImplicitContext
{
    Context getContext();
    void    setContext(Context newContext);

    string get(string key);
    string put(string key, string value);
    string remove(string key);
    bool   containsKey(string key);
};
```

The getContext operation returns the currently-set context dictionary. The setContext operation replaces the currently-set context in its entirety.

The remaining operations allow you to manipulate specific entries:

- get

  This operation returns the value associated with key. If key was not previously set, the operation returns the empty string.

- put

  This operation adds the key–value pair specified by key and value. It returns the previous value associated with key; if no value was previously associated with key, it returns the empty string. It is legal to add the empty string as a value.

- remove

  This operation removes the key–value pair specified by key. It returns the previously-set value (or the empty string if key was not previously set).

- containsKey

  This operation returns true if key is currently set and false, otherwise. You can use this operation to distinguish between a key–value pair that was explicitly added with an empty string as a value, and a key–value pair that was never added at all.

Scope of the Implicit Context

You establish the implicit context on a communicator by setting a property, `Ice.ImplicitContext`. This property controls whether a communicator has an implicit context and, if so, at what scope the context applies. The property can be set to the following values:

- `None`

  With this setting (or if `Ice.ImplicitContext` is not set at all), the communicator has no implicit context, and `getImplicitContext` returns a null proxy.
- `Shared`

  The communicator has a single implicit context that is shared by all threads. Access to the context via its `ImplicitContext` interface is interlocked, so different threads can concurrently manipulate the context without risking data corruption or reading stale values.
- `PerThread`

  The communicator maintains a separate implicit context for each thread. This allows you to propagate contexts that depend on the sending thread (for example, to send per-thread transaction IDs).

32.12.5 Interactions of Explicit, Per-Proxy, and Implicit Contexts

If you use explicit, per-proxy, and implicit contexts, it is important to be aware of their interactions:

- If you send an explicit context with an invocation, *only* that context is sent with the call, regardless of whether the proxy has a per-proxy context and whether the communicator has an implicit context.
- If you send an invocation via a proxy that has a per-proxy context, and the communicator also has an implicit context, the contents of the per-proxy and implicit context dictionaries are combined, so the *combination* of context entries of both contexts is transmitted to the server. If the per-proxy context and the implicit context contain the same key, but with different values, the *per-proxy* value takes precedence.

32.12.6 Context Use Cases

The purpose of `Ice::Context` is to permit services to be added to Ice that require some contextual information with every request. Contextual information can be used by services such as a transaction service (to provide the context of a currently

established transaction) or a security service (to provide an authorization token to the server). IceStorm (see Chapter 44) uses the context to provide an optional `cost` parameter to the service that influences how the service propagates messages to down-stream subscribers.

In general, services that require such contextual information can be implemented much more elegantly using contexts because this hides explicit Slice parameters that would otherwise have to be supplied by the application programmer with every call.

In addition, contexts, because they are optional, permit a single Slice definition to apply to implementations that use the context, as well as to implementations that do not use it. In this way, to add transactional semantics to an existing service, you do not need to modify the Slice definitions to add an extra parameter to each operation. (Adding an extra parameter would not only be inconvenient for clients, but would also split the type system into two halves: without contexts, we would need different Slice definitions for transactional and non-transactional implementations of (conceptually) a single service.)

Finally, per-proxy contexts permit context information to be passed by through intermediate parts of your program without cooperation of those intermediate parts. For example, suppose you set a per-proxy context on a proxy and then pass that proxy to another function. When that function uses the proxy to invoke an operation, the per-proxy context will still be sent. In other words, per-proxy contexts allow you to transparently propagate information via intermediaries that are ignorant of the presence of any context.

Keep in mind though that this works only within a single process. If you stringify a proxy or transmit it as a parameter over the wire, the per-proxy context is *not* preserved. (Ice does not write the per-proxy context into stringified proxies and does not marshal the per-proxy context when a proxy is marshaled.)

32.12.7 A Word of Warning

Contexts are a powerful mechanism for transparent propagation of context information, *if used correctly.* In particular, you may be tempted to use contexts as a means of versioning an application as it evolves over time. For example, version 2 of your application may accept two parameters on an operation that, in version 1, used to accept only a single parameter. Using contexts, you could supply the second parameter as a name–value pair to the server and avoid changing the Slice definition of the operation in order to maintain backward compatibility.

We *strongly* urge you to resist any temptation to use contexts in this manner. The strategy is fraught with problems:

- Missing context

  There is nothing that would compel a client to actually send a context when the server expects to receive a context: if a client forgets to send a context, the server, somehow, has to make do without it (or throw an exception).

- Missing or incorrect keys

  Even if the client does send a context, there is no guarantee that it has set the correct key. (For example, a simple spelling error can cause the client to send a value with the wrong key.)

- Incorrect values

  The value of a context is a string, but the application data that is to be sent might be a number, or it might be something more complex, such as a structure with several members. This forces you to encode the value into a string and decode the value again on the server side. Such parsing is tedious and error prone, and far less efficient than sending strongly-typed parameters. In addition, the server has to deal with string values that fail to decode correctly (for example, because of an encoding error made by the client).

None of the preceding problems can arise if you use proper Slice parameters: parameters cannot be accidentally omitted and they are strongly typed, making it much less likely for the client to accidentally send a meaningless value.

If you are concerned about how to evolve an application over time without breaking backward compatibility, Ice facets are better suited to this task (see Chapter 33). Contexts are meant to be used to transmit simple tokens (such as a transaction identifier) for services that cannot be reasonably implemented without them; you should restrict your use of contexts to that purpose and resist any temptation to use contexts for any other purpose.

Finally, be aware that, if a request is routed via one or more Ice routers, contexts may be dropped by intermediate routers if they consider them illegal. This means that, in general, you cannot rely on an arbitrary context value that is created by an application to actually still be present when a request arrives at the server—only those context values that are known to routers and that are considered legitimate are passed on. It follows that you should not abuse contexts to pass things that really should be passed as parameters.

32.13 Connection Timeouts

A synchronous remote invocation does not complete on the client side until the server has finished processing it. Occasionally, it is useful to be able to force an invocation to terminate after some time, even if it has not completed. Proxies provide the `ice_timeout` operation for this purpose:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx ice_timeout(Ice::Int t) const;
            // ...
        };
    }
}
```

For Java (and, analogously, for C#), the corresponding method is:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_timeout(int t);
    // ...
}
```

The `ice_timeout` operation creates a proxy with a timeout from an existing proxy. For example:

```
Filesystem::FilePrx myFile = ...;
FileSystem::FilePrx timeoutFile
    = FileSystem::FilePrx::uncheckedCast(
        myFile->ice_timeout(5000));

try {
    Lines text = timeoutFile->read();   // Read with timeout
} catch(const Ice::TimeoutException&) {
    cerr << "invocation timed out" << endl;
}

Lines text = myFile->read();            // Read without timeout
```

The parameter to `ice_timeout` determines the timeout value in milliseconds. A value of –1 indicates no timeout. In the preceding example, the timeout is set to five seconds; if an invocation of `read` via the `timeoutFile` proxy does not complete within five seconds, the operation terminates with an `Ice::Timeou-`

tException. On the other hand, invocations via the myFile proxy are unaffected by the timeout, that is, ice_timeout sets the timeout on a per-proxy basis.

The timeout value set on a proxy affects all networking operations: reading and writing of data as well as opening and closing of connections. If any of these operations does not complete within the timeout, the client receives an exception. Note that, if the Ice run time encounters a recoverable error condition and transparently retries an invocation, this means that the timeout applies separately to each attempt. Similarly, if a large amount of data is sent with an operation invocation in several write system calls, the timeout applies to each write, not to the invocation overall.

Timeouts that expire during reading or writing of data are indicated by a TimeoutException. For opening and closing of connections, the Ice run time reserves separate exceptions:

- ConnectTimeoutException

  This exception indicates that a connection could not be established within the specified time.

- CloseTimeoutException

  This exception indicates that a connection could not be closed within the specified time.

An application normally configures a proxy's timeout using the ice_timeout method. However, a proxy that originated from a string may already have a timeout specified, as shown in the following example:

```
// C++
string s = "ident:tcp -h somehost -t 5000:ssl -h somehost -t 5000";
```

In this case, both the TCP and SSL endpoints define a timeout of five seconds. When the Ice run time establishes a connection using one of these endpoints, it uses the endpoint's timeout unless one was specified explicitly via ice_timeout.

The Ice run time also supports two configuration properties that override the timeouts of every proxy regardless of the settings established via ice_timeout or defined in stringified proxies:

- Ice.Override.ConnectTimeout

  This property defines a timeout that is only used for connection establishment. If not defined, its default value is –1 (no timeout). If a proxy has multiple endpoints, the timeout applies to each endpoint separately.

- `Ice.Override.Timeout`

  This property defines the timeout for invocations. If no value is defined for `Ice.Override.ConnectTimeout`, the value of `Ice.Override.Timeout` is also used as the timeout for connection establishment. If not defined, the default value is –1 (no timeout).

Note that timeouts are "soft" timeouts, in the sense that they are not precise, real-time timeouts. (The precision is limited by the capabilities of the underlying operating system.) You should also be aware that timeouts are considered fatal error conditions by the Ice run time and result in connection closure on the client side. Furthermore, any other requests pending on the same connection also fail with an exception. Timeouts are meant to be used to prevent a client from blocking indefinitely in case something has gone wrong with the server; they are not meant as a mechanism to routinely abort requests that take longer than intended.

32.14 Oneway Invocations

As mentioned in Chapter 2, the Ice run time supports *oneway* invocations. A oneway invocation is sent on the client side by writing the request to the client's local transport buffers; the invocation completes and returns control to the application code as soon as it has been accepted by the local transport. Of course, this means that a oneway invocation is unreliable: it may never be sent (for example, because of a network failure) or it may not be accepted in the server (for example, because the target object does not exist).

This is an issue in particular if you use *active connection management* (see Section 36.4): if a server closes a connection at the wrong moment, it is possible for the client to lose already-buffered oneway requests. We therefore recommend that you disable active connection management for the server side if clients use oneway (or batched oneway—see Section 32.16) requests. In addition, if clients use oneway requests and your application initiates server shutdown, it is the responsibility of your application to ensure either that it can cope with the potential loss of buffered oneway requests, or that it does not shut down the server at the wrong moment (while clients still have oneway requests that are buffered, but not yet sent).

If anything goes wrong with a oneway request, the client-side application code does not receive any notification of the failure; the only errors that are reported to the client are local errors that occur on the client side during call invocation (such as failure to establish a connection, for example).

As a consequence of oneway invocation, if you call `ice_ping` on a oneway proxy, successful completion does *not* indicate that the target object exists and could successfully be contacted—you will receive an exception only if something goes wrong on the client side, but not if something goes wrong on the server side. Therefore, if you want to use `ice_ping` with a oneway proxy and be certain that the target object exists and can successfully be contacted, you must first convert the oneway proxy into a twoway proxy. For example, in C++:

```
SomeObjectPrx onewayPrx = ...; // Get a oneway proxy

try {
    onewayPrx->ice_twoway()->ice_ping();
} catch(const Ice::Exception&)
    cerr << "object not reachable" << endl;
}
```

Oneway invocations are received and processed on the server side like any other incoming request. If necessary, a server can distinguish a oneway invocation by examining the `requestId` member of `Ice::Current`: a non-zero value denotes a twoway request, whereas a value of zero indicates a oneway request.

Oneway invocations do not incur any return traffic from the server to the client: the server never sends a reply message in response to a oneway invocation (see Chapter 37). This means that oneway invocations can result in large efficiency gains, especially for large numbers of small messages, because the client does not have to wait for the reply to each message to arrive before it can send the next message.

In order to be able to invoke an operation as oneway, two conditions must be met:

- The operation must have a `void` return type, must not have any out-parameters, and must not have an exception specification.

  This requirement reflects the fact that the server does not send a reply for a oneway invocation to the client: without such a reply, there is no way to return any values or exceptions to the client.

  If you attempt to invoke an operation that returns values to the client as a oneway operation, the Ice run time throws a `TwowayOnlyException`.

- The proxy on which the operation is invoked must support a stream-oriented transport (such as TCP or SSL).

  Oneway invocations require a stream-oriented transport. (To get something like a oneway invocation for datagram transports, you need to use a datagram invocation—see Section 32.15.)

  If you attempt to create a oneway proxy for an object that does not offer a stream-oriented transport, the Ice run time throws a `NoEndpointException`.

Despite their theoretical unreliablity, in practice, oneway invocations are reliable (but not infallible [19]): they are sent via a stream-oriented transport, so they cannot get lost except when the connection is shutting down (see Section 36.6.2) or fails entirely. In particular, the transport uses its usual flow control, so the client cannot overrun the server with messages. On the client-side, the Ice run time will block if the client's transport buffers fill up, so the client-side application code cannot overrun its local transport.

Consequently, oneway invocations normally do not block the client-side application code and return immediately, provided that the client does not consistently generate messages faster than the server can process them. If the rate at which the client invokes operations exceeds the rate at which the server can process them, the client-side application code will eventually block in an operation invocation until sufficient room is available in the client's transport buffers to accept the invocation. If your application requires that oneway requests never block the calling thread, you can use asynchronous oneway invocations instead.

Regardless of whether the client exceeds the rate at which the server can process incoming oneway invocations, the execution of oneway invocations in the server proceeds asynchronously: the client's invocation completes before the message even arrives at the server.

One thing you need to keep in mind about oneway invocations is that they may appear to be reordered in the server: because oneway invocations are sent via a stream-oriented transport, they are guaranteed to be received in the order in which they were sent. However, the server's thread pool may dispatch each invocation in its own thread; because threads are scheduled preemptively, this may cause an invocation sent later by the client to be dispatched and executed before an invocation that was sent earlier. If oneway requests must be dispatched in order, you can use one of the serialization techniques described in Section 32.10.4.

For these reasons, oneway invocations are usually best suited to simple updates that are otherwise stateless (that is, do not depend on the surrounding context or the state established by previous invocations). Refer to Section 32.10 for more information on threading.

Creating Oneway Proxies

Ice selects between twoway, oneway, and datagram (see Section 32.15) invocations via the proxy that is used to invoke the operation. By default, all proxies are created as twoway proxies. To invoke an operation as oneway, you must create a separate proxy for oneway dispatch from a twoway proxy.

For C++, all proxies are derived from a common `IceProxy::Ice::Object` class (seeSection 6.11.1). The proxy base class contains a method to create a oneway proxy, called `ice_oneway`:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx ice_oneway() const;
            // ...
        };
    }
}
```

For Java and C#, proxies are derived from the `Ice.ObjectPrx` interface (see Section 10.11.2) and the definition of `ice_oneway` is:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_oneway();
    // ...
}
```

We can call `ice_oneway` to create a oneway proxy and then use the proxy to invoke an operation as follows. (We show the C++ version here—the Java and C# versions are analogous.)

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Get a oneway proxy.
//
Ice::ObjectPrx oneway;
try {
    oneway = o->ice_oneway();
} catch (const Ice::NoEndpointException&) {
    cerr << "No endpoint for oneway invocations" << endl;
}

// Down-cast to actual type.
```

```
//
PersonPrx onewayPerson = PersonPrx::uncheckedCast(oneway);

// Invoke an operation as oneway.
//
try {
    onewayPerson->someOp();
} catch (const Ice::TwowayOnlyException&) {
    cerr << "someOp() is not oneway" << endl;
}
```

Note that we use an `uncheckedCast` to down-cast the proxy from `ObjectPrx` to `PersonPrx`: for a oneway proxy, we cannot use a `checkedCast` because a `checkedCast` requires a reply from the server but, of course, a oneway proxy does not permit that reply. If instead you want to use a safe down-cast, you can first down-cast the twoway proxy to the actual object type and then obtain the oneway proxy:

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Safe down-cast to actual type.
//
PersonPrx person = PersonPrx::checkedCast(o);

if (person) {
    // Get a oneway proxy.
    //
    PersonPrx onewayPerson;
    try {
        onewayPerson
            = PersonPrx::uncheckedCast(person->ice_oneway());
    } catch (const Ice::NoEndpointException&) {
        cerr << "No endpoint for oneway invocations" << endl;
    }

    // Invoke an operation as oneway.
    //
    try {
        onewayPerson->someOp();
    } catch (const Ice::TwowayOnlyException&) {
        cerr << "someOp() is not oneway" << endl;
    }
}
```

Note that, while the second version of this code is somewhat safer (because it uses a safe down-cast), it is also slower (because the safe down-cast incurs the cost of an additional twoway message).

32.15 Datagram Invocations

Datagram invocations are the equivalent of oneway invocations for datagram transports. As for oneway invocations, datagram invocations can be sent only for operations that have a `void` return type and do not have out-parameters or an exception specification. (Attempts to use a datagram invocation with an operation that does not meet these criteria result in a `TwowayOnlyException`.) In addition, datagram invocations can only be used if the proxy's endpoints include at least one UDP transport; otherwise, the Ice run time throws a `NoEndpointException`.

The semantics of datagram invocations are similar to oneway invocations: no return traffic flows from the server to the client and proceed asynchronously with respect to the client; a datagram invocation completes as soon as the client's transport has accepted the invocation into its buffers. However, datagram invocations differ in one respect from oneway invocations in that datagram invocations optionally support multicast semantics. Furthermore, datagram invocations have additional error semantics:

- Individual invocations may be lost or received out of order.

  On the wire, datagram invocations are sent as true datagrams, that is, individual datagrams may be lost, or arrive at the server out of order. As a result, not only may operations be dispatched out of order, an individual invocation out of a series of invocations may be lost. (This cannot happen for oneway invocations because, if a connection fails, *all* invocations are lost once the connection breaks down.)

- UDP packets may be duplicated by the transport.

  Because of the nature of UDP routing, it is possible for datagrams to arrive in duplicate at the server. This means that, for datagram invocations, Ice does *not* guarantee at-most-once semantics (see page 14): if UDP datagrams are duplicated, the same invocation may be dispatched more than once in the server.

- UDP packets are limited in size.

  The maximum size of an IP datagram is 65,535 bytes. Of that, the IP header consumes 20 bytes, and the UDP header consumes 8 bytes, leaving 65,507 bytes as the maximum payload. If the marshaled form of an invoca-

tion, including the Ice request header (see Chapter 37) exceeds that size, the invocation is lost. (Exceeding the size limit for a UDP datagram is indicated to the application by a `DatagramLimitException`.)

Because of their unreliable nature, datagram invocations are best suited to simple update messages that are otherwise stateless. In addition, due to the high probability of loss of datagram invocations over wide area networks, you should restrict use of datagram invocations to local area networks, where they are less likely to be lost. (Of course, regardless of the probability of loss, you must design your application such that it can tolerate lost or duplicated messages.)

Creating Datagram Proxies

To create a datagram proxy, you must call `ice_datagram` on the proxy, for example:

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Get a datagram proxy.
//
Ice::ObjectPrx datagram;
try {
    datagram = o->ice_datagram();
} catch (const Ice::NoEndpointException&) {
    cerr << "No endpoint for datagram invocations" << endl;
}

// Down-cast to actual type.
//
PersonPrx datagramPerson = PersonPrx::uncheckedCast(datagram);

// Invoke an operation as a datagram.
//
try {
    datagramPerson->someOp();
} catch (const Ice::TwowayOnlyException&) {
    cerr << "someOp() is not oneway" << endl;
}
```

As for the oneway example in Section 32.14, you can choose to first do a safe down-cast to the actual type of interface and then obtain the datagram proxy, rather than relying on an unsafe down-cast, as shown here. However, doing so may be disadvantageous for two reasons:

- Safe down-casts are sent via a stream-oriented transport. This means that using a safe down-cast will result in opening a connection for the sole purpose of verifying that the target object has the correct type. This is expensive if all the other traffic to the object is sent via datagrams.
- If the proxy does not offer a stream-oriented transport, the `checkedCast` fails with a `NoEndpointException`, so you can use this approach only for proxies that offer both a UDP endpoint and a TCP/IP and/or SSL endpoint.

32.16 Batched Invocations

Oneway and datagram invocations are normally sent as a separate message, that is, the Ice run time sends the oneway or datagram invocation to the server immediately, as soon as the client makes the call. If a client sends a number of oneway or datagram invocations in succession, the client-side run time traps into the OS kernel for each message, which is expensive. In addition, each message is sent with its own message header (see Chapter 37), that is, for *n* messages, the bandwidth for *n* message headers is consumed. In situations where a client sends a number of oneway or datagram invocations, the additional overhead can be considerable.

To avoid the overhead of sending many small messages, you can send oneway and datagram invocations in a batch: instead of being sent as a separate message, a batch invocation is placed into a client-side buffer by the Ice run time. Successive batch invocations are added to the buffer and accumulated on the client side until they are flushed, either explicitly by the client or automatically by the Ice run time.

32.16.1 Proxy Methods

The relevant APIs are part of the proxy interface:

```
// C++
namespace IceProxy {
namespace Ice {

class Object : /* ... */ {
public:
    Ice::ObjectPrx ice_batchOneway() const;
    Ice::ObjectPrx ice_batchDatagram() const;
    void ice_flushBatchRequests();
```

```
    // ...
};

}
}
```

The `ice_batchOneway` and `ice_batchDatagram` methods convert a proxy to a batch proxy. Once you obtain a batch proxy, messages sent via that proxy are buffered in the client-side run time instead of being sent immediately. Once the client has invoked one or more operations on batch proxies, it can call `ice_flushBatchRequests` to explicitly flush the batched invocations. This causes the batched messages to be sent "in bulk", preceded by a single message header. On the server side, batched messages are dispatched by a single thread, in the order in which they were written into the batch. This means that messages from a single batch cannot appear to be reordered in the server. Moreover, either all messages in a batch are delivered or none of them. (This is true even for batched datagrams.)

Asynchronous versions of `ice_flushBatchRequests` are also available; see the relevant language mapping for more information.

32.16.2 Automatic Flushing

The default behavior of the Ice run time, as governed by the configuration property `Ice.BatchAutoFlush` (see Appendix D), automatically flushes batched invocations as soon as a batched request causes the accumulated message to exceed the maximum allowable size. When this occurs, the Ice run time immediately flushes the existing batch of requests and begins a new batch with this latest request as its first element.

For batched oneway invocations, the maximum message size is established by the property `Ice.MessageSizeMax`, which defaults to 1MB. In the case of batched datagram invocations, the maximum message size is the smaller of the system's maximum size for datagram packets and the value of `Ice.MessageSizeMax`.

A client that sends batch requests cannot determine the size of the message that the Ice run time is accumulating for it; automatic flushing is enabled by default as a convenience for clients that unknowingly exceed the maximum message size. A client that requires more deterministic behavior should flush batched requests explicitly at regular intervals.

32.16.3 Flushing Communicators and Connections

The `Communicator` and `Connection` interfaces support synchronous and asynchronous versions of `flushBatchRequests`. As you might expect, the `Connection::flushBatchRequests` operation flushes all batch requests queued for a particular connection, and the `Communicator::flushBatchRequests` operation flushes the batch requests of every connection created by a communicator.

The synchronous versions of `flushBatchRequests` block the calling thread until the batch requests have been successfully written to the local transport. To avoid the risk of blocking, you must use the asynchronous versions instead (assuming they are supported by your chosen language mapping). Note also that the asynchronous version of `Communicator::flushBatchRequests` never raises an exception, even if an error occurs while flushing one of its connections.

32.16.4 Batched Datagrams

For batched datagram invocations, you need to keep in mind that, if the data for the invocations in a batch substantially exceeds the PDU size of the network, it becomes increasingly likely for an individual UDP packet to get lost due to fragmentation. In turn, loss of even a single packet causes the entire batch to be lost. For this reason, batched datagram invocations are most suitable for simple interfaces with a number of operations that each set an attribute of the target object (or interfaces with similar semantics). (Batched oneway invocations do not suffer from this risk because they are sent over stream-oriented transports, so individual packets cannot be lost.)

32.16.5 Compression

Batched invocations are more efficient if you also enable compression for the transport: many isolated and small messages are unlikely to compress well, whereas batched messages are likely to provide better compression because the compression algorithm has more data to work with.[11]

11. Regardless of whether you used batched messages or not, you should enable compression only on lower-speed links. For high-speed LAN connections, the CPU time spent doing the compression and decompression is typically longer than the time it takes to just transmit the uncompressed data.

32.16.6 Active Connection Management

As for oneway invocations (see page 1053), you should disable server-side active connection management (ACM) when using batched invocations over TCP/IP or SSL. With server-side ACM enabled, it is possible for a server to close the connection at the wrong moment and not process a batch (with no indication being returned to the client that the batch was lost).

32.17 Location Services

In Section 2.2.2 we described briefly how the Ice run time uses an intermediary, known as a *location service*, to convert the symbolic information in an indirect proxy into an endpoint that it can use to communicate with a server. This section expands on that introduction to explain in more detail how the Ice run time interacts with a location service. You can create your own location service or you can use IceGrid, which is an implementation of a location service and provides many other useful features as well (see Chapter 38). Describing how to implement a location service is outside the scope of this book.

32.17.1 Locators

A *locator* is an Ice object that is implemented by a location service. A locator object must support the Slice interface `Ice::Locator`, which defines operations that satisfy the location requirements of the Ice run time. Applications do not normally use these operations directly, but the locator object may support an implementation-specific interface derived from `Ice::Locator` that provides additional functionality. For example, IceGrid's locator object implements the derived interface `IceGrid::Query` (see Section 38.6.5).

32.17.2 Client Semantics

On the first use of an indirect proxy in an application, the Ice run time may issue a remote invocation on the locator object. This activity is transparent to the application, as shown in Figure 32.6.

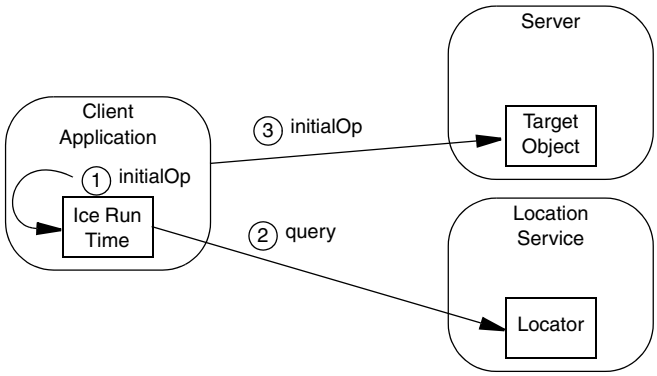


Figure 32.6. Locating an object.

1. The client invokes the operation `initialOp` on an indirect proxy.
2. The Ice run time checks an internal cache (called the *locator cache*) to determine whether a query has already been issued for the symbolic information in the proxy. If so, the cached endpoint is used and an invocation on the locator object is avoided. Otherwise, the Ice run time issues a locate request to the locator.
3. If the object is successfully located, the locator returns its current endpoints. The Ice run time in the client caches this information, establishes a connection to one of the endpoints according to the rules described in Section 36.3.1, and proceeds to send the invocation as usual.
4. If the object's endpoints cannot be determined, the client receives an exception. `NotRegisteredException` is raised when an identity, object adapter identifier or replica group identifier is not known. A client may also receive `NoEndpointException` if the location service failed to determine the current endpoints.

As far as the Ice run time is concerned, the locator simply converts the information in an indirect proxy into usable endpoints. Whether the locator's implementation is more sophisticated than a simple lookup table is irrelevant to the Ice run

time. However, the act of performing this conversion may have additional semantics that the application must be prepared to accept.

For example, when using IceGrid as your location service, the target server may be launched automatically if it is not currently running, and the locate request does not complete until that server is started and ready to receive requests. As a result, the initial request on an indirect proxy may incur additional overhead as all of this activity occurs.

Replication

An indirect proxy may substitute a replica group identifier (see page 13) in place of the object adapter identifier. In fact, the Ice run time does not distinguish between these two cases and considers a replica group identifier as equivalent to an object adapter identifier for the purposes of resolving the proxy. The location service implementation must be able to distinguish between replica groups and object adapters using only this identifier.

The location service may return multiple endpoints in response to a locate request for an adapter or replica group identifier. These endpoints might all correspond to a single object adapter that is available at several addresses, or to multiple object adapters each listening at a single address, or some combination thereof. The Ice run time attaches no semantics to the collection of endpoints, but the application can make assumptions based on its knowledge of the location service's behavior.

When a location service returns more than one endpoint, the Ice run time behaves exactly as if the proxy had contained several endpoints (see Section 36.3.1). As always, the goal of the Ice run time is to establish a connection to one of the endpoints and deliver the client's request. By default, all requests made via the proxy that initiated the connection are sent to the same server until that connection is closed.

After the connection is closed, such as by active connection management (see Section 36.4), subsequent use of the proxy causes the Ice run time to obtain another connection. Whether that connection uses a different endpoint than previous connections depends on a number of factors, but it is possible for the client to connect to a different server than for previous requests.

Locator Cache

After successfully resolving an indirect proxy, the location service must return at least one endpoint. How the service derives the list of endpoints that corresponds to the proxy is entirely implementation dependent. For example, IceGrid's loca-

tion service can be configured to respond in a variety of ways; one possibility uses a simple round-robin scheme, while another considers the system load of the target hosts when selecting endpoints.

A locate request has the potential to significantly increase the latency of the application's invocation with a proxy, and this is especially true if the locate request triggers additional implicit actions such as starting a new server process. Fortunately, this overhead is normally incurred only during the application's initial invocation on the proxy, but this impact is influenced by the Ice run time's caching behavior.

To minimize the number of locate requests, the Ice run time caches the results of previous requests. By default, the results are cached indefinitely, so that once the Ice run time has obtained the endpoints associated with an indirect proxy, it never issues another locate request for that proxy. Furthermore, the default behavior of a proxy is to cache its connection, that is, once a proxy has obtained a connection, it continues to use that connection indefinitely.

Taken together, these two caching characteristics represent the Ice run time's best efforts to optimize an application's use of a location service: after a proxy is associated with a connection, all future invocations on that proxy are sent on the same connection without any need for cache lookups, locate requests, or new connections.

If a proxy's connection is closed, the next invocation on the proxy prompts the Ice run time to consult its locator cache to obtain the endpoints from the prior locate request. Next, the Ice run time searches for an existing connection to any of those endpoints and uses that if possible, otherwise it attempts to establish a new connection to each of the endpoints until one succeeds. Only if that process fails does the Ice run time clear the entry from its cache and issue a new locate request with the expectation that a usable endpoint is returned.

The Ice run time's default behavior is optimized for applications that require minimal interaction with the location service, but some applications can benefit from more frequent locate requests. Normally this is desirable when implementing a load-balancing strategy, as we discuss in more detail below. In order to increase the frequency of locate requests, an application must configure a timeout for the locator cache and manipulate the connections of its proxies.

Locator Cache Timeout

An application can define a timeout to control the lifetime of entries in the locator cache. This timeout can be specified globally using the `Ice.Default.LocatorCacheTimeout` property (see Section D.8) and for individual proxies using

the proxy method `ice_locatorCacheTimeout` (see Section 32.11.2). The Ice run time's default behavior is equivalent to a timeout value of `-1`, meaning the cache entries never expire. Using a timeout value greater than zero causes the cache entries to expire after the specified number of seconds. Finally, a timeout value of zero disables the locator cache altogether.

The previous section explained the circumstances in which the Ice run time consults the locator cache. Briefly, this occurs only when the application has invoked an operation on a proxy and the proxy is not currently associated with a connection. If the timeout is set to zero, the Ice run time issues a new locate request immediately. Otherwise, for a non-zero timeout, the Ice run time examines the locator cache to determine whether the endpoints from the previous locate request have expired. If so, the Ice run time discards them and issues a new locate request.

Given this behavior, if your goal is to force a proxy invocation to issue locate requests more frequently, you can do so only when the proxy has no connection. You can accomplish that in several ways:

- create a new proxy, which is inherently not connected by default
- explicitly close the proxy's existing connection (see Chapter 36)
- disabling the proxy's connection caching behavior

Of these choices, the last is the most common.

Proxy Connection Caching

By default a proxy remembers its connection and uses it for all invocations until that connection is closed. You can prevent a proxy from caching its connection by calling the `ice_connectionCached` proxy method with an argument of false (see Section 32.11.2). Once connection caching is disabled, each invocation on a proxy causes the Ice run time to execute its connection establishment process.

Note that each invocation on such a proxy does not necessarily cause the Ice run time to establish a new connection. It only means that the Ice run time does not assume that it can reuse the connection of the proxy's previous invocation. Whether the Ice run time actually needs to establish a new connection for the next invocation depends on several factors, as explained in Section 36.3.

As with any feature, you should only use it when the benefits outweigh the risks. With respect to a proxy's connection caching behavior, there is certainly a small amount of computational overhead associated with executing the connection establishment process for each invocation, as well as the risk of significant overhead each time a new connection is actually created.

Simple Load Balancing

Several forms of load balancing are available to Ice applications. The simplest form uses only the endpoints contained in a direct proxy and does not require a location service. In this configuration, the application can configure the proxy to use the desired endpoint selection type (see Section 36.3.1) and connection caching behavior to achieve the desired results.

For example, suppose that a proxy contains several endpoints. In its default configuration, it uses the `Random` endpoint selection type and caches its connection. Upon the first invocation, the Ice run time selects one of the proxy's endpoints at random and uses that connection for all subsequent invocations until the connection is closed. For some applications, this form of load balancing may be sufficient.

Suppose now that we use the `Ordered` endpoint selection type instead. In this case, the Ice run time always attempts to establish connections using the endpoints in the order they appear in the proxy. Normally an application uses this configuration when there is a preferred order to the servers. Again, once connected, the application uses whichever connection was chosen indefinitely.

By disabling the proxy's connection caching behavior, the semantics undergo a significant change. Using the `Random` endpoint selection type, the Ice run time selects one of the endpoints at random and establishes a connection to it if one is not already established, and this process is repeated prior to each subsequent invocation. This is called *per-request load balancing* because each request can be directed to a different server. Using the `Ordered` endpoint selection type is not as common in this scenario; its main purpose would be to fall back on a secondary server if the primary server is not available, but it causes the Ice run time to attempt to contact the primary server during each request.

Load Balancing with a Location Service

A disadvantage of relying solely on the simple form of load balancing described in the previous section is that the client cannot make any intelligent decisions based on the status of the servers. If you want to distribute your requests in a more sophisticated way, you must either modify your clients to query the servers directly, or use a location service that can transparently direct a client to an appropriate server. For example, the IceGrid location service can monitor the system load on each server host and use that information when responding to locate requests.

The location service may return only one endpoint, which presumably represents the best server (at that moment) for the client to use. With only one endpoint

available, changing the proxy's endpoint selection type makes no difference. However, by disabling connection caching and modifying the locator cache timeout, the application can force the Ice run time to periodically retrieve an updated endpoint from the location service. For example, an application can set a locator cache timeout of thirty seconds and communicate with the selected server for that period. After the timeout has expired, the next invocation prompts the Ice run time to issue a new locate request, at which point the client might be directed to a different server.

If the location service returns multiple endpoints, the application must be designed with knowledge of how to interpret them. For instance, the location service may attach semantics to the order of the endpoints (such as least-loaded to most-loaded) and intend that the application use the endpoints in the order provided. Alternatively, the client may be free to select any of the endpoints. As a result, the application and the location service must cooperate to achieve the desired results.

You can combine the simple form of load balancing described in the previous section with an intelligent location service to gain even more flexibility. For example, suppose an application expects to receive multiple endpoints from the location service and has configured its proxy to disable connection caching and set a locator cache timeout. For each invocation, Ice run time selects one of the endpoints provided by the location service. When the timeout expires, the Ice run time issues a new locate request and obtains a fresh set of endpoints from which to choose.

32.17.3 Configuring a Client

An Ice client application must supply a proxy for the locator object, which it can do in several ways:

- by explicitly configuring an indirect proxy using the `ice_locator` proxy method (see Section 32.11.2)
- by calling `setDefaultLocator` on a communicator, after which all proxies use the given locator by default
- by defining the `Ice.Default.Locator` configuration property, which causes all proxies to use the given locator by default

The Ice run time's efforts to resolve an indirect proxy can be traced by setting the following configuration properties:

```
Ice.Trace.Network=2
Ice.Trace.Protocol=1
Ice.Trace.Locator=2
```

See Appendix D for more information on these properties.

32.17.4 Server Semantics

A location service must know the endpoints of any object adapter whose identifier can be used in an indirect proxy. For example, suppose a client uses the following proxy:

```
Object1@PublicAdapter
```

The Ice run time in the client sends the identifier `PublicAdapter` to the locator, as described in Section 32.17.2, and expects to receive the associated endpoints. The only way the location service can know these endpoints is if it is given them. When you consider that an object adapter's endpoints may not specify fixed ports, and therefore the endpoint addresses may change each time the object adapter is activated, it is clear that the best source of endpoint information is the object adapter itself. As a result, an object adapter that is properly configured (see Section 32.17.5) contacts the locator during activation to supply its identifier and current endpoints. More specifically, the object adapter registers itself with an object implementing the `Ice::LocatorRegistry` interface, whose proxy the object adapter obtains from the locator.

Registration Requirements

A location service may require that all object adapters be pre-registered via some implementation-specific mechanism. (IceGrid behaves this way by default.) This implies that activation can fail if the object adapter supplies an identifier that is unknown to the location service. In such a situation, the object adapter's `activate` operation raises `NotRegisteredException`.

In a similar manner, an object adapter that participates in a replica group (see page 13) includes the group's identifier in the locator request that is sent during activation. If the location service requires replica group members to be configured in advance, `activate` raises `NotRegisteredException` if the object adapter's identifier is not one of the group's registered participants (see Section 38.9.2).

32.17.5 Configuring a Server

An object adapter must be able to obtain a locator proxy in order to register itself with a location service. Each object adapter can be configured with its own locator proxy by defining its `Locator` property, as shown in the example below for the object adapter named `SampleAdapter`:

```
SampleAdapter.Locator=IceGrid/Locator:tcp -h locatorhost -p 10000
```

Alternatively, a server may call `setLocator` on the object adapter prior to activation. If the object adapter is not explicitly configured with a locator proxy, it uses the default locator as provided by its communicator (see Section 32.17.3).

Two other configuration properties influence an object adapter's interactions with a location service during activation:

- `AdapterId`

  Configuring a non-empty identifier for the `AdapterId` property causes the object adapter to register itself with the location service. A locator proxy must also be configured.

- `ReplicaGroupId`

  Configuring a non-empty identifier for the `ReplicaGroupId` property indicates that the object adapter is a member of a replica group (see page 13). For this property to have an effect, `AdapterId` must also be configured with a non-empty value.

We can use these properties as shown below:

```
SampleAdapter.AdapterId=SampleAdapterId
SampleAdapter.ReplicaGroupId=SampleGroupId
SampleAdapter.Locator=IceGrid/Locator:tcp -h locatorhost -p 10000
```

Refer to Section 32.17.4 for information on the pre-registration requirements a location service may enforce.

32.17.6 Process Registration

An activation service, such as an IceGrid node (see Chapter 38), needs a reliable way to gracefully deactivate a server. One approach is to use a platform-specific mechanism, such as POSIX signals. This works well on POSIX platforms when the server is prepared to intercept signals and react appropriately (see Section 31.10). On Windows platforms, it works less reliably for C++ servers, and not at all for Java servers. For these reasons, Ice provides an alternative that is both portable and reliable:

```
module Ice {
interface Process {
    ["ami"] void shutdown();
    void writeMessage(string message, int fd);
};
};
```

The Slice interface `Ice::Process` allows an activation service to request a graceful shutdown of the server. When `shutdown` is invoked, the object implementing this interface is expected to initiate the termination of its server process. The activation service may expect the server to terminate within a certain period of time, after which it may terminate the server abruptly.

One of the benefits of the Ice administrative facility (see Section 32.18) is that it creates an implementation of `Ice::Process` and makes it available via an administrative object adapter. Furthermore, IceGrid automatically enables this facility on the servers that it activates.

32.18 Administrative Facility

Ice applications often require remote administration, such as when an IceGrid node needs to gracefully deactivate a running server. The Ice run time provides an extensible, centralized facility for exporting administrative functionality. This facility consists of an object adapter named `Ice.Admin`, an Ice object activated on this adapter, and configuration properties that enable the facility and specify its features.

32.18.1 The `admin` Object

The `Ice.Admin` adapter hosts a single object whose identity name is `admin`. Although this identity name cannot be changed, you can define the identity category using the configuration property `Ice.Admin.InstanceName` (see Appendix D). If you enable the `Ice.Admin` adapter without defining this property, the category uses a UUID by default and therefore the object's identity changes with each instance of the process.

In this book, we refer to the administrative object as the `admin` object.

Facets

As explained in Chapter 33, an Ice object is actually a collection of sub-objects known as facets whose types are not necessarily related. Although facets are typically used for extending and versioning types, they also allow a group of interfaces with a common purpose to be consolidated into a single Ice object with an established interface for navigation. These qualities make facets an excellent match for the requirements of the administrative facility.

Each facet of the `admin` object represents a distinct administrative capability. The object does not have a default facet (that is, a facet with an empty name). However, the Ice run time implements two built-in facets that it adds to the `admin` object:

- `Process`, described in Section 32.18.4
- `Properties`, described in Section 32.18.5

An application can control which facets are installed with a configuration property (see Section 32.18.6). An application can also install its own facets if necessary (see Section 32.18.7). Administrative facets are not required to inherit from a common Slice interface.

32.18.2 Enabling the Object Adapter

The administrative facility is disabled by default. To enable it, you must specify endpoints for the administrative object adapter using the property `Ice.Admin.Endpoints`. In addition, you must do one of the following:

- Define the `Ice.Admin.InstanceName` property.
- Define the `Ice.Admin.ServerId` and `Ice.Default.Locator` properties. If you do not supply a value for `Ice.Admin.InstanceName`, Ice uses a UUID by default.

The `Ice.Admin.ServerId` and `Ice.Default.Locator` properties are typically used in conjunction with an activation service such as IceGrid, as discussed in Section 38.22.

The endpoints for the `Ice.Admin` adapter must be chosen with caution. Section 32.18.8 addresses the security considerations of using the administrative facility.

It may be necessary to postpone the creation of the administrative object adapter until all facets are installed or other initialization activities have taken place. In this situation, you can define the following configuration property:

```
Ice.Admin.DelayCreation=1
```

When this property is set to a non-zero value, the administrative facility is disabled until the application invokes the `getAdmin` operation on the communicator (see Section 32.18.3).

32.18.3 Using the `admin` Object

A program can obtain a proxy for its `admin` object by calling the `getAdmin` operation on a communicator:

```
module Ice {
local interface Communicator {
    // ...
    Object* getAdmin();
};
};
```

This operation returns a null proxy if the administrative facility is disabled. The proxy returned by `getAdmin` cannot be used for invoking operations because it refers to the default facet and, as we mentioned previously, the `admin` object does not support a default facet. A program must first obtain a new version of the proxy that is configured with the name of a particular administrative facet before invoking operations on it. Although it cannot be used for invocations, the original proxy is still useful because it contains the endpoints of the `Ice.Admin` object adapter and therefore the program may elect to export that proxy to a remote client.

Remote Administration

To administer a program remotely, somehow you must obtain a proxy for the program's `admin` object. There are several ways for the administrative client to accomplish this:

- Construct the proxy itself, assuming that it knows the `admin` object's identity, facets, and endpoints. The format of the stringified proxy is as follows:

  `instance-name``/admin -f` *`admin-facet`*`:`*`admin-endpoints`*

  The identity category, represented here by *`instance-name`*, is the value of the `Ice.Admin.InstanceName` property or a UUID if that property is not defined. (Clearly, the use of a UUID makes the proxy much more difficult for a client to construct on its own.) The name of the administrative facet is supplied as the value of the `-f` option, and the endpoints of the `Ice.Admin`

adapter appear last in the proxy. See Appendix E for more information on stringified proxies.

- Invoke an application-specific interface for retrieving the `admin` object's proxy.
- Use the `getServerAdmin` operation on the `IceGrid::Admin` interface, if the remote program was activated by IceGrid (see Section 38.22.3).

Having obtained the proxy, the administrative client must select a facet before invoking any operations. For example, the code below shows how to obtain the configuration properties of the remote program:

```
// C++
Ice::ObjectPrx adminObj = ...;
Ice::PropertiesAdminPrx propAdmin =
    Ice::PropertiesAdminPrx::checkedCast(adminObj,
                                         "Properties");
Ice::PropertyDict props = propAdmin->getPropertiesForPrefix("");
```

Here we used an overloaded version of `checkedCast` to supply the facet name of interest (`Properties`). We could have selected the facet using the proxy method `ice_facet` instead, as shown below:

```
// C++
Ice::ObjectPrx adminObj = ...;
Ice::PropertiesAdminPrx propAdmin =
    Ice::PropertiesAdminPrx::checkedCast(
        adminObj->ice_facet("Properties"));
Ice::PropertyDict props = propAdmin->getPropertiesForPrefix("");
```

This code is functionally equivalent to the first example.

A remote client must also know (or be able to determine) which facets are available in the target server. Typically this information is statically configured in the client, since the client must also know the interface types of any facets that it uses. If an invocation on a facet raises `FacetNotExistException`, the client may have used an incorrect facet name, or the server may have disabled the facet in question.

32.18.4 The `Process` Facet

An activation service, such as an IceGrid node (see Chapter 38), needs a reliable way to gracefully deactivate a server. One approach is to use a platform-specific mechanism, such as POSIX signals. This works well on POSIX platforms when the server is prepared to intercept signals and react appropriately (see

Section 31.10). On Windows platforms, it works less reliably for C++ servers, and not at all for Java servers. For these reasons, the `Process` facet provides an alternative that is both portable and reliable.

Section 32.18.8 discusses the security risks associated with enabling the `Process` facet.

Interface

The Slice interface `Ice::Process` allows an activation service to request a graceful shutdown of the program:

```
module Ice {
interface Process {
    ["ami"] void shutdown();
    void writeMessage(string message, int fd);
};
};
```

When `shutdown` is invoked, the object implementing this interface is expected to initiate the termination of its process. The activation service may expect the program to terminate within a certain period of time, after which it may terminate the program abruptly.

The `writeMessage` operation allows remote clients to print a message to the program's standard output (`fd` == 1) or standard error (`fd` == 2) channels.

Application Requirements

The default implementation of the `Process` facet requires cooperation from an application in order to successfully terminate a process. Specifically, the facet invokes `shutdown` on its communicator and assumes that the application uses this event as a signal to commence its termination procedure. For example, an application typically uses a thread (often the main thread) to call the communicator operation `waitForShutdown`, which blocks the calling thread until the communicator is shut down or destroyed. After `waitForShutdown` returns, the calling thread can initiate a graceful shutdown of its process.

Refer to Section 32.2 for more information on the communicator operations `shutdown` and `waitForShutdown`.

Replacing the `Process` Facet

You can replace the default `Process` facet if your application requires a different scheme for gracefully shutting itself down. To define your own facet, create a

servant that implements the `Ice::Process` interface. As an example, the servant definition shown below duplicates the functionality of the default `Process` facet:

```
class ProcessI : public Ice::Process {
public:
    ProcessI(const Ice::CommunicatorPtr& communicator) :
        _communicator(communicator)
    {}

    void shutdown(const Ice::Current&)
    {
        _communicator->shutdown();
    }

    void writeMessage(const string& msg, Ice::Int fd,
                      const Ice::Current&)
    {
        if(fd == 1) cout << msg << endl;
        else if(fd == 2) cerr << msg << endl;
    }

private:
    const Ice::CommunicatorPtr _communicator;
};
```

As you can see, the default implementation of `shutdown` simply shuts down the communicator, which initiates an orderly termination of the Ice run time's server-side components and prevents object adapters from dispatching any new requests. You can add your own application-specific behavior to the `shutdown` method to ensure that your program terminates in a timely manner.

NOTE: As explained in Section 32.2, a servant must not invoke `destroy` on its communicator while executing a dispatched operation.

To avoid the risk of a race condition, the recommended strategy for replacing the `Process` facet is to delay creation of the administrative facets so that your application has a chance to replace the facet:

```
Ice.Admin.DelayCreation=1
```

With this property defined, the application can safely remove the default `Process` facet and install its own:

```
// C++
Ice::CommunicatorPtr communicator = ...
communicator->removeAdminFacet("Process");
Ice::ProcessPtr myProcessFacet = new MyProcessFacet(...);
communicator->addAdminFacet(myProcessFacet, "Process");
```

The final step is to activate the administrative facility by calling `getAdmin` on the communicator:

```
communicator->getAdmin();
```

Integration with an Activation Service

If the `Ice.Admin.ServerId` and `Ice.Default.Locator` properties are defined, the Ice run time performs the following steps after creating the `Ice.Admin` adapter:

- Obtains proxies for the `Process` facet and the default locator
- Invokes `getRegistry` on the proxy to obtain a proxy for the locator registry
- Invokes `setServerProcessProxy` on the locator registry and supplies the value of `Ice.Admin.ServerId` along with a proxy for the `Process` facet

The identifier specified by `Ice.Admin.ServerId` must uniquely identify the process within the locator registry.

In the case of IceGrid, the node defines the `Ice.Admin.ServerId` and `Ice.Default.Locator` properties for each deployed server. The node also supplies a value for `Ice.Admin.Endpoints` if the property is not defined by the server. See Chapter 38 for more information.

32.18.5 The `Properties` Facet

An administrator may find it useful to be able to view the configuration properties of a remote Ice application. For example, the IceGrid administrative tools allow you to query the properties of active servers. The `Properties` facet supplies this functionality.

Interface

The `Ice::PropertiesAdmin` interface provides access to the communicator's configuration properties:

```
module Ice {
interface PropertiesAdmin {
    ["ami"] string getProperty(string key);
    ["ami"] PropertyDict getPropertiesForPrefix(string prefix);
};
};
```

The `getProperty` operation retrieves the value of a single property, and the `getPropertiesForPrefix` operation returns a dictionary of properties whose keys match the given prefix. These operations have the same semantics as those in the `Ice::Properties` interface described in Section 30.9.1.

32.18.6 Filtering Facets

The Ice run time enables all of its built-in administrative facets by default, and an application may install its own facets. You can control which facets the Ice run time enables using the `Ice.Admin.Facets` property. For example, the following property definition enables the `Properties` facet and leaves the `Process` facet (and any application-defined facets) disabled:

```
Ice.Admin.Facets=Properties
```

To specify more than one facet, separate them with a comma or white space. A facet whose name contains white space must be enclosed in single or double quotes.

32.18.7 Custom Facets

An application can add and remove administrative facets using the `Communicator` operations shown below:

```
module Ice {
local interface Communicator {
    // ...
    void addAdminFacet(Object servant, string facet);
    Object removeAdminFacet(string facet);
};
};
```

The `addAdminFacet` operation installs a new facet with the given name, or raises `AlreadyRegisteredException` if a facet already exists with the same name. The `removeAdminFacet` operation removes (and returns) the facet with the given name, or raises `NotRegisteredException` if no matching facet is found.

The mechanism for filtering administrative facets described in Section 32.18.6 also applies to application-defined facets. If you call `addAdminFacet` while a filter is in effect, and the name of your custom facet does not match the filter, the Ice run time will not expose your facet but instead keeps a reference to it so that a subsequent call to `removeAdminFacet` is possible.

32.18.8 Security Considerations

Exposing administrative functionality naturally makes a program vulnerable, therefore it is important that proper precautions are taken.

Issues

With respect to the default functionality, the `Properties` facet could expose sensitive configuration information, and the `Process` facet supports a `shutdown` operation that opens the door for a denial-of-service attack.

Developers should carefully consider the security implications of any additional administrative facets that an application installs.

Remedies

There are several approaches you can take to mitigate the possibility of abuse:

- Disable the administrative facility

  The administrative facility is disabled by default, and remains disabled as long as the prerequisites listed in Section 32.18.2 are not met. Note that IceGrid enables the facility in servers that it activates for the following reasons:

  - The `Process` facet allows the IceGrid node to gracefully terminate the process.
  - The `Properties` facet enables IceGrid administrative clients to obtain configuration information about activated servers.

  You could disable a facet using filtering, but doing so may disrupt IceGrid's normal operation.

- Select a proper endpoint

  A reasonably secure value for the `Ice.Admin.Endpoints` property is one that uses the local host interface (`-h 127.0.0.1`), which restricts access to clients that run on the same host. Incidentally, this is the default value that IceGrid defines for its servers, although you can override that if you like. Note that using a local host endpoint does not preclude remote administration for

IceGrid servers because IceGrid transparently routes requests on admin objects to the appropriate server via its node (see Section 38.22.3).

If your application must support administration from non-local hosts, we recommend the use of SSL and certificate-based access control (see Chapter 41).

- Filter the facets

  After choosing a suitable endpoint, you can minimize risks by filtering the facets to enable only the functionality that is required. For example, if you are not using IceGrid's server activation feature and do not require the ability to remotely terminate a program, you should disable the Process facet using the mechanism described in Section 32.18.6.

- Consider the object's identity

  The default identity of the admin object has a UUID for its category, which makes it difficult for a hostile client to guess. Depending on your requirements, the use of a UUID may be an advantage or a disadvantage. For example, in a trusted environment, the use of a UUID may create additional work, such as the need to add an interface that an administrative client can use to obtain the identity or proxy of a remote admin object. An obscure identity might be more of a hindrance in this situation, and therefore specifying a static category via the Ice.Admin.InstanceName property is a reasonable alternative. In general, however, we recommend using the default behavior.

32.19 The Ice::Logger Interface

Depending on the setting of various properties (see Chapter 30), the Ice run time produces trace, warning, or error messages. These messages are written via the Ice::Logger interface:

```
module Ice {
    local interface Logger {
        void print(string message);
        void trace(string category, string message);
        void warning(string message);
        void error(string message);
        Logger cloneWithPrefix(string prefix);
    };
};
```

The `cloneWithPrefix` operation returns a new logger that logs to the same destination but with a different prefix. (The prefix is used to, for example, provide the name of the process writing the log messages.)

32.19.1 The Default Logger

A default logger is instantiated when you create a communicator. The default logger logs to the standard error output. The `trace` operation accepts a `category` parameter in addition to the error message; this allows you to separate trace output from different subsystems by sending the output through a filter.

You can obtain the logger that is attached to a communicator:

```
module Ice {
    local interface Communicator {
        Logger getLogger();
    };
};
```

32.19.2 Custom Loggers

You have several options if you wish to install a logger other than the default one:

- Select one of the other built-in loggers (see Section 32.19.3), which allow you to log to a file, to the **`syslog`** on Unix, and to the Windows event log
- Supply your own logger implementation in an `InitializationData` parameter (see Section 32.3) when you create a communicator
- Load a logger implementation dynamically via the Ice plug-in facility (see Section 32.19.4)

Changing the `Logger` object that is attached to a communicator allows you to integrate Ice messages into your own message handling system. For example, for a complex application, you might have an existing logging framework. To integrate Ice messages into that framework, you can create your own `Logger` implementation that logs messages to the existing framework.

When you destroy a communicator, its logger is *not* destroyed. This means that you can safely use a logger even beyond the lifetime of its communicator.

32.19.3 Built-In Loggers

A file-based logger, enabled via the `Ice.LogFile` property, is available for all supported languages and platforms. Ice also provides Unix- and Windows-specific

logger implementations. Finally, for .NET, the default Ice logger uses a TraceListener and so can be customized at run time via configuration.

Refer to Appendix D for more information on the properties mentioned in this section.

Unix

You can activate a logger that logs via the Unix **syslog** implementation by setting the Ice.UseSyslog property. This logger is available for C++, Java, and C# applications.

Windows

On Windows, subclasses of Ice::Service use the Windows application event log by default (see Section 8.3.2). The event log implementation is available for C++ applications.

.NET

The default logger in Ice for .NET writes its messages using the System.Diagnostics.Trace facility. By default, the Ice run time registers a ConsoleTraceListener that writes to stderr. You can disable the logging of messages via this trace listener by setting the property Ice.ConsoleListener to zero.

You can change the trace listener for your application via the application's configuration file. For example:

```
<configuration>
  <system.diagnostics>
    <trace autoflush="true" indentsize="4">
      <listeners>
        <add name="Console"
             type="System.Diagnostics.EventLogTraceListener"
             initializeData="true"/>
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

This configures a trace listener that logs to the Windows event log.

32.19.4 Logger Plug-Ins

Installing a custom logger using the Ice plug-in facility has several advantages. Because the logger plug-in is specified by a configuration property and loaded dynamically by the Ice run time, an application requires no code changes in order to utilize a custom logger implementation. Furthermore, a logger plug-in takes precedence over the per-process logger (see Section 32.19.5) and the logger supplied in the `InitializationData` argument during communicator initialization, meaning you can use a logger plug-in to override the logger that an application installs by default.

Installing a C++ Logger Plug-In

To install a logger plug-in in C++, you must first define a subclass of `Ice::Logger`:

```
class MyLoggerI : public Ice::Logger {
public:

    virtual void print(const std::string& message);
    virtual void trace(const std::string& category,
                       const std::string& message);
    virtual void warning(const std::string& message);
    virtual void error(const std::string& message);
    virtual LoggerPtr void cloneWithPrefix(const string& prefix);

    // ...
};
```

Next, supply a factory function that installs your custom logger by returning an instance of `Ice::LoggerPlugin`:

```
extern "C"
{

ICE_DECLSPEC_EXPORT Ice::Plugin*
createLogger(const Ice::CommunicatorPtr& communicator,
             const std::string& name,
             const Ice::StringSeq& args)
{
    Ice::LoggerPtr logger = new MyLoggerI;
    return new Ice::LoggerPlugin(communicator, logger);
}

}
```

The factory function can have any name you wish; we used `createLogger` in this example. See Section 32.25.1 for more information on plug-in factory functions.

The definition of `LoggerPlugin` is shown below:

```
namespace Ice {
class LoggerPlugin {
public:
    LoggerPlugin(const CommunicatorPtr&, const LoggerPtr&);

    virtual void initialize();
    virtual void destroy();
};
}
```

The constructor installs your logger into the given communicator. The `initialize` and `destroy` methods are empty, but you can subclass `LoggerPlugin` and override these methods if necessary.

Finally, define a configuration property that loads your plug-in into an application:

```
Ice.Plugin.MyLogger=mylogger:createLogger
```

The plug-in's name in this example is `MyLogger`; again, you can use any name you wish. The value of the property represents the plug-in's entry point, in which `mylogger` is the abbreviated form of its shared library or DLL, and `createLogger` is the name of the factory function.

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for C++ run time:

```
Ice.Plugin.MyLogger.cpp=mylogger:createLogger
```

Refer to Appendix D for more information on the `Ice.Plugin` properties.

Installing a Java Logger Plug-In

To install a logger plug-in in Java, you must first define a subclass of `Ice.Logger`:

```
public class MyLoggerI implements Ice.Logger {

    public void print(String message) { ... }
    public void trace(String category, String message) { ... }
    public void warning(String message) { ... }
    public void error(String message) { ... }
```

```
    public Logger cloneWithPrefix(String prefix) { ... }

    // ...
}
```

Next, define a factory class that installs your custom logger by returning an instance of `Ice.LoggerPlugin`:

```
public class MyLoggerPluginFactoryI implements Ice.PluginFactory {
    public Ice.Plugin create(Ice.Communicator communicator,
                             String name, String[] args)
    {
        Ice.Logger logger = new MyLoggerI();
        return new Ice.LoggerPlugin(communicator, logger);
    }
}
```

The factory class can have any name you wish; in this example, we used `MyLoggerPluginFactoryI`. See Section 32.25.1 for more information on plug-in factories.

The definition of `LoggerPlugin` is shown below:

```
package Ice;

public class LoggerPlugin implements Plugin {
    public LoggerPlugin(Communicator communicator, Logger logger)
    {
        // ...
    }

    public void initialize() { }

    public void destroy() { }
}
```

The constructor installs your logger into the given communicator. The `initialize` and `destroy` methods are empty, but you can subclass `LoggerPlugin` and override these methods if necessary.

Finally, define a configuration property that loads your plug-in into an application:

```
Ice.Plugin.MyLogger=MyLoggerPluginFactoryI
```

The plug-in's name in this example is `MyLogger`; again, you can use any name you wish. The value of the property is the name of the factory class.

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for Java run time:

```
Ice.Plugin.MyLogger.java=MyLoggerPluginFactoryI
```

Refer to Appendix D for more information on the `Ice.Plugin` properties.

Installing a .NET Logger Plug-In

To install a logger plug-in in .NET, you must first define a subclass of `Ice.Logger`:

```
// C#
public class MyLoggerI : Ice.Logger {

    public void print(string message) { ... }
    public void trace(string category, string message) { ... }
    public void warning(string message) { ... }
    public void error(string message) { ... }
    public Logger cloneWithPrefix(string prefix) { ... }

    // ...
}
```

Next, define a factory class that installs your custom logger by returning an instance of `Ice.LoggerPlugin`:

```
// C#
public class MyLoggerPluginFactoryI : Ice.PluginFactory {
    public Ice.Plugin create(Ice.Communicator communicator,
                             string name, string[] args)
    {
        Ice.Logger logger = new MyLoggerI();
        return new Ice.LoggerPlugin(communicator, logger);
    }
}
```

The factory class can have any name you wish; in this example, we used `MyLoggerPluginFactoryI`. See Section 32.25.1 for more information on plug-in factories. Typically the logger implementation and the factory are compiled into a single assembly.

The definition of `LoggerPlugin` is shown below:

```
// C#
namespace Ice {
public partial class LoggerPlugin : Plugin {
    public LoggerPlugin(Communicator communicator, Logger logger)
    {
        // ...
    }

    public void initialize() { }

    public void destroy() { }
}
}
```

The constructor installs your logger into the given communicator. The `initialize` and `destroy` methods are empty, but you can subclass `LoggerPlugin` and override these methods if necessary.

Finally, define a configuration property that loads your plug-in into an application:

```
Ice.Plugin.MyLogger=mylogger.dll:MyLoggerPluginFactoryI
```

The plug-in's name in this example is `MyLogger`; again, you can use any name you wish. The value of the property is the entry point for the factory, consisting of an assembly name followed by the name of the factory class.

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for .NET run time:

```
Ice.Plugin.MyLogger.clr=mylogger.dll:MyLoggerPluginFactoryI
```

Refer to Appendix D for more information on the `Ice.Plugin` properties.

32.19.5 The Per-Process Logger

Ice allows you to install a per-process custom logger. This logger is used by all communicators that do not have their own specific logger established at the time a communicator is created.

You can set a per-process logger in C++ by calling `Ice::setProcessLogger`, and you can retrieve the per-process logger by calling `Ice::getProcessLogger`:

```
LoggerPtr getProcessLogger();
void setProcessLogger(const LoggerPtr&);
```

If you call getProcessLogger without having called setProcessLogger first, the Ice run time installs a default per-process logger. Note that if you call setProcessLogger, only communicators created after that point will use this per-process logger; communicators created earlier use the logger that was in effect at the time they were created. (This also means that you can call setProcess-Logger multiple times; communicators created after that point will use whatever logger was established by the last call to setProcessLogger.)

getProcessLogger and setProcessLogger are language-specific APIs that are not defined in Slice. Therefore, for Java and C#, these methods appear in the Ice.Util class.

For applications that use the Application or Service convenience classes and do not explicitly configure a logger, these classes set a default per-process logger that uses the Ice.ProgramName property as a prefix for log messages. The Application class is described in the server-side language mapping chapters; more information on the Service class can be found in Section 8.3.2.

32.19.6 C++ Utility Classes

The Ice run time supplies a collection of utility classes that make use of the logger facility simpler and more convenient. Each of the logger's four operations has a corresponding helper class:

```
namespace Ice {
class Print {
public:
    Print(const LoggerPtr&);
    void flush();
    ...
};

class Trace {
public:
    Trace(const LoggerPtr&, const std::string&);
    void flush();
    ...
};

class Warning {
public:
    Warning(const LoggerPtr&);
    void flush();
```

```
    ...
};

class Error {
public:
    Error(const LoggerPtr&);
    void flush();
    ...
};
}
```

The only notable difference among these classes is the extra argument to the `Trace` constructor; this argument represents the trace category.

To use one of the helper classes in your application, you simply instantiate it and compose your message:

```
if (errorCondition) {
    Error err(communicator->getLogger());
    err << "encountered error condition: " << errorCondition;
}
```

The Ice run time defines the necessary stream insertion operators so that you can treat an instance of a helper class as if it were a standard C++ output stream. When the helper object is destroyed, its destructor logs the message you have composed. If you want to log more than one message using the same helper object, invoke the `flush` method on the object to log what you have composed so far and reset the object for a new message.

The helper classes also supply insertion operators to simplify the task of logging an exception. The operators accept instances of `std::exception` (from which all Ice exceptions derive) and log the string returned by the `what` method. If you are using GCC, you can also enable the configuration property `Ice.PrintStackTraces` (see Appendix D), which causes the helper classes to log the stack trace of the exception in addition to the value of `what`.

32.20 The `Ice::Stats` Interface

The Ice run time reports bytes sent and received over the wire on every operation invocation via the `Ice::Stats` interface:

```
module Ice {
    local interface Stats {
        void bytesSent(string protocol, int num);
        void bytesReceived(string protocol, int num);
    };

    local interface Communicator {
        Stats getStats();
        // ...
    };
};
```

The Ice run time calls bytesReceived as it reads from the network and bytesSent as it writes to the network. A very simple implementation of the Stats interface could look like the following:

```
class MyStats : public virtual Ice::Stats {
public:
    virtual void bytesSent(const string& prot, Ice::Int num)
    {
        cerr << prot << ": sent " << num << "bytes" << endl;
    }

    virtual void bytesReceived(const string& prot, Ice::Int)
    {
        cerr << prot << ": received " << num << "bytes" << endl;
    }
};
```

To register your implementation, you must pass it in an InitializationData parameter when you call initialize to create the communicator (see Section 32.3):

```
Ice::InitializationData id;
id.stats = new MyStats;
Ice::CommunicatorPtr ic = Ice::initialize(id);
```

You can install a Stats object on either the client or the server side (or both). Here is some example output produced by installing a MyStats object in a simple server:

```
tcp: received 14 bytes
tcp: received 32 bytes
tcp: sent 26 bytes
tcp: received 14 bytes
tcp: received 33 bytes
tcp: sent 25 bytes
...
```

In practice, your `Stats` implementation will probably be a bit more sophisticated: for example, the object can accumulate statistics in member variables and make the accumulated statistics available via member functions, instead of simply printing everything to the standard error output.

32.21 Location Transparency

One of the useful features of the Ice run time is that it is *location transparent*: the client does not need to know where the implementation of an Ice object resides; an invocation on an object automatically is directed to the correct target, whether the object is implemented in the local address space, in another address space on the same machine, or in another address space on a remote machine. Location transparency is important because it allows us to change the location of an object implementation without breaking client programs and, by using IceGrid (see Chapter 38), addressing information such as domain names and port numbers can be externalized so they do not appear in stringified proxies.

For invocations that cross address space boundaries (or more accurately, cross communicator boundaries), the Ice run time dispatches requests via the appropriate transport. However, for invocations that are via proxies for which the proxies and the servants that process the invocation share the same communicator (so-called *collocated* invocations), the Ice run time, by default, does not send the invocation via the transport specified in the proxy. Instead, collocated invocations are short-cut inside the Ice run time and dispatched directly.[12]

The reason for this is efficiency: if collocated invocations were sent via TCP/IP, for example, invocations would still be sent via the operating system kernel (using the back plane instead of a network) and would incur the full cost of creating TCP/IP connections, marshaling requests into packets, trapping in and

12. Note that if the proxy and the servant do not use the same communicator, the invocation is *not* collocated, even though caller and callee are in the same address space.

out of the kernel, and so on. By optimizing collocated requests, much of this overhead can be avoided, so collocated invocations are almost as fast as a local function call.

For efficiency reasons, collocated invocations are not completely location transparent, that is, a collocated call has semantics that differ in some ways from calls that cross address-space boundaries. Specifically, collocated invocations differ from ordinary invocations in the following respects:

- Collocated invocations are dispatched in the calling thread instead of being dispatched using the server's concurrency model.
- The object adapter holding state is ignored: collocated invocations proceed normally even if the target object's adapter is in the holding state.
- For collocated invocations, classes and exceptions are never sliced. Instead, the receiver always receives a class or exception as the derived type that was sent by the sender.
- If a collocated invocation throws an exception that is not in an operation's exception specification, the original exception is raised in the client instead of `UnknownUserException`. (This applies to the C++ mapping only.)
- Class factories are ignored for collocated invocations.
- Timeouts on invocations are ignored.
- If an operation implementation uses an in parameter that is passed by reference as a temporary variable, the change affects the value of the in parameter in the caller (instead of modifying a temporary copy of the parameter on the callee side only).

In practice, these differences rarely matter. The most likely cause of surprises with collocated invocations is dispatch in the calling thread, that is, a collocated invocation behaves like a local, synchronous procedure call. This can cause problems if, for example, the calling thread acquires a lock that an operation implementation tries to acquire as well: unless you use recursive mutexes (see Chapter 31), this will cause deadlock.

The Ice run time uses the following semantics to determine whether a proxy is eligible for the collocated optimization:

- For an indirect proxy, collocation optimization is used if the proxy's adapter id matches the adapter id or replica group id of an object adapter in the same communicator.
- For a well-known proxy, the Ice run time queries each object adapter to determine if the servant is local.

- For a direct proxy, the Ice run time performs an endpoint search using the proxy's endpoints.

When an endpoint search is required, the Ice run time compares each of the proxy's endpoints against the endpoints of the communicator's object adapters. Only the transport, address and port are considered; other attributes of an endpoint, such as timeout settings, are not considered during this search. If a match is found, the invocation is dispatched using collocation optimization. Normally this search is executed only once, during the proxy's first invocation, although the proxy's connection caching setting influences this behavior (see Section 36.3.4).

Collocation optimization is enabled by default, but you can disable it for all proxies by setting the property `Ice.Default.CollocationOptimized=0` (see page 1861). You can also disable the optimization for an individual proxy using the factory method `ice_collocationOptimized(false)`. Finally, for proxies created using `propertyToProxy` (see Section 32.11.1), the property *`name`*`.CollocationOptimized` configures the default setting for the proxy.

32.22 Automatic Retries

Ice may automatically retry a proxy invocation after a failure. This is a powerful feature that, when used in the proper situations, can significantly improve the robustness of your application without any additional programming effort. The retry facility is governed by one overriding principle: always respect *at-most-once* semantics. As explained on page 14, at-most-once semantics dictate that the Ice run time in the client must never retry a failed proxy invocation unless Ice guarantees that the server has not already received the request, or unless the application declares that it is safe for Ice to violate at-most-once semantics for the request.

To understand the importance of obeying at-most-once semantics, consider the following Slice definition:

```
interface Account {
    long withdraw(long amount);
};
```

The `withdraw` operation removes funds from an account. If an invocation of `withdraw` fails, automatically retrying the request introduces the risk of a duplicate withdrawal unless Ice is absolutely sure that the server has not already executed the request.

The remainder of this section examines automatic retries in more detail.

32.22.1 Request Failure

Ice considers a request to have failed if any of the following conditions are true:

- A connection could not be established
- A connection was lost before the reply was received
- A timeout expired
- An exception occurred while sending the request or receiving the reply
- An error occurred in the server while dispatching the request that causes the server to return an `UnknownException` or `RequestFailedException`

NOTE: Ice considers an invocation that results in a user exception to be successful and therefore excludes it from consideration for automatic retries.

Ice must determine the answers to several questions to decide whether to retry a failed request:

1. What kind of error caused the request to fail?

   Ice does not bother retrying a request if it knows the same error is going to occur again. For example, Ice never retries an invocation that raises a `MarshalException`, which indicates that there was a problem while encoding or decoding a message. Retrying such an invocation is unlikely to change the outcome.

   Ice also never retries exceptions that derive from `RequestFailedException` because they indicate a permanent failure. One such subclass is `OperationNotExistException`, whose occurrence signals a serious problem in the application. For instance, it might mean that the client and server are using incompatible Slice definitions, or that the client is trying to invoke operations on the wrong object. The exception to this rule is `ObjectNotExistException`, which Ice does consider to be worthy of retry if the proxy in question is indirect (see page 11) because it gives an application the ability to transparently migrate an Ice object.

   In addition to user exceptions and subclasses of `RequestFailedException`, a server can also return an instance of `UnknownException`, `UnknownLocalException`, or `UnknownUserException` to indicate that it encountered an unex-

pected exception while dispatching the request. These exceptions *are* eligible for retry.

2. When did the error occur?

   If the error is still a candidate for retry, Ice needs to know whether the server has received the request. Naturally, the Ice run time in the client cannot possibly know that information until the server confirms it by sending a reply. However, to be conservative Ice assumes that the server has received the request as soon as Ice has written the *entire* protocol message to the client's local transport buffers. If the error occurred before Ice managed to write the complete message, retrying the request would not violate at-most-once semantics.

   The Ice run time in the server also has the ability to notify the client that a request was not dispatched and therefore that it is safe for the Ice run time in the client to retry the request without violating at-most-once semantics. For example, this situation can occur when the server is shutting down while there are pending requests that have yet to be executed. Sending this notification allows a client to transparently fail over to another server.

3. Does the application require strict adherence to at-most-once semantics for this request?

   An application can grant permission for Ice to violate at-most-once semantics for certain Slice operations by marking them as *idempotent*, causing Ice to retry a request that otherwise would be ineligible because the server has already received it. Section 32.22.2 discusses idempotent operations in more detail.

If Ice determines that an invocation cannot be retried, it raises the exception that caused the request failure to the application. On the other hand, if Ice does retry the invocation and the subsequent retries also fail, Ice raises the *last* exception to the application. For example, if the first attempt fails with `ConnectionRefusedException` and the retry fails with `ConnectTimeoutException`, the invocation raises `ConnectTimeoutException` to the application.

32.22.2 Idempotent Operations

Annotating a Slice operation with the `idempotent` keyword notifies Ice that it can safely violate at-most-once semantics:

```
interface Account {
    long withdraw(long amount);
    idempotent long getBalance();
};
```

Although `withdraw` clearly requires the stricter treatment, there is no harm in automatically retrying the `getBalance` operation even if the server executes the same request more than once.

In general, "read-only" operations are good candidates for the `idempotent` keyword whereas many mutating operations are not. However, the risk of duplicate requests is acceptable even for some kinds of mutating operations:

```
interface Account {
    long withdraw(long amount);
    idempotent long getBalance();
    idempotent void changeAddress(string newAddress);
};
```

Here we have marked `changeAddress` as idempotent because executing the request twice has the same effect as executing it only once.

The benefit of the `idempotent` keyword and the associated relaxation of retry semantics is that an invocation that otherwise might have raised an exception has at least one more chance to succeed. Furthermore, the application does not need to initiate the retry, and in fact the retry activities are completely transparent: if a subsequent retry succeeds, the application receives its results as if nothing went wrong. The invocation only raises an exception once Ice has reached its configured retry limits.

32.22.3 Configuring Retries

Retry Intervals

The `Ice.RetryIntervals` property configures the retry behavior for a communicator and affects invocations on every proxy created by that communicator. (Retry behavior cannot be configured on a per-proxy basis.) The value of this property consists of a series of integers separated by whitespace. The number of integers determines how many retry attempts Ice makes, and the value of each entry represents a delay in milliseconds. If this property is not defined, the default behavior is to retry once immediately after the first failure, which is equivalent to the following property definition:

```
Ice.RetryIntervals=0
```

You may want a more elaborate configuration for your application, such as a gradual increase in the delay between retries:

```
Ice.RetryIntervals=0 100 500 1000
```

With this setting, Ice retries immediately as in the default case. If the first retry attempt also fails, Ice waits 100 milliseconds before trying again, then 500 milliseconds, and finally tries one more time after waiting one second.

In some situations you may need to disable retries completely. For example, an application might implement its own retry logic and therefore require immediate notification when a failure occurs. Clients that establish a session with a Glacier2 router also need to disable retries (see Section 42.3.4). To prevent automatic retries, use a value of `-1`:

```
Ice.RetryIntervals=-1
```

Retry Logging

To monitor Ice's retry activities, configure your program with the property `Ice.Trace.Retry` set to a non-zero value:

```
Ice.Trace.Retry=1
```

When retry tracing is enabled, Ice logs a message each time it attempts a retry; the log message includes a description of the exception that prompted the retry. Ice also logs a message when it reaches the retry limit.

You can configure Ice to log even more information about retries by setting the property to `2`:

```
Ice.Trace.Retry=2
```

This setting prompts Ice to include additional details about connections and endpoints.

32.22.4 Timeouts

If a proxy invocation fails due to a timeout, the application must be prepared for Ice to raise a `TimeoutException` (see Section 32.13). However, a developer that is testing timeouts in an application may be initially confused to discover that it is taking twice as long as expected for Ice to raise the `TimeoutException`. Automatic retries are usually the reason for this situation.

For example, suppose that a proxy is configured with a one-second timeout and automatic retries are enabled with the default setting. If an invocation on that proxy fails due to a timeout and Ice determines that the invocation is eligible for

retry (using the criteria in Section 32.22.1), Ice immediately tries the invocation again and waits for another timeout period to expire before finally raising `TimeoutException`. From the application's perspective, the invocation fails after approximately two seconds.

Consequently, you can compute an approximate worst-case timeout value as follows:

$$T = t * (N + 1) + D$$

where t is the timeout value, N is the number of retry intervals, and D is the sum of the retry intervals (the total delay between retries). For example, consider our example from Section 32.22.3 again:

```
Ice.RetryIntervals=0 100 500 1000
```

Using this configuration with a one-second timeout, our approximate worst-case timeout is `1 * 5 + 1.6 = 6.6` seconds.

32.22.5 Connections

The behavior of automatic retries is intimately tied to the presence (and absence) of connections. This section describes the errors that cause Ice to close connections, and provides more details about how connections influence retries.

Connection Errors

Ice automatically closes a connection in response to certain fatal error conditions. Of these, the one that is the most likely to affect Ice applications is a timeout (see Section 32.22.4). Other errors that prompt Ice to close a connection include the following:

- a socket error occurs while performing I/O on the connection
- receiving an improperly formatted message
- dispatching an operation to a Java servant raises `OutOfMemoryError` or `AssertionError`

When Ice closes a connection in response to one of these errors, all other outstanding requests on the same connection also fail and may be retried if eligible.

Connection Status

One factor that influences retry behavior is the status of the connection on which the failed request was attempted. If the failure caused Ice to close the connection

(as discussed in the previous section), or if the request failed because Ice could not establish a connection, Ice must try to obtain another connection before it can retry the request. Section 36.3 describes the semantics of connection establishment.

It is also important to understand that Ice may not retry the invocation on the original endpoint *even if the connection that was used for the initial request remains open*. The retry behavior in this case depends on several criteria:

- whether the proxy caches its connection (see Section 36.3.4)
- whether the proxy contains multiple endpoints
- whether other connections exist to any of the proxy's endpoints
- the proxy's configured endpoint selection type (see Section 36.3.1)

Generally speaking, you must configure your application carefully if you need fine-grained control over Ice's retry behavior.

32.23 Dispatch Interceptors

A dispatch interceptor is a server-side mechanism that allows you to intercept incoming client requests before they are given to a servant. The interceptor can examine the incoming request; in particular, it can see whether the request dispatch is collocation-optimized and examine the `Current` information for the request (which provides access to the operation name, object identity, and so on).

A dispatch interceptor can dispatch a request to a servant and check whether the dispatch was successful; if not, the interceptor can choose to retry the dispatch. This functionality is useful to automatically retry requests that have failed due to a recoverable error condition, such as a database deadlock exception. (Freeze uses dispatch interceptors for this purpose in its evictor implementation—see Section 39.3.)

32.23.1 Dispatch Interceptor API

Dispatch interceptors are not defined in Slice, but are provided as an API that is specific to each programming language. The remainder of this section presents the interceptor API for C++; for Java and .NET, the API is analogous, so we do not show it here.

In C++, a dispatch interceptor has the following interface:

```
namespace Ice {
    class DispatchInterceptor : public virtual Object {
    public:
        virtual DispatchStatus dispatch(Request&) = 0;
    };

    typedef IceInternal::Handle<DispatchInterceptor>
        DispatchInterceptorPtr;
}
```

Note that a `DispatchInterceptor` *is-a* `Object`, that is, you use a dispatch interceptor as a servant.

To create a dispatch interceptor, you must derive a class from `DispatchInterceptor` and provide an implementation of the pure virtual `dispatch` function. The job of `dispatch` is to pass the request to the servant and to return a dispatch status, defined as follows:

```
namespace Ice {
    enum DispatchStatus {
        DispatchOK, DispatchUserException, DispatchAsync
    };
}
```

The enumerators indicate how the request was dispatched:

- `DispatchOK`

  The request was dispatched synchronously and completed without an exception.

- `DispatchUserException`

  The request was dispatched synchronously and raised a user exception.

- `DispatchAsync`

  The request was dispatched successfully as an asynchronous request; the result of the request is not available to the interceptor because the result is delivered to the AMD callback when the request completes.

The Ice run time provides basic information about the request to the `dispatch` function in form of a `Request` object:

```
namespace Ice {
    class Request {
    public:
        virtual bool isCollocated();
        virtual const Current& getCurrent();
    };
}
```

- `isCollocated` returns true if the dispatch is directly into the target servant as a collocation-optimized dispatch (see Section 32.21). If the dispatch is not collocation-optimized, the function returns false.
- `getCurrent` provides access to the `Current` object for the request (see Section 32.6), which provides access to information about the request, such as the object identity of the target object, the object adapter used to dispatch the request, and the operation name.

Note that `Request`, for performance reasons, is *not* thread-safe. This means that you must not concurrently dispatch from different threads using the same `Request` object. (Concurrent dispatch for different requests does not cause any problems.)

To use a dispatch interceptor, you instantiate your derived class and register it as a servant with the Ice run time in the usual way, by adding the interceptor to the ASM, or returning the interceptor as a servant from a call to `locate` on a servant locator.

Objective-C Mapping

The Objective-C mapping in Ice Touch does not support AMD, therefore the return type of the `dispatch` method is simplified to a boolean:

```
@protocol ICEDispatchInterceptor <ICEObject>
-(BOOL) dispatch:(id<ICERequest>)request;
@end
```

A return value of `YES` is equivalent to `DispatchOK` and indicates that the request completed without an exception. A return value of `NO` is equivalent to `DispatchUserException`.

Similarly, the `ICERequest` protocol omits the `isCollocated` method because collocation optimization is not supported.

32.23.2 Using a Dispatch Interceptor

Your implementation of the `dispatch` function must dispatch the request to the actual servant. Here is a very simple example implementation of an interceptor that dispatches the request to the servant passed to the interceptor's constructor:

```
class InterceptorI : public Ice::DispatchInterceptor {
public:
    InterceptorI(const Ice::ObjectPtr& servant)
        : _servant(servant) {}

    virtual Ice::DispatchStatus dispatch(Ice::Request& request) {
        return _servant->ice_dispatch(request);
    }

    Ice::ObjectPtr _servant;
};
```

Note that our implementation of `dispatch` calls `ice_dispatch` on the target servant to dispatch the request. `ice_dispatch` does the work of actually (synchronously) invoking the operation.

Also note that `dispatch` returns whatever is returned by `ice_dispatch`. For synchronous dispatch, you should always implement your interceptor in this way and not change this return value.

We can use this interceptor to intercept requests to a servant of any type as follows:

```
ExampleIPtr servant = new ExampleI;
Ice::DispatchInterceptorPtr interceptor =
    new InterceptorI(servant);
adapter->add(interceptor,
             communicator->stringToIdentity("ExampleServant"));
```

Note that, because dispatch interceptor *is-a* servant, this means that the servant to which the interceptor dispatches need not be the actual servant. Instead, it could be another dispatch interceptor that ends up dispatching to the real servant. In other words, you can chain dispatch interceptors; each interceptor's `dispatch` function is called until, eventually, the last interceptor in the chain dispatches to the actual servant.

A more interesting use of a dispatch interceptor is to retry a call if it fails due to a recoverable error condition. Here is an example that retries a request if it raises a local exception defined in Slice as follows:

```
local exception DeadlockException { /* ... */ };
```

Note that this is a `local` exception. Local exceptions that are thrown by the servant propagate to `dispatch` and can be caught there. A database might throw such an exception if the database detects a locking conflict during an update. We can retry the request in response to this exception using the following `dispatch` implementation:

```
virtual Ice::DispatchStatus dispatch(Ice::Request& request) {
    while (true) {
        try {
            return _servant->ice_dispatch(request);
        } catch (const DeadlockException&) {
            // Happens occasionally
        }
    }
}
```

Of course, a more robust implementation might limit the number of retries and possibly add a delay before retrying.
You can also retry an asynchronous dispatch. In this case, each asynchronous dispatch attempt creates a new AMD callback object.

- If the response for the retried request has been sent already, the interceptor receives a `ResponseSentException`. your interceptor must either not handle (or rethrow this exception) or return `DispatchAsync`.
- If the response for the request has not been sent yet, the Ice run time ignores any call to `ice_response` or `ice_exception` on the old AMD callback.

If an operation throws a user exception (as opposed to a local exception), the user exception cannot be caught by `dispatch` as an exception but, instead, is reported by the return value of `ice_dispatch`: a return value of `DispatchUserException` indicates that the operation raised a user exception. You can retry a request in response to a user exception as follows:

```
virtual Ice::DispatchStatus dispatch(Ice::Request& request) {
    Ice::DispatchStatus d;
    do {
        d = _servant->ice_dispatch(request);
    } while (d == Ice::DispatchUserException);
    return d;
}
```

This is fine as far as it goes, but not particularly useful because the preceding code retries if *any* kind of user exception is thrown. However, typically, we want to retry a request only if a *specific* user exception is thrown. The problem here is that the `dispatch` function does not have direct access to the actual exception that

was thrown—all it knows is that *some* user exception was thrown, but not which one.

To retry a request for a specific user exception, you need to implement your servants such that they leave some "footprint" behind if they throw the exception of interest. This allows your request interceptor to test whether the user exception should trigger a retry. There are various techniques you can use to achieve this. For example, you can use thread-specific storage to test a retry flag that is set by the servant if it throws the exception or, if you use transactions, you can attach the retry flag to the transaction context. However, doing so is more complex; the intended use case is to permit retry of requests in response to local exceptions, so we suggest you retry requests only for local exceptions.

The most common use case for a dispatch interceptor is as a default servant. Rather than having an explicit interceptor for individual servants, you can return a dispatch interceptor as the servant from a call to `locate` on a servant locator (see 32.9.2). You can then choose the "real" servant to which to dispatch the request inside `dispatch`, prior to calling `ice_dispatch`. This allows you to intercept and selectively retry requests based on their outcome, which cannot be done using a servant locator.

32.24 C++ Strings and Character Encoding

On the wire, Ice transmits all strings as Unicode strings in UTF-8 encoding (see Chapter 37). For languages other than C++, Ice uses strings in their language-native Unicode representation and converts automatically to and from UTF-8 for transmission, so applications can transparently use characters from non-English alphabets.

However, for C++, how strings are represented inside a process depends on which mapping is chosen for a particular string, the default mapping to `std::string`, or the alternative mapping to `std::wstring` (see Section 6.6.1) as well as the platform.[13] This section explains how strings are encoded by the Ice for C++ run time, and how you can achieve automatic conversion of strings in their native representation to and from UTF-8.[14]

By default, the Ice run time encodes strings as follows:

13. The explanations that follow are relevant only for C++. See Sections 32.24.8 and 32.24.9 for string conversion for other languages.

- Narrow strings (that is, strings mapped to `std::string`) are presented to the application in UTF-8 encoding and, similarly, the application is expected to provide narrow strings in UTF-8 encoding to the Ice run time for transmission.

  With this default behavior, the application code is responsible for converting between the native codeset for 8-bit characters and UTF-8. For example, if the native codeset is ISO Latin-1, the application is responsible for converting between UTF-8 and narrow (8-bit) characters in ISO Latin-1 encoding.

  Also note that the default behavior does not require the application to do anything if it only uses characters in the ASCII range. (This is because a string containing only characters in the (7-bit) ASCII range is also a valid UTF-8 string.)
- Wide strings (that is, strings mapped to `std::wstring`) are automatically encoded as Unicode by the Ice run time as appropriate for the platform. For example, for Windows, the Ice run time converts between UTF-8 and UTF-16 in little-endian representation whereas, for Linux, the Ice run time converts between UTF-8 and UTF-32 in the endian-ness appropriate for the host CPU.

  With this default behavior, wide strings are transparently converted between their on-the-wire representation and their native C++ representation as appropriate, so application code need not do anything special. (The exception is if an application uses a non-Unicode encoding, such as Shift-JIS, as its native `wstring` codeset.)

32.24.1 Installing String Converters

The default behavior of the run time can be changed by providing application-specific string converters. If you install such converters, all Slice strings will be passed to the appropriate converter when they are marshaled and unmarshaled. Therefore, the string converters allow you to convert all strings transparently into their native representation without having to insert explicit conversion calls whenever a string crosses a Slice interface boundary.

You can install string converters on a per-communicator basis when you create a communicator by setting the `stringConverter` and `wstringConverter` members of the `InitializationData` structure (see Section 32.3). Any

14. See the `demo` directory in the Ice for C++ distribution for an example of how to use string converters.

strings that use the default (`std::string`) mapping are passed through the specified `stringConverter`, and any strings that use the wide (`std::wstring`) mapping are passed through the specified `wstringConverter`.

The string converters are defined as follows:

```
namespace Ice {

class UTF8Buffer {
public:
    virtual Byte* getMoreBytes(size_t howMany,
                               Byte* firstUnused) = 0;
    virtual ~UTF8Buffer() {}
};

template<typename charT>
class BasicStringConverter : public IceUtil::Shared {
public:
    virtual Byte*
        toUTF8(const charT* sourceStart, const charT* sourceEnd,
               UTF8Buffer&) const = 0;

    virtual void fromUTF8(const Byte* sourceStart,
                          const Byte* sourceEnd,
                          std::basic_string<charT>& target) const;
};

typedef BasicStringConverter<char> StringConverter;
typedef IceUtil::Handle<StringConverter> StringConverterPtr;

typedef BasicStringConverter<wchar_t> WstringConverter;
typedef IceUtil::Handle<WstringConverter> WstringConverterPtr;

}
```

As you can see, both narrow and wide string converters are simply templates with either a narrow or a wide character (`char` or `wchar_t`) as the template parameter.

32.24.2 Converting to UTF-8

If you have a string converter installed, the Ice run time calls the `toUTF8` function whenever it needs to convert a native string into UTF-8 representation for transmission. The `sourceStart` and `sourceEnd` pointers point at the first

byte and one-beyond-the-last byte of the source string, respectively. The implementation of `toUTF8` must return a pointer to the first unused byte following the converted string.

Your implementation of `toUTF8` must allocate the returned string by calling the `getMoreBytes` member function of the `UTF8Buffer` class that is passed as the third argument. (`getMoreBytes` throws a `MemoryLimitException` if it cannot allocate enough memory.) The `firstUnused` parameter must point at the first unused byte of the allocated memory region. You can make several calls to `getMoreBytes` to incrementally allocate memory for the converted string. If you do, `getMoreBytes` may relocate the buffer in memory. (If it does, it copies the part of the string that was converted so far into the new memory region.) The function returns a pointer to the first unused byte of the (possibly relocated) memory.

Conversion with `toUTF8` can fail because `getMoreBytes` can cause the message size to exceed `Ice.MessageSizeMax`. In this case, you should let the `MemoryLimitException` thrown by `getMoreBytes` propagate to the caller.

Conversion can also fail because the encoding of the source string is internally incorrect. In that case, you should throw a `StringConversionFailed` exception from `toUTF8`.

After it has marshaled the returned string into an internal marshaling buffer, the Ice run time deallocates the string.

32.24.3 Converting from UTF-8

During unmarshaling, the Ice run time calls the `fromUTF8` member function on the corresponding string converter. The function converts a UTF-8 string into its native form as a `std::string`. (The string into which the function must place the converted characters is passed to `fromUTF8` as the `target` parameter.)

32.24.4 String Parameters in Local Calls

In C++, and indirectly in Python, Ruby, and PHP, all Ice local APIs are narrow-string based, meaning you could not for example recompile `Properties.ice` to get property names and values as wide strings.

Installing a narrow-string converter could cause trouble for these local calls if UTF-8 conversion occurs in the underlying implementation. For example, the `stringToIdentity` operation creates an intermediary UTF-8 string. If this string

contains characters that are not in your native codeset (as determined by the narrow-string converter), the `stringToIdentity` call will fail.

Likewise, when Ice reads properties from a configuration file, it converts the input (UTF-8 characters) into native strings. This conversion can also fail if the native encoding cannot convert some characters.

Most strings in local calls are never problematic because Ice does not perform any conversion, for example:

- adapter names in `createObjectAdapter`
- property names and values in `Properties`
- `ObjectAdapter::createProxy`, where the identity conversion occurs only when the proxy is marshaled

Finally, consider the Slice type `Ice::Context`, which is mapped in C++ as a `map<string, string>`. The mapping for `Context` cannot be changed to `map<wstring, wstring>`, therefore you cannot send or receive any context entry that is not in your narrow-string native encoding when a narrow-string converter is installed.

32.24.5 Built-In String Converters

Ice provides three string converters to cover common conversion requirements:

- `UnicodeWstringConverter`

  This is a string converter that converts between Unicode wide strings and UTF-8 strings. Unless you install a different string converter, this is the default converter that is used for wide strings.

- `IconvStringConverter` (Linux and Unix only)

  This is a string converter that converts strings using the Linux and Unix `iconv` conversion facility (see Section 32.24.7). It can be used to convert either wide or narrow strings.

- `WindowsStringConverter` (Windows only)

  This string converter converts between multi-byte and UTF-8 strings and uses `MultiByteToWideChar` and `WideCharToMultiByte` for its implementation.

These string converters are defined in the `Ice` namespace.

32.24.6 Convenience Functions

The `Ice` namespace provides four convenience functions that make it easy to convert strings to and from UTF-8:

```
std::string
nativeToUTF8(const Ice::StringConverterPtr&, const std::string&);

std::string
nativeToUTF8(const Ice::CommunicatorPtr&, const std::string&);

std::string
UTF8ToNative(const Ice::StringConverterPtr&, const std::string&);

std::string
UTF8ToNative(const Ice::CommunicatorPtr&, const std::string&);
```

The overloads allow you to either use the string converter that is configured on a communicator or to explicitly pass a specific string converter that performs the conversion.

32.24.7 The `iconv` String Converter

For Linux and Unix platforms, Ice provides an `IconvStringConverter` template class that uses the `iconv` conversion facility to convert between the native encoding and UTF-8. The only member function of interest is the constructor:

```
template<typename charT>
class IconvStringConverter
    : public Ice::BasicStringConverter<charT>
{
public:
    IconvStringConverter(const char* = nl_langinfo(CODESET));

    // ...
};
```

To use this string converter, you specify whether the conversion you want is for narrow or wide characters via the template argument, and you specify the corresponding native encoding with the constructor argument. For example, to create a converter that converts between ISO Latin-1 and UTF-8, you can instantiate the converter as follows:

```
InitializationData id;
id.stringConverter = new IconvStringConverter<char>("ISO-8859-1");
```

Similarly, to convert between the internal wide character encoding and UTF-8, you can instantiate a converter as follows:

```
InititializationData id;
id.stringConverter = new IconvStringConverter<wchar_t>("WCHAR_T");
```

The string you pass to the constructor must be one of the values returned by **`iconv -l`**, which lists all the available character encodings for your machine.

Using the `IconvStringConverter` template makes it easy to install code converters for any available encoding without having to explicitly write (or call) conversion routines whose implementation is typically non-trivial.

32.24.8 The Ice String Converter Plug-In

The Ice run time includes a plug-in that supports conversion between UTF-8 and native encodings on Unix and Windows platforms. You can use this plug-in to install converters for narrow and wide strings into the communicator of an existing program. This feature is primarily intended for use in scripting language extensions such as Ice for Python; if you need to use string converters in your C++ application, we recommend using the technique described in Section 32.24.1 instead.

Note that an application must be designed to operate correctly in the presence of a string converter. A string converter assumes that it converts strings in the native encoding into the UTF-8 encoding, and vice versa. An application that performs its own conversions on strings that cross a Slice interface boundary can cause encoding errors when those strings are processed by a converter.

Installing the Plug-In

You can install the plug-in using a configuration property like the one shown below:

```
Ice.Plugin.Converter=Ice:createStringConverter
    iconv=encoding[,encoding] windows=code-page
```

You can use any name you wish for the plug-in; in this example, we used `Converter`. The first component of the property value represents the plug-in's entry point, which includes the abbreviated name of the shared library or DLL (`Ice`) and the name of a factory function (`createStringConverter`).

The plug-in accepts the following arguments:

- `iconv=`*`encoding`*`[,`*`encoding`*`]`

  This argument is optional on Unix platforms and ignored on Windows platforms. If specified, it defines the `iconv` names of the narrow string encoding and the optional wide-string encoding. If this argument is not specified, the plug-in installs a narrow string converter that uses the default locale-dependent encoding.

- `windows=`*`code-page`*

  This argument is required on Windows platforms and ignored on Unix platforms. The *`code-page`* value represents a code page number, such as `1252`.

The plug-in's argument semantics are designed so that the same configuration property can be used on both Windows and Unix platforms, as shown in the following example:

```
Ice.Plugin.Converter=Ice:createStringConverter iconv=ISO8859-1
 windows=1252
```

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for C++ run time:

```
Ice.Plugin.Converter.cpp=Ice:createStringConverter iconv=ISO8859-1
 windows=1252
```

Refer to Appendix D for more information on the `Ice.Plugin` properties.

32.24.9 Dynamically Installing Custom String Converters

If the string converter plug-in described in Section 32.24.8 does not satisfy your requirements, you can implement your own solution with help from the `StringConverterPlugin` class:

```
namespace Ice {
class StringConverterPlugin : public Ice::Plugin {
public:

    StringConverterPlugin(const CommunicatorPtr& communicator,
                          const StringConverterPtr&,
                          const WstringConverterPtr& = 0);

    virtual void initialize();
```

```
    virtual void destroy();
};
}
```

The converters are installed by the `StringConverterPlugin` constructor (you can supply an argument of `0` for either converter if you do not wish to install it). The `initialize` and `destroy` methods are empty, but you can subclass `StringConverterPlugin` and override these methods if necessary.

In order to create a string converter plug-in, you must do the following:

- Define and export a "factory function" that returns an instance of `StringConverterPlugin` (see Section 32.25.1).
- Implement the converter(s) that you will pass to the `StringConverterPlugin` constructor, or use the ones included with Ice (see Appendix F).
- Package your code into a shared library or DLL.

To install your plug-in, use a configuration property like the one shown below:

```
Ice.Plugin.MyConverterPlugin=myconverter:createConverter ...
```

The first component of the property value represents the plug-in's entry point, which includes the abbreviated name of the shared library or DLL (`myconverter`) and the name of a factory function (`createConverter`).

If the configuration file containing this property is shared by programs in multiple implementation languages, you can use an alternate syntax that is loaded only by the Ice for C++ run time:

```
Ice.Plugin.MyConverterPlugin.cpp=myconverter:createConverter ...
```

Refer to Appendix D for more information on the `Ice.Plugin` properties.

32.25 Developing a Plug-In

Ice supports a plug-in facility that allows you to add new features and install application-specific customizations. Plug-ins are defined using configuration properties and loaded dynamically by the Ice run time, making it possible to install a plug-in into an existing program without modification.

Ice uses the plug-in facility to implement some of its own features. Most well-known is IceSSL, a plug-in that adds a secure transport for Ice communication (see Chapter 41). Other examples include the logger plug-in (see Section 32.19.4) and the string converter plug-in (see Section 32.24.8).

This section describes the plug-in facility in more detail and demonstrates how to implement an Ice plug-in.

32.25.1 Plug-In API

The plug-in facility defines a local Slice interface that all plug-ins must implement:

```
module Ice {
local interface Plugin {
    void initialize();
    void destroy();
};
};
```

The lifecycle of an Ice plug-in is structured to accommodate dependencies between plug-ins, such as when a logger plug-in needs to use IceSSL for its logging activities. Consequently, a plug-in object's lifecycle consists of four phases:

- Construction

  The Ice run time uses a language-specific factory API for instantiating plug-ins. During construction, a plug-in can acquire resources but must not spawn new threads or perform activities that depend on other plug-ins.

- Initialization

  After all plug-ins have been constructed, the Ice run time invokes `initialize` on each plug-in. The order in which plug-ins are initialized may be specified using a configuration property (see Section 32.25.3), otherwise the order is undefined. If a plug-in has a dependency on another plug-in, you must configure the Ice run time so that initialization occurs in the proper order. In this phase it is safe for a plug-in to spawn new threads; it is also safe for a plug-in to interact with other plug-ins and use their services, as long as those plug-ins have already been initialized.

  If `initialize` raises an exception, the Ice run time invokes `destroy` on all plug-ins that were successfully initialized (in the reverse order of initialization) and raises the original exception to the application.

- Active

  The active phase spans the time between initialization and destruction. Plug-ins must be designed to operate safely in the context of multiple threads.

- Destruction

  The Ice run time invokes `destroy` on each plug-in in the reverse order of initialization.

This lifecycle is repeated for each new communicator that an application creates and destroys.

C++ Factory

In C++, the plug-in factory is an exported function with C linkage having the following signature:

```
extern "C"
{
ICE_DECLSPEC_EXPORT Ice::Plugin*
functionName(const Ice::CommunicatorPtr& communicator,
             const std::string& name,
             const Ice::StringSeq& args);
}
```

You can define the function with any name you wish. We recommend that you use the `ICE_DECLSPEC_EXPORT` macro to ensure that the function is exported correctly on all platforms. Since the function uses C linkage, it must return the plug-in object as a regular C++ pointer and not as an Ice smart pointer. Furthermore, the function must not raise C++ exceptions; if an error occurs, the function must return zero.

The arguments to the function consist of the communicator that is in the process of being initialized, the name assigned to the plug-in, and any arguments that were specified in the plug-in's configuration.

Java Factory

In Java, a plug-in factory must implement the `Ice.PluginFactory` interface:

```
package Ice;

public interface PluginFactory {
    Plugin create(Communicator communicator,
                  String name,
                  String[] args);
}
```

The arguments to the `create` method consist of the communicator that is in the process of being initialized, the name assigned to the plug-in, and any arguments that were specified in the plug-in's configuration.

The `create` method can return `null` to indicate that a general error occurred, or it can raise `PluginInitializationException` to provide more detailed information. If any other exception is raised, the Ice run time wraps it inside an instance of `PluginInitializationException`.

.NET Factory

In .NET, a plug-in factory must implement the `Ice.PluginFactory` interface:

```
namespace Ice {
    public interface PluginFactory
    {
        Plugin create(Communicator communicator,
                      string name,
                      string[] args);
    }
}
```

The arguments to the `create` method consist of the communicator that is in the process of being initialized, the name assigned to the plug-in, and any arguments that were specified in the plug-in's configuration.

The `create` method can return `null` to indicate that a general error occurred, or it can raise `PluginInitializationException` to provide more detailed information. If any other exception is raised, the Ice run time wraps it inside an instance of `PluginInitializationException`.

32.25.2 Plug-In Configuration

Plug-ins are installed using a configuration property of the following form:

```
Ice.Plugin.Name=entry_point [arg ...]
```

In most cases you can assign an arbitrary name to a plug-in. In the case of IceSSL, however, the plug-in requires that its name be `IceSSL`.

The value of *`entry_point`* is a language-specific representation of the plug-in's factory. In C++, it consists of the name of the shared library or DLL containing the factory function, along with the name of the factory function. In Java, the entry point is the name of the factory class, while in .NET the entry point also includes the assembly.

The language-specific nature of plug-in properties can present a problem when applications that are written in multiple implementation languages attempt

to share a configuration file. Ice supports an alternate syntax for plug-in properties that alleviates this issue:

```
Ice.Plugin.Name.cpp=...     # C++ plug-in
Ice.Plugin.Name.java=...    # Java plug-in
Ice.Plugin.Name.clr=...     # .NET (Common Language Runtime) plug-in
```

Plug-in properties having a suffix of `.cpp`, `.java`, or `.clr` are loaded only by the appropriate Ice run time and ignored by others.

Refer to Appendix D for more information on these properties.

32.25.3 Advanced Topics

This section discusses additional aspects of the Ice plug-in facility that may be of use to applications with special requirements.

Plug-In Dependencies

If a plug-in has a dependency on another plug-in, you must ensure that Ice initializes the plug-ins in the proper order. Suppose that a custom logger implementation depends on IceSSL; for example, the logger may need to transmit log messages securely to another server. We start with the following C++ configuration:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
Ice.Plugin.MyLogger=MyLogger:createMyLogger
```

The problem with this configuration is that it does not specify the order in which the plug-ins should be loaded and initialized. If the Ice run time happens to initialize `MyLogger` first, the plug-in's `initialize` method will fail if it attempts to use the services of the uninitialized IceSSL plug-in.

To remedy the situation, we need to add one more property:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
Ice.Plugin.MyLogger=MyLogger:createMyLogger
Ice.PluginLoadOrder=IceSSL, MyLogger
```

Using the `Ice.PluginLoadOrder` property we can guarantee that the plug-ins are loaded in the correct order. Appendix D describes this property in more detail.

The Plug-In Manager

`PluginManager` is the name of an internal Ice object that is responsible for managing all aspects of Ice plug-ins. This object supports a Slice interface of the

same name, and an application can obtain a reference to this object using the following communicator operation:

```
module Ice {
local interface Communicator {
    PluginManager getPluginManager();
    // ...
};
};
```

The `PluginManager` interface offers three operations:

```
module Ice {
local interface PluginManager {
    void initializePlugins();
    Plugin getPlugin(string name);
    void addPlugin(string name, Plugin pi);
};
};
```

The `initializePlugins` operation is used in special cases when an application needs to manually initialize one or more plug-ins, as discussed in the next section.

The `getPlugin` operation returns a reference to a specific plug-in. The `name` argument must match an installed plug-in, otherwise the operation raises `NotRegisteredException`. This operation is useful when a plug-in exports an interface that an application can use to query or customize its attributes or behavior.

Finally, `addPlugin` provides a way for an application to install a plug-in directly, without the use of a configuration property.

Delayed Initialization

It is sometimes necessary for an application to manually configure a plug-in prior to its initialization. For example, SSL keys are often protected by a passphrase, but a developer may be understandably reluctant to specify that passphrase in a configuration file because it would be exposed in clear text. The developer would likely prefer to configure the IceSSL plug-in with a password callback instead; however, this must be done before the plug-in is initialized and attempts to load the SSL key. The solution is to configure the Ice run time so that it postpones the initialization of its plug-ins:

```
Ice.InitPlugins=0
```

When this property is set to zero, initializing plug-ins becomes the application's responsibility. The example below demonstrates how to perform this initialization:

```
// C++
Ice::CommunicatorPtr ic = ...
Ice::PluginManagerPtr pm = ic->getPluginManager();
IceSSL::PluginPtr ssl = pm->getPlugin("IceSSL");
ssl->setPasswordPrompt(...);
pm->initializePlugins();
```

After obtaining the IceSSL plug-in and establishing the password callback, the application invokes `initializePlugins` on the plug-in manager object to commence plug-in initialization.

32.26 Custom Class Loaders

Certain features of the Ice for Java run-time necessitate dynamic class loading. Applications with special requirements can supply a custom class loader for Ice to use in the following situations:

- Unmarshaling user exceptions and instances of concrete Slice classes
- Loading Ice plug-ins
- Loading IceSSL certificate verifiers and password callbacks

If an application does not supply a class loader (or if the application-supplied class loader fails to locate a class), the Ice run time attempts to load the class as follows:

- using the system class loader
- using the current thread's class loader
- using the default class loader (that is, by calling `Class.forName`)

Note that an application must install object factories for any abstract Slice classes it might receive, regardless of whether the application also installs a custom class loader. Refer to Section 10.14.4 for more information on factories.

To install a custom class loader, set the `classLoader` member of `Ice.InitializationData` prior to creating a communicator:

```
Ice.InitializationData initData = new Ice.InitializationData();
initData.classLoader = new MyClassLoader();
Ice.Communicator communicator =
    Ice.Util.initialize(args, initData);
```

Section 32.3 discusses communicator initialization in greater detail.

32.27 Summary

In this chapter, we explored the server-side run time in detail. Communicators are the main handle to the Ice run time. They provide access to a number of run time resources and allow you to control the life cycle of a server. Object adapters provide a mapping between abstract Ice objects and concrete servants. Various implementation techniques are at your disposal to control the trade-off between performance and scalability; in particular, servant locators are a central mechanism that permits you to choose an implementation technique that matches the requirements of your application.

Ice provides both oneway and datagram invocations. These provide performance gains in situations where an application needs to provide numerous stateless updates. Batching such invocations permits you to increase performance even further.

The Ice logging mechanism is user extensible, so you can integrate Ice messages into arbitrary logging frameworks, and the `Ice::Stats` interface permits you to collect statistics for network bandwidth consumption.

Finally, even though Ice is location transparent, in the interest of efficiency, collocated invocations do not behave in all respects like remote invocations. You need to be aware of these differences, especially for applications that are sensitive to thread context.

Chapter 33
Facets and Versioning

33.1 Introduction

Facets provide a general-purpose mechanism for non-intrusively extending the type system of an application. This is particularly useful for versioning an application. Section 33.2 introduces the facet concept and presents the relevant APIs. Section 33.3 presents a few traditional approaches to versioning and their problems. Sections 33.4 to 33.6 show how to use facets to implement versioning, and Section 33.7 discusses design choices when adding versioning to a system.

33.2 Concept and APIs

Up to this point, we have presented an Ice object as a single conceptual entity, that is, as an object with a single most-derived interface and a single identity, with the object being implemented by a single servant. However, an Ice object is more

correctly viewed as a collection of one or more sub-objects known as *facets*, as shown in Figure 33.1.

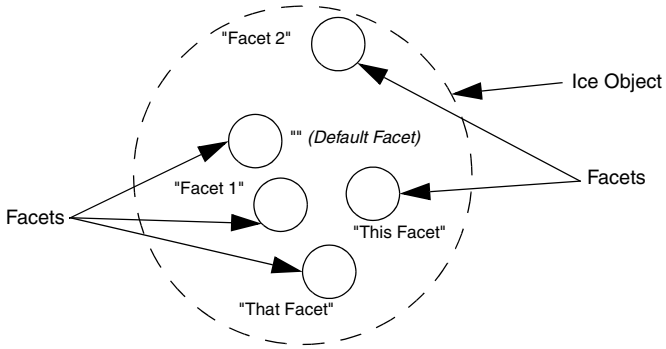


Figure 33.1. An Ice object with five facets sharing a single object identity.

Figure 33.1 shows a single Ice object with five facets. Each facet has a name, known as the *facet name*. Within a single Ice object, all facets must have unique names. Facet names are arbitrary strings that are assigned by the server that implements an Ice object. A facet with an empty facet name is legal and known as the *default facet*. Unless you arrange otherwise, an Ice object has a single default facet; by default, operations that involve Ice objects and servants operate on the default facet.

Note that all the facets of an Ice object share the same single identity, but have different facet names. Recall the definition of `Ice::Current` we saw in Section 32.6 once more:

```
module Ice {
    local dictionary<string, string> Context;

    enum OperationMode { Normal, \Nonmutating, \Idempotent };

    local struct Current {
        ObjectAdapter   adapter;
        Identity        id;
        string          facet;
        string          operation;
        OperationMode   mode;
```

```
        Context         ctx;
        int             requestId;
    };
};
```

By definition, if two facets have the same `id` field, they are part of the same Ice object. Also by definition, if two facets have the same `id` field, their `facet` fields have different values.

Even though Ice objects usually consist of just the default facet, it is entirely legal for an Ice object to consist of facets that all have non-empty names (that is, it is legal for an Ice object not to have a default facet).

Each facet has a single most-derived interface. There is no need for the interface types of the facets of an Ice object to be unique. It is legal for two facets of an Ice object to implement the same most-derived interface.

Each facet is implemented by a servant. All the usual implementation techniques for servants are available to implement facets—for example, you can implement a facet using a servant locator. Typically, each facet of an Ice object has a separate servant, although, if two facets of an Ice object have the same type, they can also be implemented by a single servant (for example, using a default servant, as described in Section 32.9.2).

33.2.1 Server-Side Facet Operations

On the server side, the object adapter offers a number of operations to support facets:

```
namespace Ice {
  dictionary<string, Object> FacetMap;

  local interface ObjectAdapter {
     Object*  addFacet(Object servant, Identity id, string facet);
     Object*  addFacetWithUUID(Object servant, string facet);
     Object   removeFacet(Identity id, string facet);
     Object   findFacet(Identity id, string facet);

     FacetMap findAllFacets(Identity id);
     FacetMap removeAllFacets(Identity id);
     // ...
  };
};
```

These operations have the same semantics as the corresponding “normal” operations (`add`, `addWithUUID`, `remove`, and `find`), but also accept a facet name. The

corresponding "normal" operations are simply convenience operations that supply an empty facet name. For example, `remove(id)` is equivalent to `removeFacet(id, "")`, that is, `remove(id)` operates on the default facet.

`findAllFacets` returns a dictionary of *<facet-name, servant>* pairs that contains all the facets for the given identity.

`removeAllFacets` removes all facets for a given identity from the active servant map, that is, it removes the corresponding Ice object entirely. The operation returns a dictionary of *<facet-name, servant>* pairs that contains all the removed facets.

These operations are sufficient for the server to create Ice objects with any number of facets. For example, assume that we have the following Slice definitions:

```
module Filesystem {
    // ...

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };
};

module FilesystemExtensions {
    // ...

    class DateTime extends TimeOfDay {
        // ...
    };

    struct Times {
        DateTime createdDate;
        DateTime accessedDate;
        DateTime modifiedDate;
    };

    interface Stat {
        idempotent Times getTimes();
    };
};
```

Here, we have a `File` interface that provides operations to read and write a file, and a `Stat` interface that provides access to the file creation, access, and modification time. (Note that the `Stat` interface is defined in a different module and could

also be defined in a different source file.) If the server wants to create an Ice object that contains a `File` instance as the default facet and a `Stat` instance that provides access to the time details of the file, it could use:

```
// Create a File instance.
//
Filesystem::FilePtr file = new FileI;

// Create a Stat instance.
//
FilesystemExctensions::DateTimePtr dt
    = new FilesystemExtensions::DateTime;
FilesystemExtensions::Times times;
times.createdDate = dt;
times.accessedDate = dt;
times.modifiedDate = dt;
FilesystemExtensions::StatPtr stat = new StatI(times);

// Register the File instance as the default facet.
//
Filesystem::FilePrx filePrx = myAdapter->addWithUUID(file);

// Register the Stat instance as a facet with name "Stat".
//
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

The first few lines simply create and initialize a `FileI` and `StatI` instance. (The details of this do not matter here.) All the action is in the last two statements:

```
Filesystem::FilePrx filePrx = myAdapter->addWithUUID(file);
myAdapter->addFacet(stat, filePrx->ice_getIdentity(), "Stat");
```

This registers the `FileI` instance with the object adapter as usual. (In this case, we let the Ice run time generate a UUID as the object identity.) Because we are calling `addWithUUID` (as opposed to `addFacetWithUUID`), the instance becomes the default facet.

The second line adds a facet to the instance with the facet name `Stat`. Note that we call `ice_getIdentity` on the `File` proxy to pass an object identity to `addFacet`. This guarantees that the two facets share the same object identity.

Note that, in general, it is a good idea to use `ice_getIdentity` to obtain the identity of an existing facet when adding a new facet. That way, it is guaranteed that the facets share the same identity. (If you accidentally pass a different identity to `addFacet`, you will not add a facet to an existing Ice object, but instead register a new Ice object; using `ice_getIdentity` makes this mistake impossible.)

33.2.2 Client-Side Facet Operations

On the client side, which facet a request is addressed to is implicit in the proxy that is used to send the request. For an application that does not use facets, the facet name is always empty so, by default, requests are sent to the default facet.

The client can use a `checkedCast` to obtain a proxy for a particular facet. For example, assume that the client obtains a proxy to a `File` instance as shown in Section 33.2.1. The client can cast between the `File` facet and the `Stat` facet (and back) as follows:

```
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Get the Stat facet.
//
FilesystemExtensions::StatPrx stat
    = FilesystemExtensions::StatPrx::checkedCast(file, "Stat");

// Go back from the Stat facet to the File facet.
//
Filesystem::FilePrx file2
    = Filesystem::FilePrx::checkedCast(stat, "");

assert(file2 == file); // The two proxies are identical.
```

This example illustrates that, given any facet of an Ice object, the client can navigate to any other facet by using a `checkedCast` with the facet name.

If an Ice object does not provide the specified facet, `checkedCast` returns null:

```
FilesystemExtensions::StatPrx stat
    = FilesystemExtensions::StatPrx::checkedCast(file, "Stat");

if (!stat) {
    // No Stat facet on this object, handle error...
} else {
    FilesystemExtensions::Times times = stat->getTimes();

    // Use times struct...
}
```

Note that `checkedCast` also returns a null proxy if a facet exists, but the cast is to the wrong type. For example:

```
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Cast to the wrong type.
//
SomeTypePrx prx = SomeTypePrx::checkedCast(file, "Stat");

assert(!prx); // checkedCast returns a null proxy.
```

If you want to distinguish between non-existence of a facet and the facet being of the incorrect type, you can first obtain the facet as type `Object` and then down-cast to the correct type:

```
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Get the facet as type Object.
//
Ice::ObjectPrx obj = Ice::ObjectPrx::checkedCast(file, "Stat");
if (!obj) {
    // No facet with name "Stat" on this Ice object.
} else {
    FilesystemExtensions::StatPrx stat =
        FilesystemExtensions::StatPrx::checkedCast(file);
    if (!stat) {
        // There is a facet with name "Stat", but it is not
        // of type FilesystemExtensions::Stat.
    } else {
        // Use stat...
    }
}
```

This last example also illustrates that

```
StatPrx::checkedCast(prx, "")
```

is *not* the same as

```
StatPrx::checkedCast(prx)
```

The first version explicitly requests a cast to the default facet. This means that the Ice run time first looks for a facet with the empty name and then attempts to down-cast that facet (if it exists) to the type `Stat`.

The second version requests a down-cast that *preserves* whatever facet is currently effective in the proxy. For example, if the `prx` proxy currently holds the

facet name "Joe", then (if `prx` points at an object of type `Stat`) the run time returns a proxy of type `StatPrx` that also stores the facet name "Joe".

It follows that, to navigate between facets, you must always use the two-argument version of `checkedCast`, whereas, to down-cast to another type while preserving the facet name, you must always use the single-argument version of `checkedCast`.

You can always check what the current facet of a proxy is by calling `ice_getFacet`:

```
Ice::ObjectPrx obj = ...;

cout << obj->ice_getFacet() << endl; // Print facet name
```

This prints the facet name. (For the default facet, `ice_getFacet` returns the empty string.)

33.2.3 Exception Semantics

As we pointed out on page 118, `ObjectNotExistException` and `FacetNotExistException` have the following semantics:

- `ObjectNotExistException`

  This exception is raised only if no facets exist at all for a given object identity.
- `FacetNotExistException`

  This exception is raised only if at least one facet exists for a given object identity, but not the specific facet that is the target of an operation invocation.

If you are using servant locators (see Section 32.7) or default servants (Section 32.9.2), you must take care to preserve these semantics. In particular, if you return null from a servant locator's `locate` operation, this appears to the client as an `ObjectNotExistException`. If the object identity for a request is known (that is, there is at least one facet with that identity), but no facet with the specified name exists, you must explicitly throw a `FacetNotExistException` from `activate` instead of simply returning null.

33.3 The Versioning Problem

Once you have developed and deployed a distributed application, and once the application has been in use for some time, it is likely that you will want to make some changes to the application. For example, you may want to add new function-

ality to a later version of the application, or you may want to change some existing aspect of the application. Of course, ideally, such changes are accomplished without breaking already deployed software, that is, the changes should be backward compatible. Evolving an application in this way is generally known as *versioning*.

Versioning is an aspect that previous middleware technologies have addressed only poorly (if at all). One of the purposes of facets is to allow you to cleanly create new versions of an application without compromising compatibility with older, already deployed versions.

33.3.1 Versioning by Addition

Suppose that we have deployed our file system application and want to add extra functionality to a new version. Specifically, let us assume that the original version (version 1) only provides the basic functionality to use files, but does not provide extra information, such as the modification date or the file size. The question is then, how can we upgrade the existing application with this new functionality? Here is a small excerpt of the original (version 1) Slice definitions once more:

```
// Version 1

module Filesystem {
    // ...

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };
};
```

Your first attempt at upgrading the application might look as follows:

```
// Version 2

module Filesystem {
    // ...

    class DateTime extends TimeOfDay {  // New in version 2
        // ...
    };

    struct Times {                      // New in version 2
        DateTime createdDate;
```

```
        DateTime accessedDate;
        DateTime modifiedDate;
    };

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;

        idempotent Times getTimes();    // New in version 2
    };
};
```

Note that the version 2 definition does not change anything that was present in version 1; instead, it only adds two new types and adds an operation to the `File` interface. Version 1 clients can continue to work with both version 1 and version 2 `File` objects because version 1 clients do not know about the `getTimes` operation and therefore will not call it; version 2 clients, on the other hand, can take advantage of the new functionality. The reason this works is that the Ice protocol invokes an operation by sending the operation name as a string on the wire (rather than using an ordinal number or hash value to identify the operation). Ice guarantees that any future version of the protocol will retain this behavior, so it is safe to add a new operation to an existing interface without recompiling all clients.

However, this approach contains a pitfall: the tacit assumption built into this approach is that no version 2 client will ever use a version 1 object. If the assumption is violated (that is, a version 2 client uses a version 1 object), the version 2 client will receive an `OperationNotExistException` when it invokes the new `getTimes` operation because that operation is supported only by version 2 objects.

Whether you can make this assumption depends on your application. In some cases, it may be possible to ensure that version 2 clients will never access a version 1 object, for example, by simultaneously upgrading all server's from version 1 to version 2, or by taking advantage of application-specific constraints that ensure that version 2 clients only contact version 2 objects. However, for some applications, doing this is impractical.

Note that you could write version 2 clients to catch and react to an `OperationNotExistException` when they invoke the `getTimes` operation: if the operation succeeds, the client is dealing with a version 2 object, and if the operation raises `OperationNotExistsException`, the client is dealing with a version 1 object. However, doing this can be rather intrusive to the code, loses static type safety, and is rather inelegant. (There is no other way to tell a version 1 object from a version 2 object because both versions have the same type ID.)

In general, versioning addition makes sense when you need to add an operation or two to an interface, and you can be sure that version 2 clients do not access version 1 objects. Otherwise, other approaches are needed.

33.3.2 Versioning by Derivation

Given the limitations of the preceding approach, you may decide to upgrade the application as follows instead:

```
module Filesystem {     // Version 1
    // ...

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };
};

module FilesystemV2 {   // New in version 2
    // ...

    class DateTime extends TimeOfDay {
        // ...
    };

    struct Times {
        DateTime createdDate;
        DateTime accessedDate;
        DateTime modifiedDate;
    };

    interface File extends Filesystem::File {
        idempotent Times getTimes();
    };
};
```

The idea is to present the new functionality in an interface that is derived from the version 1 interface. The version 1 types are unchanged and the new functionality is presented via new types that are backward compatible: a version 2 `File` object can be passed where a version 1 `File` object is expected because `FilesystemV2::File` is derived from `Filesystem::File`. Even better, if a version 2 component of the system receives a proxy of formal type `Filesystem::File`, it can determine at run time whether the actual run-time type is

`FilesystemV2::File` by attempting a down-cast: if the down-cast succeeds, it is dealing with a version 2 object; if the down-cast fails, it is dealing with a version 1 object. This is essentially the same as versioning by addition, but it is cleaner as far as the type system is concerned because the two different versions can be distinguished via their type IDs.

At this point, you may think that versioning by derivation solves the problem elegantly. Unfortunately, the truth turns out to be a little harsher:

- As the system evolves further, and new versions are added, each new version adds a level of derivation to the inheritance tree. After a few versions, particularly if your application also uses inheritance for its own purposes, the resulting inheritance graph very quickly turns into a complex mess. (This becomes most obvious if the application uses multiple inheritance—after a few versioning steps, the resulting inheritance hierarchy is usually so complex that it exceeds the ability of humans to comprehend it.)

- Real-life versioning requirements are not as simple as adding a new operation to an object. Frequently, versioning requires changes such as adding a field to a structure, adding a parameter to an operation, changing the type of a field or a parameter, renaming an operation, or adding a new exception to an operation. However, versioning by derivation (and versioning by addition) can handle none of these changes.

- Quite often, functionality that is present in an earlier version needs to be removed for a later version (for example, because the older functionality has been supplanted by a different mechanism or turned out to be inappropriate). However, there is no way to *remove* functionality through versioning by derivation. The best you can do is to re-implement a base operation in the derived implementation of an interface and throw an exception. However, the deprecated operation may not have an exception specification, or if it does, the exception specification may not include a suitable exception. And, of course, doing this perverts the type system: after all, if an interface has an operation that throws an exception whenever the operation is invoked, why does the operation exist in the first place?

There are other, more subtle reasons why versioning by derivation is unsuitable in real-life situations. Suffice it to say here that experience has shown the idea to be unworkable: projects that have tried to use this technique for anything but the most trivial versioning requirements have inevitably failed.

33.3.3 Explicit Versioning

Yet another twist on the versioning theme is to explicitly version everything, for example:

```
module Filesystem {
    // ...

    interface FileV1 extends NodeV1 {
        idempotent LinesV1 read();
        idempotent void write(LinesV1 text) throws GenericErrorV1;
    };

    class DateTimeV2 extends TimeOfDayV2 {
        // ...
    };

    struct TimesV2 {
        DateTimeV2 createdDate;
        DateTimeV2 accessedDate;
        DateTimeV2 modifiedDate;
    };

    interface FileV2 extends NodeV2 {
        idempotent LinesV2 read();
        idempotent void write(LinesV2 text) throws GenericErrorV2;
        idempotent TimesV2 getTimes();
    };
};
```

In essence, this approach creates as many separate definitions of each data type, interface, and operation as there are versions. It is easy to see that this approach does not work very well:

- Because, at the time version 1 is produced, it is unknown what might need to change for version 2 and later versions, *everything* has to be tagged with a version number. This very quickly leads to an incomprehensible type system.
- Because every version uses its own set of separate types, there is no type compatibility. For example, a version 2 type cannot be passed where a version 1 type is expected without explicit copying.
- Client code must be written to explicitly deal with each separate version. This pollutes the source code at all points where a remote call is made or a Slice data type is passed; the resulting code quickly becomes incomprehensible.

Other approaches, such as placing the definitions for each version into a separate module (that is, versioning the enclosing module instead of each individual type) do little to mitigate these problems; the type incompatibility issues and the need to explicitly deal with versioning remain.

33.4 Versioning with Facets

A negative aspect of all the approaches in Section 33.3 is that they change the type system in intrusive ways. In turn, this forces unacceptable programming contortions on clients. Facets allow you to solve the versioning problem more elegantly because they do not change an existing type system but extend it instead. We already saw this approach in operation on page 1124, where we added date information about a file to our file system application without disturbing any of the existing definitions.

In the most general sense, facets provide a mechanism for implementing multiple interfaces for a single object. The key point is that, to add a new interface to an object, none of the existing definitions have to be touched, so no compatibility issues can arise. More importantly, the decision as to which facet to use is made at run time instead of at compile time. In effect, facets implement a form of late binding and, therefore, are coupled to the type system more loosely than any of the previous approaches.

Used judiciously, facets can handle versioning requirements more elegantly than other mechanisms. Apart from the straight extension of an interface as shown in Section 33.2.1, facets can also be used for more complex changes. For example, if you need to change the parameters of an operation or modify the fields of a structure, you can create a new facet with operations that operate on the changed data types. Quite often, the implementation of a version 2 facet in the server can even re-use much of the version 1 functionality, by delegating some version 2 operations to a version 1 implementation.

33.5 Facet Selection

Given that we have decided to extend an application with facets, we have to deal with the question of how clients select the correct facet. The answer typically involves an explicit selection of a facet sometime during client start-up. For example, in our file system application, clients always begin their interactions

with the file system by creating a proxy to the root directory. Let us assume that our versioning requirements have led to version 1 and version 2 definitions of directories as follows:

```
module Filesystem { // Original version
    // ...

    interface Directory extends Node {
        idempotent NodeSeq list();
        // ...
    };
};

module FilesystemV2 {
    // ...

    enum NodeType { Directory, File };

    class NodeDetails {
        NodeType type;
        string name;
        DateTime createdTime;
        DateTime accessedTime;
        DateTime modifiedTime;
        // ...
    };

    interface Directory extends Filesystem::Node {
       idempotent NodeDetailsSeq list();
       // ...
    };
};
```

In this case, the semantics of the `list` operation have changed in version 2. A version 1 client uses the following code to obtain a proxy to the root directory:

```
// Create a proxy for the root directory
//
Ice::ObjectPrx base
    = communicator()->stringToProxy("RootDir:default -p 10000");
if (!base)
    throw "Could not create proxy";

// Down-cast the proxy to a Directory proxy
//
```

```
Filesystem::DirectoryPrx rootDir
    = Filesystem::DirectoryPrx::checkedCast(base);
if (!rootDir)
    throw "Invalid proxy";
```

For a version 2 client, the bootstrap code is almost identical—instead of down-casting to `Filesystem::Directory`, the client selects the "V2" facet during the down-cast to the type `FileSystemV2::Directory`:

```
// Create a proxy for the root directory
//
Ice::ObjectPrx base
    = communicator()->stringToProxy("RootDir:default -p 10000");
if (!base)
    throw "Could not create proxy";

// Down-cast the proxy to a V2 Directory proxy
//
FilesystemV2::DirectoryPrx rootDir
    = FilesystemV2::DirectoryPrx::checkedCast(base, "V2");
if (!rootDir)
    throw "Invalid proxy";
```

Of course, we can also create a client that can deal with both version 1 and version 2 directories: if the down-cast to version 2 fails, the client is dealing with a version 1 server and can adjust its behavior accordingly.

33.6 Behavioral Versioning

On occasion, versioning requires changes in behavior that are not manifest in the interface of the system. For example, we may have an operation that performs some work, such as:

```
interface Foo {
    void doSomething();
};
```

The same operation on the same interface exists in both versions, but the *behavior* of `doSomething` in version 2 differs from that in version 1. The question is, how do we best deal with such behavioral changes?

Of course, one option is to simply create a version 2 facet and to carry that facet alongside the original version 1 facet. For example:

```
module V2 {

    interface Foo {    // V2 facet
        void doSomething();
    };
};
```

This works fine, as far as it goes: a version 2 client asks for the "V2" facet and then calls `doSomething` to get the desired effect. Depending on your circumstances, this approach may be entirely reasonable. However, if there are such behavioral changes on several interfaces, the approach leads to a more complex type system because it duplicates each interface with such a change.

A better alternative can be to create two facets of the same type, but have the implementation of those facets differ. With this approach, both facets are of type `::Foo::doSomething`. However, the implementation of `doSomething` checks which facet was used to invoke the request and adjusts its behavior accordingly:

```
void
FooI::doSomething(const Ice::Current& c)
{
    if (c.facet == "V2") {
        // Provide version 2 behavior...
    } else {
        // Provide version 1 behavior...
    }
}
```

This approach avoids creating separate types for the different behaviors, but has the disadvantage that version 1 and version 2 objects are no longer distinguishable to the type system. This can matter if, for example, an operation accepts a `Foo` proxy as a parameter. Let us assume that we also have an interface `FooProcessor` as follows:

```
interface FooProcessor {
    void processFoo(Foo* w);
};
```

If `FooProcessor` also exists as a version 1 and version 2 facet, we must deal with the question of what should happen if a version 1 `Foo` proxy is passed to a version 2 `processFoo` operation because, at the type level, there is nothing to prevent this from happening.

You have two options to deal with this situation:

- Define working semantics for mixed-version invocations. In this case, you must come up with sensible system behavior for all possible combinations of versions.
- If some of the mixed-version combinations are disallowed (such as passing a version 1 `Foo` proxy to a version 2 `processFoo` operation), you can detect the version mismatch in the server by looking at the `Current::facet` member and throwing an exception to indicate a version mismatch. Simultaneously, write your clients to ensure they only pass a permissible version to `processFoo`. Clients can ensure this by checking the facet name of a proxy before passing it to `processFoo` and, if there is a version mismatch, changing either the `Foo` proxy or the `FooProcessor` proxy to a matching facet:

```
FooPrx foo = ...;                // Get a Foo...
FooProcessorPrx fooP = ...;      // Get a FooProcessor...

string fooFacet = foo->ice_getFacet();
string fooPFacet = fooP->ice_getFacet();
if (fooFacet != fooPFacet) {
    if (fooPFacet == "V2") {
        error("Cannot pass a V1 Foo to a V2 FooProcessor");
    } else {
        // Upgrade FooProcessor from V1 to V2
        fooP = FooProcessorPrx::checkedCast(fooP, "V2");
        if (!fooP) {
            error("FooProcessor does not have a V2 facet");
        } else {
            fooP->processFoo(foo);
        }
    }
}
```

33.7 Design Considerations

Facets allow you to add versioning to a system, but they are merely a mechanism, not a solution. You still have to make a decision as to how to version something. For example, at some point, you may want to deprecate a previous version's behavior; at that point, you must make a decision how to handle requests for the deprecated version. For behavioral changes, you have to decide whether to use separate interfaces or use facets with the same interface. And, of course, you must have compatibility rules to determine what should happen if, for example, a

version 1 object is passed to an operation that implements version 2 behavior. In other words, facets cannot do your thinking for you and are no panacea for the versioning problem.

The biggest advantage of facets is also the biggest drawback: facets delay the decision about the types that are used and their behavior until run time. While this provides a lot of flexibility, it is significantly less type safe than having explicit types that can be statically checked at compile time: if you have a problem relating to incorrect facet selection, the problem will be visible only at run time and, moreover, will be visible only if you actually execute the code that contains the problem, and execute it with just the right data.

Another danger of facets is to abuse them. As an extreme example, here is an interface that provides an arbitrary collection of objects of arbitrary type:

```
interface Collection {};
```

Even though this interface is empty, it can provide access to an unlimited number of objects of arbitrary type in the form of facets. While this example is extreme, it illustrates the design tension that is created by facets: you must decide, for a given versioning problem, how and at what point of the type hierarchy to split off a facet that deals with the changed functionality. The temptation may be to "simply add another facet" and be done with it. However, if you do that, your objects are in danger of being nothing more than loose conglomerates of facets without rhyme or reason, and with little visibility of their relationships in the type system.

In object modeling terms, the relationship among facets is weaker than an *is-a* relationship (because facets are often not type compatible among each other). On the other hand, the relationship among facets is stronger than a *has-a* relationship (because all facets of an Ice object share the same object identity).

It is probably best to treat the relationship of a facet to its Ice object with the same respect as an inheritance relationship: if you were omniscient and could have designed your system for all current and future versions simultaneously, many of the operations that end up on separate facets would probably have been in the same interface instead. In other words, adding a facet to an Ice object most often implies that the facet has an *is-partly-a* relationship with its Ice object. In particular, if you think about the life cycle of an Ice object and find that, when an Ice object is deleted, all its facets must be deleted, this is a strong indication of a correct design. On the other hand, if you find that, at various times during an Ice object's life cycle, it has a varying number of facets of varying types, that is a good indication that you are using facets incorrectly.

Ultimately, the decision comes down to deciding whether the trade-off of static type safety versus dynamic type safety is worth the convenience and back-

ward compatibility. The answer depends strongly on the design of your system and individual requirements, so we can only give broad advice here. Finally, there will be a point where no amount of facet trickery will get past the point when "yet one more version will be the straw that breaks the camel's back." At that point, it is time to stop supporting older versions and to redesign the system.

33.8 Summary

Facets provide a way to extend a type system by loosely coupling new type instances to existing ones. This shifts the type selection process from compile to run time and implements a form of late binding. Due to their loose coupling among each other, facets are better suited to solve the versioning problem than other approaches. However, facets are not a panacea that would solve the versioning problem for free, and careful design is still necessary to come up with versioned systems that remain understandable and maintain consistent semantics.

Chapter 34
Object Life Cycle

34.1 Chapter Overview

This chapter discusses object life cycle, in particular with respect to concurrency. The majority of examples in this chapter use C++. However, the issues discussed here apply equally to all programming languages. Moreover, object life cycle is a surprisingly complex topic, so we suggest that you read this chapter in detail, regardless of your choice of programming language.

Sections 34.2 through 34.4 discuss the fundamentals of object life cycle, in particular, what it means for an object to exist and not exist and how life cycle relates to proxies, servants, and Ice objects. Section 34.5 covers object creation and Section 34.6 discusses object destruction, with particular emphasis on concurrency issues. Section 34.7 discusses alternative approaches to object destruction and Section 34.8 discusses a number of architectural issues with respect to object identity. Section 34.9 presents an implementation of the file system application with life cycle support in C++ and Java and, finally, Section 34.10 shows how to deal with objects that are abandoned by clients.

34.2 Introduction

Object life cycle generally refers to how an object-oriented application (whether distributed or not) creates and destroys objects. For distributed applications, life cycle management presents particular challenges. For example, destruction of objects often can be surprisingly complex, especially in threaded applications. Before we go into the details of object creation and destruction, we need to have a closer look what we mean by the terms "life cycle" and "object" in this context.

Object life cycle refers to the act of creation and destruction of objects. For example, with our file system application, we may start out with an empty file system that only contains a root directory. Over time, clients (by as yet unspecified means) add new directories and files to the file system. For example, a client might create a new directory called `MyPoems` underneath the root directory. Some time later, the same or a different client might decide to remove this directory again, returning the file system to its previous empty state. This pattern of creation and destruction is known as object life cycle.

The life cycle of distributed objects raises a number of interesting and challenging questions. For example, what should happen if a client destroys a file while another client is reading or writing that file? And how do we prevent two files with the same name from existing in the same directory? Another interesting scenario is illustrated by the following sequence of events:

1. Client A creates a file called `DraftPoem` in the root directory and uses it for a while.
2. Some time later, client B destroys the `DraftPoem` file so it no longer exists.
3. Some time later still, client C creates a new `DraftPoem` file in the root directory, with different contents.
4. Finally, client A attempts to access the `DraftPoem` file it created earlier.

What should happen when, in the final step, client A tries to use the `DraftPoem` file? Should the client's attempt succeed and simply operate on the new contents of the file that were placed there by client C? Or should client A's attempt fail because, after all, the new `DraftPoem` file is, in a sense, a completely different file from the original one, even though it has the same name?

The answers to such questions cannot be made in general. Instead, meaningful answers depend on the semantics that each individual application attaches to object life cycle. In this chapter, we will explore the various possible interpretations and how to implement them correctly, particularly for threaded applications.

34.3 Object Existence and Non-Existence

Before we talk about how to create and destroy objects, we need to look at a more basic concept, namely that of object existence. What does it mean for an object to "exist" and, more fundamentally, what do we mean by the term "object"?

As mentioned in Section 2.2.2, an *Ice object* is a conceptual entity, or abstraction that does not really exist. On the client side, the concrete representation of an Ice object is a proxy and, on the server side, the concrete representation of an Ice object is a servant. Proxies and servants are the concrete programming-language artifacts that represent Ice objects.

Because Ice objects are abstract, conceptual entities, they are invisible to the Ice run time and to the application code—only proxies and servants are real and visible. It follows that, to determine whether an *Ice* object exists, any determination must rely on proxies and servants, because they are the only tangible entities in the system.

34.3.1 Object Non-Existence

Here is the definitive statement of what it means for an Ice object to *not* exist:

> *An Ice object does not exist if an invocation on the object raises an `ObjectNotExistException`.*

This may seem self-evident but, on closer examination, is a little more subtle than you might expect. In particular, Ice object existence has meaning only *within the context of a particular invocation*. If that invocation raises `ObjectNotExistException`, the object is known to not exist. Note that this says nothing about whether concurrent or future requests to that object will also raise `ObjectNotExistException`—they may or may not, depending on the semantics that are implemented by the application.

Also note that, because all the Ice run time knows about are servants, an `ObjectNotExistException` really indicates that a *servant* for the request could not be found at the time the request was made. This means that, ultimately, it is the application that attaches the meaning "the Ice object does not exist" to this exception.

In theory, the application can attach any meaning it likes to `ObjectNotExistException` and a server can throw this exception for whatever reason it sees fit; in practice, however, we recommend that you do not do this because it breaks with existing convention and is potentially confusing. You should reserve this exception for its intended meaning and not abuse it for other purposes.

34.3.2 Object Existence

The preceding definition does not say anything about object existence if something other than `ObjectNotExistException` is returned in response to a particular request. So, here is the definitive statement of what it means for an Ice object to exist:

> *An Ice object exists if a twoway invocation on the object either succeeds, raises a user exception, or raises `FacetNotExistException` or `OperationNotExistException`.*

It is self-evident that an Ice object exists if a twoway invocation on it succeeds: obviously, the object received the invocation, processed it, and returned a result. However, note the qualification: this is true only for *twoway* invocations; for oneway and datagram invocations (see Sections 32.14 and 32.15), nothing can be inferred about the existence of the corresponding Ice object by invoking an operation on it: because there is no reply from the server, the client-side Ice run time has no idea whether the request was dispatched successfully in the server or not. This includes user exceptions, `ObjectNotExistException`, `FacetNotExistException`, and `OperationNotExistException`—these exceptions are never raised by oneway and datagram invocations, regardless of the actual state of the target object.

If a twoway invocation raises a user exception, the Ice object obviously exists: the Ice run time never raises user exceptions so, for an invocation to raise a user exception, the invocation was dispatched successfully in the server, and the operation implementation in the servant raised the exception.

If a twoway invocation raises `FacetNotExistException`, we do know that the corresponding Ice object indeed exists: the Ice run time raises `FacetNotExistException` only if it can find the identity of the target object in the Active Servant Map (ASM), but cannot find the facet (see Chapter 33) that was specified by the client.[1]

1. Note that, if you use servant locators (see Section 32.7), for these semantics to hold, your servant locator must correctly raise `FacetNotExistException` (instead of returning null or raising `ObjectNotExistException`) if an Ice object exists, but the particular facet does not exist.

As a corollary to the preceding two definitions, we can state:

A facet does not exist if a twoway invocation on the object raises `ObjectNotExistException` or `FacetNotExistException`.

A facet exists if a twoway invocation on the object either succeeds, or raises `OperationNotExistException`.

These definitions simply capture the fact that a facet is a "sub-object" of an Ice object: if an invocation raises `ObjectNotExistException`, we know that the facet does not exist either because, for a facet to exist, its Ice object must exist.

If an operation raises `OperationNotExistException`, we know that both the target Ice object and the target facet exist. However, the operation that the client attempted to invoke does not. (This is possible only if you use dynamic invocation (see Chapter 35) or if you have mis-matched Slice definitions for client and server.)

34.3.3 Indeterminate Object State

The preceding definitions clearly state under what circumstances we can conclude that an Ice object (or its facet) does or does not exist. However, the preceding definitions are incomplete because operation invocations can have outcomes other than success or failure with `ObjectNotExistException`, `FacetNotExistException`, or `OperationNotExistException`. For example, a client might receive a `MarshalException`, `UnknownLocalException`, `UnknownException`, or `TimeoutException`. In that case, the client cannot draw any conclusions about whether the Ice object on which it invoked a twoway operation exists or not—the exceptions simply indicate that something went wrong while the invocation was processed. So, to complete our definitions, we can state:

If a twoway invocation raises a run-time exception other than `ObjectNotExistException`, `FacetNotExistException`, or `OperationNotExistException`, nothing is known about the existence or non-existence of the Ice object that was the target of the invocation. Furthermore, it is impossible to determine the state of existence of an Ice object with a oneway or datagram invocation.

34.3.4 Authoritative Semantics

The preceding definitions capture the fact that, to make a determination of object existence or non-existence, the client-side Ice run time must be able to contact the server and, moreover, receive a reply from the server:

- If the server can be contacted and returns a successful reply for an invocation, the Ice object exists.
- If the server can be contacted and returns an `ObjectNotExistException` (or `FacetNotExistException`), the Ice object (or facet) does not exist. If the server returns an `OperationNotExistException`, the Ice object (and its facet) exists, but does not provide the requested operation, which indicates a type mismatch due to client and server using out-of-sync Slice definitions or due to incorrect use of dynamic invocation.
- If the server cannot be contacted, does not return a reply (as for oneway and datagram invocations), or if anything at all goes wrong with the process of sending an invocation, processing it in the server, and returning the reply, nothing is known about the state of the Ice object, including its existence or non-existence.

Another way of looking at this is that a decision as to whether an object exists or not is *never* made by the Ice run time and, instead, is *always* made by the server-side *application* code:

- If an invocation completes successfully, the server-side application code was involved because it processed the invocation.
- If an invocation returns `ObjectNotExistException` or `FacetNotExistException`, the server-side application code was also involved:
  - either the Ice run time could not find a servant for the invocation in the ASM, in which case the application code was involved by virtue of not having added a servant to the ASM in the first place, or
  - the Ice run time consulted a servant locator that explicitly returned null or raised `ObjectNotExistException` or `FacetNotExistException`.

This means that `ObjectNotExistException` and `FacetNotExistException` are *authoritative*: when you receive these exceptions, you can always believe what they tell you—the Ice run time never raises these exceptions without consulting your code, either implicitly (via an ASM lookup) or explicitly (by calling a servant locator's `locate` operation).

These semantics are motivated by the need to keep the Ice run time stateless with respect to object existence. For example, it would be nice to have stronger semantics, such as a promise that "once an Ice object has existed and been destroyed, all future requests to that Ice object also raise `ObjectNotExistException`". However, to implement these semantics, the Ice run time would have to remember all object identities that were used in the past, and prevent their reuse for new Ice objects. Of course, this would be inherently non-scalable. In addition,

it would prevent applications from controlling object identity; allowing such control for applications is important however, for example, to link the identity of an Ice object to its persistent state in a database (see Chapter 39).

Note that, if the implementation of an operation calls another operation, dealing with `ObjectNotExistException` may require some care. For example, suppose that the client holds a proxy to an object of type `Service` and invokes an operation `provideService` on it:

```
ServicePrx service = ...;

try {
    service->provideService();
} catch (const ObjectNotExistException&) {
    // Service does not exist.
}
```

Here is the implementation of `provideService` in the server, which makes a call on a helper object to implement the operation:

```
void
ServiceI::provideService(const Ice::Current&)
{
    // ...
    proxyToHelper->someOp();
    // ...
}
```

If `proxyToHelper` happens to point at an object that was destroyed previously, the call to `someOp` will throw `ObjectNotExistException`. If the implementation of `provideService` does not intercept this exception, the exception will propagate all the way back to the client, who will conclude that the service has been destroyed when, in fact, the service still exists but the helper object used to implement `provideService` no longer exists.

Usually, this scenario is not a serious problem. Most often, the helper object cannot be destroyed while it is needed by `provideService` due to the way the application is structured. In that case, no special action is necessary because `someOp` will never throw `ObjectNotExistException`. On the other hand, if it is possible for the helper object to be destroyed, `provideService` can wrap a try-catch block for `ObjectNotExistException` around the call to `someOp` and throw an appropriate user exception from the exception handler (such as `ResourceUnavailable` or similar).

34.4 Life Cycle of Proxies, Servants, and Ice Objects

It is important to be aware of the different roles of proxies, servants, and Ice objects in a system. Proxies are the client-side representation of Ice objects and servants are the server-side representation of Ice objects. Proxies, servants, and Ice objects have completely independent life cycles. Clients can create and destroy proxies with or without a corresponding servant or Ice object in existence, servers can create and destroy servants with or without a corresponding proxy or Ice object in existence and, most importantly, Ice objects can exist or not exist regardless of whether corresponding proxies or servants exist. Here are a few examples to illustrate this:

```
{
    Ice::ObjectPtr obj
        = communicator->stringToProxy("hello:tcp -p 10000");
    // Proxy exists now.

} // Proxy ceases to exist.
```

This code creates a proxy to an Ice object with the identity `Hello`. The server for this Ice object is expected to listen for invocations on the same host as the client, on port 10000, using the TCP/IP protocol. The proxy exists as soon as the call to `stringToProxy` completes and, thereafter, can by used by the client to make invocations on the corresponding Ice object.

However, note that this code says nothing at all about whether or not the corresponding Ice object exists. In particular, there might not be any Ice object with the identity `Hello`. Or there might be such an object, but the server for it may be down or unreachable. It is only when the client makes an invocation on the proxy that we get to find out whether the object exists, does not exist, or cannot be reached.

Similarly, at the end of the scope enclosing the `obj` variable in the preceding code, the proxy goes out of scope and is destroyed. Again, this says nothing about the state of the corresponding Ice object or its servant. This shows that the life cycle of a proxy is completely independent of the life cycle of its Ice object and the servant for that Ice object: clients can create and destroy proxies whenever they feel like it, and doing so has no implications for Ice objects or servant creation or destruction.

Here is another code example, this time for the server side:

```
{
    FileIPtr file = new FileI("DraftPoem", root);
    // Servant exists now.

} // Servant ceases to exist.
```

Here, the server instantiates a servant for a `File` object by creating a `FileI` instance. The servant comes into being as soon as the call to `new` completes and ceases to exist as soon as the scope enclosing the `file` variable closes. Note that, as for proxies, the life cycle of the servant is completely independent of the life cycle of proxies and Ice objects. Clearly, the server can create and destroy a servant regardless of whether there are any proxies in existence for the corresponding Ice object.

Similarly, an Ice object can exist even if no servants exist for it. For example, our Ice objects might be persistent and stored in a database; in that case, if we switch off the server for our Ice objects, no servants exist for these Ice objects, even though the Ice objects continue to exist—the Ice objects are temporarily inaccessible, but exist regardless and, once their server is restarted, will become accessible again.

Conversely, a servant can exist without its corresponding Ice object. The mere creation of a servant does nothing, as far as the Ice run time is concerned. It is only once a servant is added to the ASM (or a servant locator returns the server from `locate`) that the servant incarnates its Ice object.

Finally, an Ice object can exist independently of proxies and servants. For example, returning to the database example, we might have an Ice server that acts as a front end to an online telephone book: each entry in the phone book corresponds to a separate Ice object. When a client invokes an operation, the server uses the identity of the incoming request to determine which Ice object is the target of the request and then contacts the back-end database to, for example, return the street address of the entry. With such a design, entries can be added to and removed from the back-end database quite independently of what happens to proxies and servants—the server finds out whether an Ice object exists only when it accesses the back-end database.

The only time that the life cycle of an Ice object and a servant are linked is during an invocation on that Ice object: for an invocation to complete successfully, a servant must exist *for the duration of the invocation*. What happens to the servant thereafter is irrelevant to clients and, in general, is irrelevant to the corresponding Ice object.

It is important to be clear about the independence of the life cycles of proxies, servants, and Ice objects because this independence has profound implications for

how you need to implement object life cycle. In particular, to destroy an Ice object, a client cannot simply destroy its proxy for an object because the server is completely unaware when a client does this.[2]

34.5 Object Creation

Now that we understand what it means for an Ice object to exist, we can look at what is involved in creating an Ice object. Fundamentally, there is only one way for an Ice object to come into being: the server must instantiate a servant for the object and add an entry for that servant to the ASM (or, alternatively, arrange for a servant locator to return a servant from its `locate` operation—see Section 32.7).[3]

One obvious way for a server to create a servant is to, well, simply instantiate it and add it to the ASM of its own accord. For example:

```
DirectoryIPtr root = new DirectoryI("/", 0);
adapter->addWithUUID(root); // Ice object exists now
```

The servant exists as soon as the call to `new` completes, and the Ice object exists as soon as the code adds the servant to the ASM: at that point, the Ice object becomes reachable to clients who hold a proxy to it.

This is the way we created Ice objects for our file system application in earlier chapters. However, doing so is not all that interesting because the only files and directories that exist are those that the server decides to create when it starts up. What we really want is a way for *clients* to create and destroy directories and files.

34.5.1 Object Factories

The canonical way to create an object is to use the factory pattern [2]. The factory pattern, in a nutshell, says that objects are created by invoking an operation (usually called `create`) on an object factory:[4]

2. Distributed object systems such as DCOM implement these semantics. However, this design is inherently non-scalable because of the cost of globally tracking proxy creation and destruction.

3. For the remainder of this chapter, we will ignore the distinction between using the ASM and a servant locator and simply assume that the code uses the ASM. This is because servant locators do not alter the discussion: if `locate` returns a servant, that is the same as a successful lookup in the ASM; if `locate` returns null or throws `ObjectNotExistException`, that is the same as an unsuccessful lookup in the ASM.

```
interface PhoneEntry {
    idempotent string name();
    idempotent string getNumber();
    idempotent void setNumber(string phNum);
};

exception PhoneEntryExists {
    string name;
    string phNum;
};

interface PhoneEntryFactory {
    PhoneEntry* create(string name, string phNum)
        throws PhoneEntryExists;
};
```

The entries in the phone book consist of simple name–number pairs. The interface to each entry is called `PhoneEntry` and provides operations to read the name and to read and write the phone number. (For a real application, the objects would likely be more complex and encapsulate more state. However, these simple objects will do for the purposes of this discussion.)

To create a new entry, a client calls the `create` operation on a `PhoneEntryFactory` object. (The factory is a singleton object [2], that is, only one instance of that interface exists in the server.) It is the job of `create` to create a new `PhoneEntry` object, using the supplied name as the object identity.

An immediate consequence of using the name as the object identity is that `create` can raise a `PhoneEntryExists` exception: presumably, if a client attempts to create an entry with the same name as an already-existing entry, we need to let the client know about this. (Whether this is an appropriate design is something we examine more closely in Section 34.8.)

`create` returns a proxy to the newly-created object, so the client can use that proxy to invoke operations. However, this is by convention only. For example, `create` could be a `void` operation if the client has some other way to eventually get a proxy to the new object (such as creating the proxy from a string, or locating the proxy via a search operation). Alternatively, you could define "bulk" creation operations that allow clients to create several new objects with a single RPC. As

4. Rather than continue with the file system example, we will simplify the discussion for the time being by using the phone book example mentioned earlier; we will return to the file system application to explore more complex issues in Section 34.9.

far as the Ice run time is concerned, there is nothing special about a factory operation: a factory operation is just like any other operation; it just so happens that a factory operation creates a new Ice object as a side effect of being called, that is, the *implementation* of the operation is what creates the object, not the Ice run time.

Also note that `create` accepts a `name` and a `phNum` parameter, so it can initialize the new object. This is not compulsory, but generally a good idea. An alternate factory operation could be:

```
interface PhoneEntryFactory {
    PhoneEntry* create(string name)
        throws PhoneEntryExists;
};
```

With this design, the assumption is that the client will call `setNumber` after it has created the object. However, in general, allowing objects that are not fully initialized is a bad idea: it all too easily happens that a client either forgets to complete the initialization, or happens to crash or get disconnected before it can complete the initialization. Either way, we end up with a partially-initialized object in the system that can cause surprises later.[5]

Similarly, so-called *generic* factories are also something to be avoided:

```
dictionary<string, string> Params;

exception CannotCreateException {
    string reason;
};

interface GenericFactory {
    Object* create(Params p)
        throws CannotCreateException;
};
```

The intent here is that a `GenericFactory` can be used to create any kind of object; the `Params` dictionary allows an arbitrary number of parameters to be passed to the `create` operation in the form of name–value pairs, for example:

5. This is the approach taken by COM's `CoCreateObject`, which suffers from just that problem.

```
GenericFactoryPrx factory = ...;

Ice::ObjectPrx obj;
Params p;

// Make a car.
//
p["Make"] = "Ford";
p["Model"] = "Falcon";
obj = factory->create(p);
CarPrx car = CarPrx::checkedCast(obj);

// Make a horse.
//
p.clear();
p["Breed"] = "Clydesdale";
p["Sex"] = "Male";
obj = factory->create(p);
HorsePrx horse = HorsePrx::checkedCast(obj);
```

We strongly discourage you from creating factory interfaces such as this, unless you have a good overriding reason: generic factories undermine type safety and are much more error-prone than strongly-typed factories.

34.5.2 Implementing a Factory Operation

The implementation of an object factory is simplicity itself. Here is how we could implement the `create` operation for our `PhoneEntryFactory`:

```
PhoneEntryPrx
PhoneEntryFactory::create(const string& name,
                          const string& phNum,
                          const Current& c)
{
    try {
        CommunicatorPtr comm = c.adapter.getCommunicator();
        PhoneEntryPtr servant = new PhoneEntryI(name, phNum);
        return PhoneEntryPrx::uncheckedCast(
            c.adapter->add(servant,
                           comm->stringToIdentity(name)));
    } catch (const Ice::AlreadyRegisteredException&) {
        throw PhoneEntryExists(name, phNum);
    }
}
```

The `create` function instantiates a new `PhoneEntryI` object (which is the servant for the new `PhoneEntry` object), adds the servant to the ASM, and returns the proxy for the new object. Adding the servant to the ASM is what creates the new Ice object, and client requests are dispatched to the new object as soon as that entry appears in the ASM (assuming the object adapter is active).

Note that, even though this code contains no explicit lock, it is thread-safe. The `add` operation on the object adapter is atomic: if two clients concurrently add a servant with the same identity, exactly one thread succeeds in adding the entry to the ASM; the other thread receives an `AlreadyRegisteredException`. Similarly, if two clients concurrently call `create` for different entries, the two calls execute concurrently in the server (if the server is multi-threaded); the implementation of `add` in the Ice run time uses appropriate locks to ensure that concurrent updates to the ASM cannot corrupt anything.

34.6 Object Destruction

Now that clients can create `PhoneEntry` objects, let us consider how to allow clients to destroy them again. One obvious design is to add a `destroy` operation to the factory—after all, seeing that a factory knows how to create objects, it stands to reason that it also knows how to destroy them again:

```
exception PhoneEntryNotExists {
    string name;
    string phNum;
};

interface PhoneEntryFactory {
    PhoneEntry* create(string name, string phNum)
        throws PhoneEntryExists;
    void destroy(PhoneEntry* pe)      // Bad idea!
        throws PhoneEntryNotExists;
};
```

While this works (and certainly can be implemented without problems), it is generally a bad idea. For one, an immediate problem we need to deal with is what should happen if a client passes a proxy to an already-destroyed object to `destroy`. We could raise an `ObjectNotExistException` to indicate this, but that is not a good idea because it makes it ambiguous as to which object does not exist: the factory, or the entry. (By convention, if a client receives an `ObjectNotExist-Exception` for an invocation, what does not exist is the object the operation was

targeted at, not some other object that in turn might be contacted by the operation.) This forces us to add a separate `PhoneEntryNotExists` exception to deal with the error condition, which makes the interface a little more complex.

A second and more serious problem with this design is that, in order to destroy an entry, the client must not only know which entry to destroy, but must also know *which factory created the entry*. For our example, with only a single factory, this is not a serious concern. However, for more complex systems with dozens of factories (possibly in multiple server processes), it rapidly becomes a problem: for each object, the application code somehow has to keep track of which factory created what object; if any part of the code ever loses track of where an object originally came from, it can no longer destroy that object.

Of course, we could mitigate the problem by adding an operation to the `PhoneEntry` interface that returns a proxy to its factory. That way, clients could ask each object to provide the factory that created the object. However, that needlessly complicates the Slice definitions and really is just a band-aid on a fundamentally flawed design. A much better choice is to add the `destroy` operation to the `PhoneEntry` interface instead:

```
interface PhoneEntry {
    idempotent string name();
    idempotent string getNumber();
    idempotent void setNumber(string phNum);
    void destroy();
};
```

With this approach, there is no need for clients to somehow keep track of which factory created what object. Instead, given a proxy to a `PhoneEntry` object, a client simply invokes the `destroy` operation on the object and the `PhoneEntry` obligingly commits suicide. Note that we also no longer need a separate exception to indicate the "object does not exist" condition because we can raise `ObjectNotExistException` instead—the exception exists precisely to indicate this condition and, because `destroy` is now an operation on the phone entry itself, there is no ambiguity about which object it is that does not exist.

34.6.1 Idempotency and Life Cycle Operations

You may be tempted to write the life cycle operations as follows:

```
interface PhoneEntry {
    // ...
    idempotent void destroy(); // Wrong!

};

interface PhoneEntryFactory {
    idempotent PhoneEntry* create(string name, string phNum)
                                throws PhoneEntryExists;
};
```

The idea is that `create` and `destroy` can be `idempotent` operations because it is safe to let the Ice run time retry the operation in the event of a temporary network failure. However, this assumption is not true. To see why, consider the following scenario:

1. A client invokes `destroy` on a phone entry.
2. The Ice run time sends the request to the server on the wire.
3. The connection goes down just after the request was sent, but before the reply for the request arrives in the client. It so happens that the request was received by the server and acted upon, and the reply from the server back to the client is lost because the connection is still down.
4. The Ice run time tries to read the reply for the request and realizes that the connection has gone down. Because the operation is marked idempotent, the run time attempts to re-establish the connection and send the request a second time, which happens to work.
5. The server receives the request to destroy the entry but, because the entry is destroyed already, the server returns an `ObjectNotExistException` to the client, which the Ice run time passes to the application code.
6. The application receives an `ObjectNotExistException` and falsely concludes that it tried to destroy a non-existent object when, in fact, the object did exist and was destroyed as intended.

A similar scenario can be constructed for `create`: in that case, the application will receive a `PhoneEntryExists` exception when, in fact, the entry did not exist and was created successfully.

These scenarios illustrate that create and destroy are *never* idempotent: sending one `create` or `destroy` invocation for a particular object is not the same as sending two invocations: the outcome depends on whether the first invocation succeeded or not, so `create` and `destroy` are not idempotent.

34.6.2 Implementing a `destroy` Operation

As far as the Ice run time is concerned, the act of destroying an Ice object is to remove the mapping between its proxy and its servant. In other words, an Ice object is destroyed when we remove its ASM entry. Once the ASM entry is gone, incoming operations for the object raise `ObjectNotExistException`, as they should.

So, here is the most simple version of `destroy`:

```
void
PhoneEntryI::destroy(const Current& c)
{
    try {
        c.adapter->remove(c.id);
    } catch (const Ice::NotRegisteredException&)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
}
```

The implementation removes the ASM entry for the servant, thereby destroying the Ice object. If the entry does not exist (presumably, because the object was destroyed previously), `destroy` throws an `ObjectNotExistException`, as you would expect.

Object Destruction and Concurrency

The ASM entry is removed as soon as `destroy` calls `remove` on the object adapter. Assuming that we implement `create` as we saw earlier, so no other part of the code retains a smart pointer to the servant[6], this means that the ASM holds the only smart pointer to the servant, so the servant's reference count is 1. Once the ASM entry is removed (and its smart pointer destroyed), the reference count of the servant drops to zero. In C++, this triggers a call to the destructor of the servant, and the heap-allocated servant is deleted just as it should be; in languages such as Java and C#, this makes the servant eligible for garbage collection, so it will be deleted eventually as well.

Things get more interesting if we consider concurrent scenarios. One such scenario involves concurrent calls to `create` and `destroy`. Suppose we have the following sequence of events:

1. Client A creates a phone entry.

6. Or reference to the servant, in languages such as Java and C#.

2. Client A passes the proxy for the entry to client B.
3. Client A destroys the entry again.
4. Client A calls `create` for the same entry (passing the same name, which serves as the object identity) and, concurrently, client B calls `destroy` on the entry.

Clearly, something is strange about this scenario, because it involves two clients asking for conflicting things, with one client trying to create an object that existed previously, while another client tries to destroy the object that—unbeknownst to that client—was destroyed earlier.

Exactly what is seen by client A and client B depends on how the operations are dispatched in the server. In particular, the outcome depends on the order in which the calls on the object adapter to `add` (in `create`) and `remove` (in `destroy`) on the servant are executed:

- If the thread processing client A's invocation executes `add` before the thread processing client B's invocation, client A's call to `add` succeeds. Internally, the calls to `add` and `remove` are serialized, and client B's call to `remove` blocks until client A's call to `add` has completed. The net effect is that both clients see their respective invocations complete successfully.
- If the thread processing client Bs invocation executes `remove` before the thread processing client A's invocation executes `add`, client B's thread receives a `NotRegisteredException`, which results in an `ObjectNotExist-Exception` in client B. Client A's thread then successfully calls `add`, creating the object and returning its proxy.

This example illustrates that, if life cycle operations interleave in this way, the outcome depends on thread scheduling. However, as far as the Ice run time is concerned, doing this is perfectly safe: concurrent access does not cause problems for memory management or the integrity of data structures.

The preceding scenario allows two clients to attempt to perform conflicting operations. This is possible because clients can control the object identity of each phone entry: if the object identity were hidden from clients and assigned by the server (the server could assign a UUID to each entry, for example), the above scenario would not be possible. We will return to a more detailed discussion of such object identity issues in Section 34.8.

Concurrent Execution of Life Cycle and Non-Life Cycle Operations[7]

Another scenario relates to concurrent execution of ordinary (non-life cycle) operations and `destroy`:

- Client A holds a proxy to an existing object and passes that proxy to client B.
- Client B calls the `setNumber` operation on the object.
- Client A calls `destroy` on the object *while Client B's call to `setNumber` is still executing*.

The immediate question is what this means with respect to memory management. In particular, client A's thread calls `remove` on the object adapter while client B's thread is still executing inside the object. If this call to `remove` were to delete the servant immediately, it would delete the servant while client B's thread is still executing inside the servant, with potentially disastrous results.

The answer is that this cannot happen. Whenever the Ice run time dispatches an incoming invocation to a servant, it increments the servant's reference count for the duration of the call, and decrements the reference count again once the call completes. Here is what happens to the servant's reference count for the preceding scenario:

1. Initially, the servant is idle, so its reference count is at least 1 because the ASM entry stores a smart pointer to the servant. (The remainder of these steps assumes that the ASM stores the *only* smart pointer to the servant, so the reference count is exactly 1.)
2. Client B's invocation of `setNumber` arrives and the Ice run time increments the reference count to 2 before dispatching the call.
3. While `setNumber` is still executing, client A's invocation of `destroy` arrives and the Ice run time increments the reference count to 3 before dispatching the call.
4. Client A's thread calls `remove` on the object adapter, which destroys the smart pointer in the ASM and so decrements the reference to 2.
5. Either `setNumber` or `destroy` may complete first. It does not matter which call completes—either way, the Ice run time decrements the reference count as the call completes, so after one of these calls completes, the reference count drops to 1.
6. Eventually, when the final call (`setNumber` or `destroy`) completes, the Ice run time decrements the reference count once again, which causes the count to drop to zero. In turn, this triggers the call to `delete` (which calls the servant's destructor).

7. This section applies to C++ only.

The net effect is that, while operations are executing inside a servant, the servant's reference count is always greater than zero. As the invocations complete, the reference count drops until, eventually, it reaches zero. However, that can only happen once no operations are executing, that is, once the servant is idle. This means that the Ice run time guarantees that a servant's destructor runs only once the final operation invocation has drained out of the servant, so it is impossible to "pull memory out from underneath an executing invocation".[8]

34.6.3 Cleaning Up Servant State

Here is a very simple implementation of our `PhoneEntryI` servant. (Methods are inlined for convenience only. Also note that, for the time being, this code ignores concurrency issues, which we return to in Section 34.6.5.)

```
class PhoneEntryI : public PhoneEntry {
public:
    PhoneEntryI(const string& name, const string& phNum)
        : _name(name), _phNum(phNum)
    {
    }

    virtual string
    name(const Current&) {
        return _name;
    }

    virtual string
    getNumber(const Current&) {
        return _phNum;
    }

    virtual void
    setNumber(const string& phNum, const Current&) {
        _phNum = phNum;
    }

    virtual void
    destroy(const Current& c) {
```

8. For garbage-collected languages, such as C# and Java, the language run time provides the same semantics: while the servant can be reached via any reference in the application or the Ice run time, the servant will not be reclaimed by the garbage collector.

```
        try {
           c.adapter->remove(c.id);
        } catch (const Ice::NotRegisteredException&)
           throw Ice::ObjectNotExistException(__FILE__, __LINE__);
        }
    }

private:
    const string _name;
    string _phNum;
};
```

With this servant, `destroy` does just the right thing: it calls `delete` on the servant once the servant is idle, which in turn calls the destructor, so the memory used by the `_name` and `_phNum` data members is reclaimed.

However, real servants are rarely this simple. In particular, destruction of an Ice object may involve non-trivial actions, such as flushing a file, committing a transaction, making a remote call on another object, or updating a hardware device. For example, instead of storing the details of a phone entry in member variables, the servant could be implemented to store the details in a file; in that case, destroying the Ice object would require closing the file. Seeing that the Ice run time calls the destructor of a servant only once the servant becomes idle, the destructor would appear to be an ideal place to perform such actions, for example:

```
class PhoneEntryI : public PhoneEntry {
public:
    // ...

    ~PhoneEntryI()
    {
        _myStream.close(); // Bad idea
    }

private:
    fstream _myStream;
};
```

The problem with this code is that it can fail, for example, if the file system is full and buffered data cannot be written to the file. Such clean-up failure is a general issue for non-trivial servants: for example, a transaction can fail to commit, a remote call can fail if the network goes down, or a hardware device can be temporarily unresponsive.

If we encounter such a failure, we have a serious problem: we cannot inform the client of the error because, as far as the client is concerned, the `destroy` call

completed just fine. The client will therefore assume that the Ice object was correctly destroyed. However, the system is now in an inconsistent state: the Ice object was destroyed (because its ASM entry was removed), but the object's state still exists (possibly with incorrect values), which can cause errors later.

Another reason for avoiding such state clean-up in C++ destructors is that destructors cannot throw exceptions: if they do, and do so in the process of being called during unwinding of the stack due to some other exception, the program goes directly to `terminate` and does not pass "Go". (There are a few exotic cases in which it is possible to throw from a destructor and get away with it but, in general, is an excellent idea to maintain the no-throw guarantee for destructors.) So, if anything goes wrong during destruction, we are in a tight spot: we are forced to swallow any exception that might be encountered by the destructor, and the best we can do is log the error, but not report it to the client.

Finally, using destructors to clean up servant state does not port well to languages such as Java and C#. For these languages, similar considerations apply to error reporting from a finalizer and, with Java, finalizers may not run at all. Therefore, we recommend that you perform any clean-up actions in the body of `destroy` instead of delaying clean-up until the servant's destructor runs.

Note that the foregoing does *not* mean that you cannot reclaim servant resources in destructors; after all, that is what destructors are for. But it *does* mean that you should not try to reclaim resources from a destructor if the attempt can fail (such as deleting records in an external system as opposed to, for example, deallocating memory or adjusting the value of variables in your program).

34.6.4 Life Cycle and Collection Operations

The factory we defined in Section 34.5.1 is what is known as a *pure* object factory because `create` is the only operation it provides. However, it is common for factories to do double duty and also act as collection managers that provide additional operations, such as `list` and `find`:

```
// ...

sequence<PhoneEntry*> PhoneEntries;

interface PhoneEntryFactory {
    PhoneEntry* create(string name, string phNum)
        throws PhoneEntryExists;
```

```
    idempotent PhoneEntry find(string name);
    idempotent PhoneEntries list();
};
```

`find` returns a proxy for the phone entry with the given name, and a null proxy if no such entry exists. `list` returns a sequence that contains the proxies of all existing entries.

Here is a simple implementation of `find`:

```
PhoneEntryPrx
PhoneEntryFactory::find(const string& name, const Current& c)
{
    CommunicatorPtr comm = c.adapter->getCommunicator();

    PhoneEntryPrx pe;
    Identity id = comm->stringToIdentity(name);
    if (c.adapter->find(id)) {
        pe = PhoneEntryPrx::uncheckedCast(
                                c.adapter->createProxy(id));
    }
    return pe;
}
```

If an entry exists in the ASM for the given name, the code creates a proxy for the corresponding Ice object and returns it. This code works correctly even for threaded servers: because the look-up of the identity in the ASM is atomic, there is no problem with other threads concurrently modifying the ASM (for example, while servicing calls from other clients to `create` or `destroy`).

Cyclic Dependencies

Unfortunately, implementing `list` is not as simple because it needs to iterate over the collection of entries, but the object adapter does not provide any iterator for ASM entries.[9] Therefore, we must maintain our own list of entries inside the factory:

9. The reason for this is that, during iteration, the ASM would have to be locked to protect it against concurrent access, but locking the ASM would prevent call dispatch during iteration and easily cause deadlocks.

```
class PhoneEntryFactoryI : public PhoneEntryFactory
{
public:
    // ...

    void remove(const string&, const ObjectAdapterPtr&) {
        IceUtil::Mutex::Lock lock(_namesMutex);

        set<string>::iterator i = _names.find(name);
        if (i != _names.end())
            _names.erase(i);
    }

private:
    IceUtil::Mutex _namesMutex;
    set<string> _names;
};
```

The idea is to have a set of names of existing entries, and to update that set in `create` and `destroy` as appropriate. However, for threaded servers, that raises a concurrency issue: if we have clients that can concurrently call `create`, `destroy`, and `list`, we need to interlock these operations to avoid corrupting the `_names` set (because STL containers are not thread-safe). This is the purpose of the mutex `_namesMutex` in the factory: `create`, `destroy`, and `list` can each lock this mutex to ensure exclusive access to the `_names` set.

Another issue is that our implementation of `destroy` must update the set of entries that is maintained by the factory. This is the purpose of the `remove` member function: it removes the specified name from the `_names` set (of course, under protection of the `_namesMutex` lock). However, `destroy` is a method on the `PhoneEntryI` servant, whereas `remove` is a method on the factory, so the servant must know how to reach the factory. Because the factory is a singleton, we can fix this by adding a static `_factory` member to the `PhoneEntryI` class:

```
class PhoneEntryI : public PhoneEntry {
public:
    // ...

    static PhoneEntryFactoryIPtr _factory;

private:
    const string _name;
    string _phNum;
};
```

The code in `main` then creates the factory and initializes the static member variable, for example:

```
PersonI::_factory = new PersonFactoryI;

// Add factory to ASM and activate object
// adapter here...
```

This works, but it leaves a bad taste in our mouth because it sets up a cyclic dependency between the phone entry servants and the factory: the factory knows about the servants, and each servant knows about the factory so it can call `remove` on the factory. In general, such cyclic dependencies are a bad idea: if nothing else, they make a design harder to understand.

We could remove the cyclic dependency by moving the `_names` set and its associated mutex into a separate class instance that is referenced from both `PhoneEntryFactoryI` and `PhoneEntryI`. That would get rid of the cyclic dependency as far as the C++ type system is concerned but, as we will see later, it would not really help because the factory and its servants turn out to be mutually dependent regardless (because of concurrency issues). So, for the moment, we'll stay with this design and examine better alternatives after we have explored the concurrency issues in more detail.

With this design, we can implement `list` as follows:

```
PhoneEntries
PhoneEntryFactoryI::list(const Current& c)
{
    Mutex::Lock lock(_namesMutex);

    CommunicatorPtr comm = c.adapter->getCommunicator();

    PhoneEntries pe;
    set<string>::const_iterator i;
    for (i = _names.begin(); i != _names.end(); ++i) {
        ObjectPrx o = c.adapter->createProxy(
                                comm->stringToIdentity(name));
        pe.push_back(PhoneEntryPrx::uncheckedCast(o));
    }

    return pe;
}
```

Note that `list` acquires a lock on the mutex, to prevent concurrent modification of the `_names` set by `create` and `destroy`. In turn, our `create` implementation now also locks the mutex:

```
PhoneEntryPrx
PhoneEntryFactory::create(const string& name,
                          const string& phNum,
                          const Current& c)
{
    Mutex::Lock lock(_namesMutex);

    PhoneEntryPrx pe;
    try {
        CommunicatorPtr comm = c.adapter->getCommunicator();
        PhoneEntryPtr servant = new PhoneEntryI(name, phNum);
        pe = PhoneEntryPrx::uncheckedCast(
            c.adapter->add(servant,
                           comm->stringToIdentity(name)));
    } catch (const Ice::AlreadyRegisteredException&) {
        throw PhoneEntryExists(name, phNum);
    }
    _names.insert(name);

    return pe;
}
```

With this implementation, we are safe if `create` and `list` run concurrently: only one of the two operations can acquire the lock at a time, so there is no danger of corrupting the `_names` set.

`destroy` is now trivial to implement: it simply removes the ASM entry and calls `remove` on the factory:

```
void
PhoneEntryI::destroy(const Current& c)
{
    // Note: not quite correct yet.
    c.adapter->remove(_name);
    _factory->remove(_name, c.adapter);
}
```

Note that this is not quite correct because we have not yet considered concurrency issues for `destroy`. We will consider this issue in the following section.

34.6.5 Life Cycle and Normal Operations

So far, we have mostly ignored the implementations of `getNumber` and `setNumber`. Obviously, `getNumber` and `setNumber` must be interlocked against concurrent access—without this interlock, concurrent requests from

clients could result in one thread writing to the `_phNum` member while another thread is reading it, with unpredictable results. (Conversely, the `name` operation need not have an interlock because the name of a phone entry is immutable.) To interlock `getNumber` and `setNumber`, we can add a mutex `_m` to `PhoneEntryI`:

```
class PhoneEntryI : public PhoneEntry {
public:
    // ...

    static PhoneEntryFactoryIPtr _factory;

private:
    const string _name;
    string _phNum;
    Mutex  _m;
};
```

The `getNumber` and `setNumber` implementations then lock `_m` to protect `_phNum` from concurrent access:

```
string
PhoneEntryI::name(const Current&)
{
    return _name;
}

string
PhoneEntryI::getNumber(const Current&)
{
    // Incorrect implementation!

    Mutex::Lock lock(_m);

    return _phNum;
}

void
PhoneEntryI::setNumber(const string& phNum, const Current&)
{
    // Incorrect implementation!

    Mutex::Lock lock(_m);

    _phNum = phNum;
}
```

This looks good but, as it turns out, `destroy` throws a spanner in the works: as shown, this code suffers from a rare, but real, race condition. Consider the situation where a client calls `destroy` at the same time as another client calls `setNumber`. In a server with a thread pool with more than one thread, the calls can be dispatched in separate threads and can therefore execute concurrently.

The following sequence of events can occur:

- The thread dispatching the `setNumber` call locates the servant, enters the operation implementation, and is suspended by the scheduler immediately on entry to the operation, before it can lock `_m`.
- The thread dispatching the `destroy` call locates the servant, enters `destroy`, successfully removes the servant from the ASM and the `_names` set, and returns.
- The thread that was suspended in `setNumber` is scheduled again, locks `_m`, and now operates on a conceptually already-destroyed Ice object.

Note that, even if we lock `_m` in destroy, the problem persists:

```
void
PhoneEntryI::destroy(const Current& c)
{
    // Note: still not correct.

    IceUtil::Mutex::Lock lock(_m);

    c.adapter->remove(_name);
    _factory->remove(_name, c.adapter);
}
```

Even though destroy now locks `_m` and so cannot run concurrently with `getNumber` and `setNumber`, the preceding scenario can still arise. The problem here is that a thread can enter the servant and be suspended before it gets a chance to acquire a lock. With the code as it stands, this is not a problem: `setNumber` will simply update the `_phNum` member variable in a servant that no longer has an ASM entry. In other words, the Ice object is already destroyed—it just so happens that the servant for that Ice object is still hanging around because there is still an operation executing inside it. Any updates to the servant will succeed (even though they are useless because the servant's destructor will run as soon as the last invocation leaves the servant.)

Note that this scenario is not unique to C++ and can arise even with Java synchronized operations: in that case, a thread can be suspended just after the Ice run time has identified the target servant, but before it actually calls the operation

on the target servant. While the thread is suspended, another thread can execute `destroy`.

While this race condition does not affect our implementation, it *does* affect more complex applications, particularly if the servant modifies external state, such as a file system or database. For example, `setNumber` could modify a file in the file system; in that case, `destroy` would delete that file and probably close a file descriptor or stream. If we were to allow `setNumber` to continue executing after `destroy` has already done its job, we would likely encounter problems: `setNumber` might not find the file where it expects it to be or try to use the closed file descriptor and return an error; or worse, `setNumber` might end up re-creating the file in the process of updating the already-destroyed entry's phone number. (What exactly happens depends on how we write the code for each operation.)

Of course, we can try to anticipate these scenarios and handle the error conditions appropriately, but doing this for complex systems with complex servants rapidly gets out of hand: in each operation, we would have to ask ourselves what might happen if the servant is destroyed concurrently and, if so, take appropriate recovery action.

It is preferable to instead deal with interleaved invocations of `destroy` and other operations in a systematic fashion. We can do this by adding a `_destroyed` member to the `PhoneEntryI` servant. This member is initialized to false by the constructor and set to true by `destroy`. On entry to every operation (including `destroy`), we lock the mutex, test the `_destroyed` flag, and throw `ObjectNotExistException` if the flag is set:

```
class PhoneEntryI : public PhoneEntry {
public:
    // ...

    static PhoneEntryFactoryIPtr _factory;

private:
    const string _name;
    string _phNum;
    bool   _destroyed;
    Mutex  _m;
};

PhoneEntryI::PhoneEntryI(const string& name, const string& phNum)
    : _name(name), _phNum(phNum), _destroyed(false)
{
}
```

```
string
PhoneEntryI::name(const Current&)
{
    Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);

    return _name;
}

string
PhoneEntryI::getNumber(const Current&)
{
    Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);

    return _phNum;
}

void
PhoneEntryI::setNumber(const string& phNum, const Current&)
{
    Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);

    _phNum = phNum;
}

void
PhoneEntryI::destroy(const Current& c)
{
    Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);

    _destroyed = true;
    c.adapter->remove(_name);
    _factory->remove(_name, c.adapter); // Dubious!
}
```

If you are concerned about the repeated code on entry to every operation, you can put that code into a member function or base class to make it reusable (although the benefits of doing so are probably too minor to make this worthwhile).

Using the `_destroyed` flag, if an operation is dispatched and suspended before it can lock the mutex and, meanwhile, `destroy` runs to completion in another thread, it becomes impossible for an operation to operate on the state of such a "zombie" servant: the test on entry to each operation ensures that any operation that runs after `destroy` immediately raises `ObjectNotExistException`.

Also note the "dubious" comment in `destroy`: the operation first locks `_m` and, while holding that lock, calls `remove` on the factory, which in turn locks its own `_namesMutex`. This is not wrong as such, but as we will see shortly, it can easily lead to deadlocks if we modify the application later.

34.7 Removing Cyclic Dependencies

In Section 34.6.4, we mentioned that factoring the `_names` set and its mutex into a separate class instance does not really solve the cyclic dependency problem, at least not in general. To see why, suppose that we want to extend our factory with a new `getDetails` operation:

```
// ...

struct Details {
    PhoneEntry* proxy;
    string name;
    string phNum;
};

sequence<Details> DetailsSeq;

interface PhoneEntryFactory {
    // ...

    DetailsSeq getDetails();
};
```

This type of operation is common in collection managers: instead of returning a simple list of proxies, `getDetails` returns a sequence of structures, each of which contains not only the object's proxy, but also some of the state of the corresponding object. The motivation for this is performance: with a plain list of

proxies, the client, once it has obtained the list, is likely to immediately follow up with one or more remote calls for each object in the list in order to retrieve their state (for example, to display the list of objects to the user). Making all these additional remote procedure calls is inefficient, and an operation such as `getDetails` gets the job done with a single RPC instead.

To implement `getDetails` in the factory, we need to iterate over the set of entries and invoke the `getNumber` operation on each object. (These calls are collocated and therefore very efficient, so they do not suffer the performance problem that a client calling the same operations would suffer.) However, this is potentially dangerous because the following sequence of events is possible:

- Client A calls `getDetails`.
- The implementation of `getDetails` must lock `_namesMutex` to prevent concurrent modification of the `_names` set during iteration.
- Client B calls `destroy` on a phone entry.
- The implementation of `destroy` locks the entry's mutex `_m`, sets the `_destroyed` flag, and then calls `remove`, which attempts to lock `_namesMutex` in the factory. However, `_namesMutex` is already locked by `getDetails`, so `remove` blocks until `_m` is unlocked again.
- `getDetails`, while iterating over its set of entries, happens to call `getNumber` on the entry that is currently being destroyed by client B. `getNumber`, in turn, tries to lock its mutex `_m`, which is already locked by `destroy`.

At this point, the server deadlocks: `getDetails` holds a lock on `_namesMutex` and waits for `_m` to become available, and `destroy` holds a lock on `_m` and waits for `_namesMutex` to become available, so neither thread can make progress.

To get rid of the deadlock, we have two options:

- Rearrange the locking such that deadlock becomes impossible.
- Abandon the idea of calling back from the servants into the factory and use *reaping* instead.

We will explore both options in the next two sections.

34.7.1 Deadlock-Free Lock Acquisition

For our example, it is fairly easy to avoid the deadlock: instead of holding the lock for the duration of `destroy`, we set the `_destroyed` flag under protection of the lock and unlock `_m` again before calling `remove` on the factory:

```
void
PhoneEntryI::destroy(const Current& c)
{
    {
        Mutex::Lock lock(_m);

        if (_destroyed)
            throw ObjectNotExistException(__FILE__, __LINE__);

        _destroyed = true;

    } // _m is unlocked here.

    _factory->remove(_name, c.adapter);
}
```

Now deadlock is impossible because no function holds more than one lock, and no function calls another function while it holds a lock. However, rearranging locks in this fashion can be quite difficult for complex applications. In particular, if an application uses callbacks that do complex things involving several objects, it can be next to impossible to prove that the code is free of deadlocks. The same is true for applications that use condition variables and suspend threads until a condition becomes true.

At the core of the problem is that concurrency can create circular locking dependencies: an operation on the factory (such as `getDetails`) can require the same locks as a concurrent call to `destroy`. This is one reason why threaded code is harder to write than sequential code—the interactions among operations require locks, but dependencies among these locks are not obvious. In effect, locks set up an entirely separate and largely invisible set of dependencies. For example, it was easy to spot the mutual dependency between the factory and the servants due to the presence of `remove`; in contrast, it was much harder to spot the lurking deadlock in `destroy`. Worse, deadlocks may not be found during testing and discovered only after deployment, when it is much more expensive to rectify the problem.

34.7.2 Reaping

Instead of trying to arrange code such that it is deadlock-free in the presence of callbacks, it is often easier to change the code to avoid the callbacks entirely and to use an approach known as *reaping*:

- `destroy` marks the servant as destroyed and removes the ASM entry as usual, but it does not call back into the factory to update the `_names` set.
- Whenever a collection manager operation, such as `list`, `getDetails`, or `find` is called, the factory checks for destroyed servants and, if it finds any, removes them from the `_names` set.

Reaping can make for a much cleaner design because it avoids both the cyclic type dependency and the cyclic locking dependency.

A Simple Reaping Implementation

To implement reaping, we need to change our `PhoneEntryI` definition a little. It no longer has a static `_factory` smart pointer back to the factory (because it no longer calls `remove`). Instead, the servant now provides a member function `_isZombie` that the factory calls to check whether the servant was destroyed some time in the past:

```
class PhoneEntryI : public PhoneEntry {
public:
    // ...

    bool _isZombie() const;

private:
    const string _name;
    string _phNum;
    bool   _destroyed;
    Mutex  _m;
};
```

The implementation of `_isZombie` is trivial: it returns the `_destroyed` flag under protection of the lock:

```
bool
PhoneEntryI::_isZombie() const
{
    Mutex::Lock lock(_m);

    return _destroyed;
}
```

The `destroy` operation no longer calls back into the factory to update the `_names` set; instead, it simply sets the `_destroyed` flag and removes the ASM entry:

```
void
PhoneEntryI::destroy(const Current& c)
{
    Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);

    _destroyed = true;
    c.adapter->remove(c.id);
}
```

The factory now, instead of storing just the names of existing servants, maintains a map that maps the name of each servant to its smart pointer:

```
class PhoneEntryFactoryI : public PhoneEntryFactory
{
public:
    // Constructor and Slice operations here...

private:
    typedef map<string, PhoneEntryIPtr> PMap;
    PMap _entries;
    Mutex _entriesMutex;
};
```

During `create` (and other operations, such as `list`, `getDetails`, and `find`), we scan for zombie servants and remove them from the `_entries` map:

```
PhoneEntryPrx
PhoneEntryFactory::create(const string& name,
                          const string& phNum,
                          const Current& c)
{
    Mutex::Lock lock(_entriesMutex);

    PhoneEntryPrx pe;
    PhoneEntryIPtr servant = new PhoneEntryI(name, phNum);

    // Try to create new object.
    //
    try {
        CommunicatorPtr comm = c.adapter->getCommunicator();
```

```
        pe = PhoneEntryPrx::uncheckedCast(c.adapter->add(
                                servant,
                                comm->stringToIdentity(name)));
    } catch (const Ice::AlreadyRegisteredException&) {
        throw PhoneEntryExists(name, phNum);
    }

    // Scan for zombies.
    //
    PMap::iterator i = _entries.begin();
    while (i != _entries.end())
    {
        if (i->second->_isZombie())
            _entries.erase(i++);
        else
            ++i;
    }
    _entries[name] = servant;

    return pe;
}
```

The implementations of `list`, `getDetails`, and `find` scan for zombies as well. Because they need to iterate over the existing entries anyway, reaping incurs essentially no extra cost:

```
PhoneEntries
PhoneEntryFactoryI::list(const Current& c)
{
    Mutex::Lock lock(_entriesMutex);

    CommunicatorPtr comm = c.adapter->getCommunicator();

    PhoneEntries pe;
    PMap::iterator i = _entries.begin();
    for (i = _entries.begin(); i != _entries.end(); ++i) {
        if (i->second->_isZombie()) {
            _entries.erase(i++);
        } else {
            ObjectPrx o = c.adapter->createProxy(
                                comm->stringToIdentity(i->first));
            pe.push_back(PhoneEntryPrx::uncheckedCast(o));
            ++i;
        }
    }
```

```
    return pe;
}

// Similar for getDetails and find...
```

This is a much cleaner design: there is no cyclic dependency between the factory and the servants, either implicit (in the type system) or explicit (as a locking dependency). Moreover, the implementation is easier to understand once you get used to the idea of reaping: there is no need to follow complex callbacks and to carefully analyze the order of lock acquisition. (Note that, depending on how state is maintained for servants, you may also need to reap during start-up and shutdown.)

In general, we recommend that you use a reaping approach in preference to callbacks for all but the most trivial applications: it simply is a better approach that is easier to maintain and understand.

Alternative Reaping Implementations

You may be concerned that reaping increases the cost of `create` from $O(\log n)$ to $O(n)$ because `create` now iterates over all existing entries and locks and unlocks a mutex in each servant (whereas, previously, it simply added each new servant to the `_names` set). Often, this is not an issue because life cycle operations are called infrequently compared to normal operations. However, you will notice the additional cost if you have a large number of servants (in the thousands or more) and life cycle operations are called frequently.

If you find that `create` is a bottleneck (by profiling, not by guessing!), you can change to a more efficient implementation by adding zombie servants to a separate zombie list. Reaping then iterates over the zombie list and removes each servant in the zombie list from the `_entries` map before clearing the zombie list. This reduces the cost of reaping to be proportional to the number of *zombie* servants instead of the *total* number of servants. In addition, it allows us to remove the `_isZombie` member function and to lock and unlock `_entriesMutex` only once instead of locking a mutex in each servant as part of `_isZombie`. We will see such an implementation in Section 34.9.

You may also be concerned about the number of zombie servants that can accumulate in the server if `create` is not called for some time. For most applications, this is not a problem: the servants occupy memory, but no other resources because `destroy` can clean up scarce resources, such as file descriptors or network connections before it turns the servant into a zombie. If you really need to prevent accumulation of zombie servants, you can reap from a background thread

that runs periodically, or you can count the number of zombies and trigger a reaping pass once that number exceeds some threshold.

34.8 Object Identity and Uniqueness

In Section 34.5.1, we mentioned that it may not be a good idea to allow clients to control the object identity of the Ice objects they create. Here is the scenario from Section 34.2 once more, re-cast in terms of our phone book application:

1. Client A creates a new phone entry for Fred.
2. Client A passes the proxy for the Fred entry as a parameter of a remote call to another part of the system, say, server B.
3. Server B remembers Fred's proxy.
4. Client A decides that the entry for Fred is no longer needed and calls Fred's `destroy` operation.
5. Some time later, client C creates a new phone entry for a different person whose name also happens to be Fred.
6. Server B decides to get Fred's phone number by calling `getNumber` on the proxy it originally obtained from client A.

At this point, things are likely to go wrong: server B thinks that it has obtained the phone number of the original Fred, but that entry no longer exists and has since been replaced by a new entry for a different person (who presumably has a different phone number).

What has happened here is that Fred has been reincarnated because the same object identity was used for two different objects. In general, such reused object identities are a bad idea. For example, consider the following interfaces:

```
interface Process {
    void launch(); // Start process.
};

interface Missile {
    void launch(); // Kill lots of people.
};
```

Replaying the preceding scenario, if client A creates a `Process` object called "Thunderbird" and destroys that object again, and client C creates a `Missile` object called "Thunderbird", when server B calls `launch`, it will launch a missile instead of a process.

To be fair, in reality, this scenario is unlikely because it tacitly assumes that both objects are implemented by the same object adapter but, in a realistic scenario, the same server would be unlikely to implement both launching of processes and missiles. However, if you have objects that derive from common base interfaces, so objects of different types share the same operation names, this problem is real: operation invocations can easily end up in a destroyed and later recreated object.

Specifically, the preceding scenario illustrates that, when the Ice run time dispatches a request, exactly three items determine where the request ends up being processed:

- the endpoint at which the server listens for incoming requests
- the identity of the Ice object that is the target of the request
- the name of the operation that is to be invoked on the Ice object

If object identities are insufficiently unique, a request intended for one object can end up being sent to a completely different object, provided that the original object used the same identity, that both provide an operation with the same name, and that the parameters passed to one operation happen to decode correctly when interpreted as the parameters to the other operation. (This is rare, but not impossible, depending on the type and number of parameters.)

The crucial question is, what do we mean by "insufficiently unique"? As far as the call dispatch is concerned, identities must be unique only per object adapter. This is because the ASM does not allow you to add two entries with the same object identity; by enforcing this, the ASM ensures that each object identity belongs to exactly one servant. (Note that the converse, namely, that servants in ASM entries must be unique, is *not* the case: the ASM allows you to map different object identities to the same servant, which is useful to, for example, implement stateless facade objects—see Section 32.9.2.) So, as far as the Ice run time is concerned, it is perfectly OK to reuse object identities for different Ice objects.

Note that the Ice run time cannot prevent reuse of object identities either. Doing so would require the run time to remember every object identity that has ever been used, which does not scale. Instead, the Ice run time makes the application responsible for ensuring that object identities are "sufficiently unique".

You can deal with the identity reuse problem in several ways. One option is to do nothing and simply ignore the problem. While this sounds facetious, it is a viable option for many applications because, due to their nature, identity reuse is simply impossible. For example, if you use a social security number as a person's identity, the problem cannot arise because the social security number of a deceased person is not given to another person.

Another option is to allow identity reuse and to write your application such that it can deal with such identities: if nothing bad happens when an identity is reused, there is no problem. (This is the case if you know that the life cycles of the proxies for two different objects with the same identity can never overlap.)

The third option is to ensure that object identities are guaranteed unique, for example, by establishing naming conventions that make reuse impossible, or by using a UUID as the object identity (see Section 32.4.4). This can be useful even for applications for which identity reuse does not pose a problem. For example, if you use IceGrid well-known proxies (see Section 38.6), globally-unique object identities allow you to move a server to a different machine without invalidating proxies to these objects that are held by clients.

In general, we recommend that if an Ice object naturally contains a unique item of state (such as a social security number), you should use that item as the object identity. On the other hand, if the natural object identity is insufficiently unique (as is the case with names of phone book entries), you should use a UUID as the identity. (This is particularly useful for anonymous transient objects, such as session objects, that may not have a natural identity.)

34.9 Object Life Cycle for the File System Application

Now that we have had a look at the issues around object life cycle, let us return to our file system application and add life cycle operations to it, so clients can create and destroy files and directories.

To destroy a file or directory, the obvious choice is to add a `destroy` operation to the `Node` interface:

```
module Filesystem {

    exception GenericError {
        string reason;
    };
    exception PermissionDenied extends GenericError {};
    exception NameInUse extends GenericError {};
    exception NoSuchName extends GenericError {};

    interface Node {
        idempotent string name();
        void destroy() throws PermissionDenied;
```

```
    };

    // ...
};
```

Note that `destroy` can throw a `PermissionDenied` exception. This is necessary because we must prevent attempts to destroy the root directory.

The `File` interface is the same as the one we saw in Chapter 5:

```
module Filesystem {
    // ...

    sequence<string> Lines;

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    // ...
};
```

Note that, because `File` derives from `Node`, it inherits the `destroy` operation we defined for `Node`.

The `Directory` interface now looks somewhat different from the previous version:

- The `list` operation returns a sequence of structures instead of a list of proxies: for each entry in a directory, the `NodeDesc` structure provides the name, type, and proxy of the corresponding file or directory.
- Directories provide a `find` operation that returns the description of the nominated node. If the nominated node does not exist, the operation throws a `NoSuchName` exception.
- The `createFile` and `createDirectory` operations create a file and directory, respectively. If a file or directory already exists, the operations throw a `NameInUse` exception.

Here are the corresponding definitions:

```
module Filesystem {
    // ...

    enum NodeType { DirType, FileType };

    struct NodeDesc {
```

```
        string name;
        NodeType type;
        Node* proxy;
    };

    sequence<NodeDesc> NodeDescSeq;

    interface Directory extends Node {
        idempotent NodeDescSeq list();
        idempotent NodeDesc find(string name) throws NoSuchName;
        File* createFile(string name) throws NameInUse;
        Directory* createDirectory(string name) throws NameInUse;
    };
};
```

Note that this design is somewhat different from the factory design we saw in Section 34.5.1. In particular, we do not have a single object factory; instead, we have as many factories as there are directories, that is, each directory creates files and directories only in that directory.

The motivation for this design is twofold:

- Because all files and directories that can be created are immediate descendants of their parent directory, we avoid the complexities of parsing path names for a separator such as "/". This keeps our example code to manageable size. (A real-world implementation of a distributed file system would, of course, be able to deal with path names.)
- Having more than one object factory presents interesting implementation issues that we will explore in the remainder of this chapter.

The following two sections describe the implementation of this design in C++ and Java. You can find the full code of the implementation (including languages other than C++ and Java) in the `demo/book/lifecycle` directory of your Ice distribution.

34.9.1 Implementing Object Life Cycle in C++

The implementation of our life cycle design has the following characteristics:

- It uses UUIDs as the object identities for nodes. This avoids the object reincarnation problems we discussed in Section 34.8.
- When `destroy` is called on a node, the node needs to destroy itself and inform its parent directory that it has been destroyed (because the parent directory is the node's factory and also acts as a collection manager for child nodes).

Note that, in contrast to the code in Chapter 9, the entire implementation resides in a `FilesystemI` namespace instead of being part of the `Filesystem` namespace. Doing this is not essential, but is a little cleaner because it keeps the implementation in a namespace that is separate from the Slice-generated namespace.

The `NodeI` Base Class

To begin with, let us look at the definition of the `NodeI` class:

```
namespace FilesystemI {

    class DirectoryI;
    typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

    class NodeI : public virtual Filesystem::Node {
    public:
        virtual std::string name(const Ice::Current&);
        Ice::Identity id() const;

    protected:
        NodeI(const std::string& name,
              const DirectoryIPtr& parent);

        const std::string _name;
        const DirectoryIPtr _parent;
        bool _destroyed;
        Ice::Identity _id;
        IceUtil::Mutex _m;
    };

    // ...
}
```

The purpose of the `NodeI` class is to provide the data and implementation that are common to both `FileI` and `DirectoryI`, which use implementation inheritance from `NodeI`.

As in Chapter 9, `NodeI` provides the implementation of the `name` operation and stores the name of the node and its parent directory in the `_name` and `_parent` members. (The root directory's `_parent` member is null.) These members are immutable and initialized by the constructor and, therefore, `const`.

The `_destroyed` member, protected by the mutex `_m`, prevents the race condition we discussed in Section 34.6.5. The constructor initializes

_destroyed to false and creates an identity for the node (stored in the _id member):

```
FilesystemI::NodeI::NodeI(const string& name,
                          const DirectoryIPtr& parent)
    : _name(name), _parent(parent), _destroyed(false)
{
    _id.name = parent ? IceUtil::generateUUID() : "RootDir";
}
```

The id member function returns a node's identity, stored in the _id data member. The node must remember this identity because it is a UUID and is needed when we create a proxy to the node:

```
Identity
FilesystemI::NodeI::id() const
{
    return _id;
}
```

The data members of NodeI are protected instead of private to keep them accessible to the derived FileI and DirectoryI classes. (Because the implementation of NodeI and its derived classes is quite tightly coupled, there is little point in making these members private and providing separate accessors and mutators for them.)

The implementation of the Slice name operation simply returns the name of the node, but also checks whether the node has been destroyed, as described in Section 34.6.5:

```
string
FilesystemI::NodeI::name(const Current&)
{
    IceUtil::Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);

    return _name;
}
```

This completes the implementation of the NodeI base class.

The DirectoryI Class

Next, we need to look at the implementation of directories. The DirectoryI class derives from NodeI and the Slice-generated Directory skeleton class.

Of course, it must implement the pure virtual member functions for its Slice operations, which leads to the following (not yet complete) definition:

```
namespace FilesystemI {

    // ...

    class DirectoryI : virtual public NodeI,
                       virtual public Filesystem::Directory {
    public:
        virtual Filesystem::NodeDescSeq list(const Ice::Current&);
        virtual Filesystem::NodeDesc find(const std::string&,
                                          const Ice::Current&);
        Filesystem::FilePrx createFile(const std::string&,
                                       const Ice::Current&);
        Filesystem::DirectoryPrx
                        createDirectory(const std::string&,
                                        const Ice::Current&);
        virtual void destroy(const Ice::Current&);
        // ...

    private:
        // ...
    };
}
```

Each directory stores its contents in a map that maps the name of a directory to its servant:

```
namespace FilesystemI {

    // ...

    class DirectoryI : virtual public NodeI,
                       virtual public Filesystem::Directory {
    public:
        // ...

        DirectoryI(const ObjectAdapterPtr& a,
                   const std::string& name,
                   const DirectoryIPtr& parent = 0);

        void removeEntry(const std::string& name);

    private:
        typedef std::map<std::string, NodeIPtr> Contents;
```

```
        Contents _contents;
        // ...
    };
}
```

Note that we use the inherited member `_m` to interlock operations.

The constructor simply initializes the `NodeI` base class:

```
FilesystemI::DirectoryI::DirectoryI(const string& name,
                                    const DirectoryIPtr& parent)
    : NodeI(name, parent)
{
}
```

The `removeEntry` member function is called by the child to remove itself from its parent's `_contents` map:

```
void
FilesystemI::DirectoryI::removeEntry(const string& name)
{
    IceUtil::Mutex::Lock lock(_m);
    Contents::iterator i = _contents.find(name);
    if(i != _contents.end())
    {
        _contents.erase(i);
    }
}
```

Here is the `destroy` member function for directories:

```
void
FilesystemI::DirectoryI::destroy(const Current& c)
{
    if (!_parent)
        throw PermissionDenied("Cannot destroy root directory");

    {
        IceUtil::Mutex::Lock lock(_m);

        if (_destroyed)
            throw ObjectNotExistException(__FILE__, __LINE__);

        if (!_contents.empty())
            throw PermissionDenied("Cannot destroy non-empty direc
tory");

        c.adapter->remove(id());
        _destroyed = true;
```

```
    }

    _parent->removeEntry(_name);
}
```

The code first prevents destruction of the root directory and then checks whether this directory was destroyed previously. It then acquires the lock and checks that the directory is empty. Finally, `destroy` removes the ASM entry for the destroyed directory and removes itself from its parent's `_contents` map. Note that, for the reason we explained in Section 34.7.1, we call `removeEntry` outside the synchronization.

The `createDirectory` implementation locks the mutex before checking whether the directory already contains a node with the given name (or an invalid empty name). If not, it creates a new servant, adds it to the ASM and the `_contents` map, and returns its proxy:

```
DirectoryPrx
FilesystemI::DirectoryI::createDirectory(const string& name,
                                         const Current& c)
{
    IceUtil::Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);

    if (name.empty() || _contents.find(name) != _contents.end())
        throw NameInUse(name);

    DirectoryIPtr d = new DirectoryI(name, this);
    ObjectPrx node = c.adapter->add(d, d->id());
    _contents[name] = d;
    return DirectoryPrx::uncheckedCast(node);
}
```

The `createFile` implementation is identical, except that it creates a file instead of a directory:

```
FilePrx
FilesystemI::DirectoryI::createFile(const string& name,
                                    const Current& c)
{
    IceUtil::Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);
```

```
    if (name.empty() || _contents.find(name) != _contents.end())
        throw NameInUse(name);

    FileIPtr f = new FileI(name, this);
    ObjectPrx node = c.adapter->add(f, f->id());
    _contents[name] = f;
    return FilePrx::uncheckedCast(node);
}
```

Here is the implementation of `list`:

```
NodeDescSeq
FilesystemI::DirectoryI::list(const Current& c)
{
    IceUtil::Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);

    NodeDescSeq ret;
    for (Contents::const_iterator i = _contents.begin();
         i != _contents.end(); ++i)
    {
        NodeDesc d;
        d.name = i->first;
        d.type = FilePtr::dynamicCast(i->second)
            ? FileType : DirType;
        d.proxy = NodePrx::uncheckedCast(
            c.adapter->createProxy(i->second->id()));
        ret.push_back(d);
    }
    return ret;
}
```

After acquiring the lock, the code iterates over the directory's contents and adds a `NodeDesc` structure for each entry to the returned vector.

The `find` operation proceeds along similar lines:

```
NodeDesc
FilesystemI::DirectoryI::find(const string& name,
                              const Current& c)
{
    IceUtil::Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);
```

```
    Contents::const_iterator pos = _contents.find(name);
    if (pos == _contents.end())
        throw NoSuchName(name);

    NodeIPtr p = pos->second;
    NodeDesc d;
    d.name = name;
    d.type = FilePtr::dynamicCast(p) ? FileType : DirType;
    d.proxy = NodePrx::uncheckedCast(
        c.adapter->createProxy(p->id()));
    return d;
}
```

The `FileI` Class

The constructor of `FileI` is trivial: it simply initializes the data members of its base class::

```
FilesystemI::FileI::FileI(const string& name,
                          const DirectoryIPtr& parent)
    : NodeI(name, parent)
{
}
```

The implementation of the three member functions of the `FileI` class is also trivial, so we present all three member functions here:

```
Lines
FilesystemI::FileI::read(const Current&)
{
    IceUtil::Mutex::Lock lock(_m);

    if (_destroyed)
        throw ObjectNotExistException(__FILE__, __LINE__);

    return _lines;
}

// Slice File::write() operation.

void
FilesystemI::FileI::write(const Lines& text, const Current&)
{
    IceUtil::Mutex::Lock lock(_m);

    if (_destroyed)
```

```
        throw ObjectNotExistException(__FILE__, __LINE__);

    _lines = text;
}

void
FilesystemI::FileI::destroy(const Current& c)
{
    {
        IceUtil::Mutex::Lock lock(_m);

        if (_destroyed)
            throw ObjectNotExistException(__FILE__, __LINE__);

        c.adapter->remove(id());
        _destroyed = true;
    }

    _parent->removeEntry(_name);
}
```

Concurrency Considerations

The preceding implementation is provably deadlock free. All member functions hold only one lock at a time, so they cannot deadlock with each other or themselves. While the locks are held, the functions do not call other member functions that acquire locks, so any potential deadlock can only arise by concurrent calls to another mutating function, either on the same node or on different nodes. For concurrent calls on the same node, deadlock is impossible because such calls are strictly serialized on the mutex `_m`; for concurrent calls to `destroy` on different nodes, each node locks its respective mutex `_m`, releases `_m` again, and then acquires and releases a lock on its parent (by calling `removeEntry`), also making deadlock impossible.

34.9.2 Implementing Object Life Cycle in Java

The implementation of our life cycle design has the following characteristics:

- It uses UUIDs as the object identities for nodes. This avoids the object reincarnation problems we discussed in Section 34.8.
- When `destroy` is called on a node, the node needs to destroy itself and inform its parent directory that it has been destroyed (because the parent directory is the node's factory and also acts as a collection manager for child nodes).

Note that, in contrast to the code in Chapter 13, the entire implementation resides in a `FilesystemI` package instead of being part of the `Filesystem` package. Doing this is not essential, but is a little cleaner because it keeps the implementation in a package that is separate from the Slice-generated package.

The `NodeI` Interface

Our `DirectoryI` and `FileI` servants derive from a common `NodeI` base interface. This interface is not essential, but useful because it allows us to treat servants of type `DirectoryI` and `FileI` polymorphically:

```
package FilesystemI;

public interface NodeI
{
    Ice.Identity id();
}
```

The only method is the `id` method, which returns the identity of the corresponding node.

The `DirectoryI` Class

As in Chapter 13, the `DirectoryI` class derives from the generated base class `_DirectoryDisp`. In addition, the class implements the `NodeI` interface. `DirectoryI` must implement each of the Slice operations, leading to the following outline:

```
package FilesystemI;

import Ice.*;
import Filesystem.*;

public class DirectoryI extends _DirectoryDisp
                        implements NodeI
{
    public Identity
    id();

    public synchronized String
    name(Current c);

    public synchronized NodeDesc[]
    list(Current c);

    public synchronized NodeDesc
```

```
    find(String name, Current c) throws NoSuchName;

    public synchronized FilePrx
    createFile(String name, Current c) throws NameInUse;

    public synchronized DirectoryPrx
    createDirectory(String name, Current c) throws NameInUse;

    public void
    destroy(Current c) throws PermissionDenied;

    // ...
}
```

To support the implementation, we also require a number of methods and data members:

```
package FilesystemI;

import Ice.*;
import Filesystem.*;

public class DirectoryI extends _DirectoryDisp
                        implements NodeI
{
    // ...

    public DirectoryI();
    public DirectoryI(String name, DirectoryI parent);

    public synchronized void
    removeEntry(String name);

    private String _name;       // Immutable
    private DirectoryI _parent; // Immutable
    private Identity _id;       // Immutable
    private boolean _destroyed;
    private java.util.Map<String, NodeI> _contents;
}
```

The `_name` and `_parent` members store the name of this node and a reference to the node's parent directory. (The root directory's `_parent` member is null.) Similarly, the `_id` member stores the identity of this directory. The `_name`, `_parent`, and `_id` members are immutable once they have been initialized by the constructor. The `_destroyed` member prevents the race condition we discussed in Section 34.6.5; to interlock access to `_destroyed` (as well as the

`_contents` member) we can use synchronized methods (as for the `name` method), or use a `synchronized(this)` block.

The `_contents` map records the contents of a directory: it stores the name of an entry, together with a reference to the child node.

Here are the two constructors for the class:

```
public DirectoryI()
{
    this("/", null);
}

public DirectoryI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;
    _id = new Identity();
    _destroyed = false;
    _contents = new java.util.HashMap<String, NodeI>();

    _id.name = parent == null ? "RootDir" : Util.generateUUID();
}
```

The first constructor is a convenience function to create the root directory with the fixed identity "RootDir" and a null parent.

The real constructor initializes the `_name`, `_parent`, `_id`, `_destroyed`, and `_contents` members. Note that nodes other than the root directory use a UUID as the object identity.

The `removeEntry` method is called by the child to remove itself from its parent's `_contents` map:

```
public synchronized void
removeEntry(String name)
{
    _contents.remove(name);
}
```

The implementation of the Slice `name` operation simply returns the name of the node, but also checks whether the node has been destroyed, as described in Section 34.6.5:

```
public synchronized String
name(Current c)
{
    if (_destroyed)
        throw new ObjectNotExistException();
    return _name;
}
```

Note that this method is synchronized, so the `_destroyed` member cannot be accessed concurrently.

Here is the `destroy` member function for directories:

```
public void
destroy(Current c) throws PermissionDenied
{
    if (_parent == null)
        throw new PermissionDenied("Cannot destroy root directory"
);

    synchronized(this) {
        if (_destroyed)
            throw new ObjectNotExistException();

        if (_contents.size() != 0)
            throw new PermissionDenied(
                "Cannot destroy non-empty directory");

        c.adapter.remove(id());
        _destroyed = true;
    }

    _parent.removeEntry(_name);
}
```

The code first prevents destruction of the root directory and then checks whether this directory was destroyed previously. It then acquires the lock and checks that the directory is empty. Finally, `destroy` removes the ASM entry for the destroyed directory and removes itself from its parent's `_contents` map. Note that, for the reason we explained in Section 34.7.1, we call `removeEntry` outside the synchronization.

The `createDirectory` implementation acquires the lock before checking whether the directory already contains a node with the given name (or an invalid empty name). If not, it creates a new servant, adds it to the ASM and the `_contents` map, and returns its proxy:

```
public synchronized DirectoryPrx
createDirectory(String name, Current c) throws NameInUse
{
    if (_destroyed)
        throw new ObjectNotExistException();

    if (name.length() == 0 || _contents.containsKey(name))
        throw new NameInUse(name);

    DirectoryI d = new DirectoryI(name, this);
    ObjectPrx node = c.adapter.add(d, d.id());
    _contents.put(name, d);
    return DirectoryPrxHelper.uncheckedCast(node);
}
```

The `createFile` implementation is identical, except that it creates a file instead of a directory:

```
public synchronized FilePrx
createFile(String name, Current c) throws NameInUse
{
    if (_destroyed)
        throw new ObjectNotExistException();

    if (name.length() == 0 || _contents.containsKey(name))
        throw new NameInUse(name);

    FileI f = new FileI(name, this);
    ObjectPrx node = c.adapter.add(f, f.id());
    _contents.put(name, f);
    return FilePrxHelper.uncheckedCast(node);
}
```

Here is the implementation of `list`:

```
public synchronized NodeDesc[]
list(Current c)
{
    if(_destroyed)
        throw new ObjectNotExistException();

    NodeDesc[] ret = new NodeDesc[_contents.size()];
    java.util.Iterator<java.util.Map.Entry<String, NodeI> > pos =
        _contents.entrySet().iterator();
    for(int i = 0; i < _contents.size(); ++i) {
        java.util.Map.Entry<String, NodeI> e = pos.next();
        NodeI p = e.getValue();
```

```
        ret[i] = new NodeDesc();
        ret[i].name = e.getKey();
        ret[i].type = p instanceof FileI
            ? NodeType.FileType : NodeType.DirType;
        ret[i].proxy = NodePrxHelper.uncheckedCast(
            c.adapter.createProxy(p.id()));
    }
    return ret;
}
```

After acquiring the lock, the code iterates over the directory's contents and adds a `NodeDesc` structure for each entry to the returned array.

The `find` operation proceeds along similar lines:

```
public synchronized NodeDesc
find(String name, Current c) throws NoSuchName
{
    if (_destroyed)
        throw new ObjectNotExistException();

    NodeI p = _contents.get(name);
    if (p == null)
        throw new NoSuchName(name);

    NodeDesc d = new NodeDesc();
    d.name = name;
    d.type = p instanceof FileI
        ? NodeType.FileType : NodeType.DirType;
    d.proxy = NodePrxHelper.uncheckedCast(
        c.adapter.createProxy(p.id()));
    return d;
}
```

The `FileI` Class

The `FileI` class is similar to the `DirectoryI` class. The data members store the name, parent, and identity of the file, as well as the `_destroyed` flag and the contents of the file (in the `_lines` member). The constructor initializes these members:

```
package FilesystemI;

import Ice.*;
import Filesystem.*;
import FilesystemI.*;
```

```
public class FileI extends _FileDisp
                   implements NodeI
{
    // ...

    public FileI(String name, DirectoryI parent)
    {
        _name = name;
        _parent = parent;
        _destroyed = false;
        _id = new Identity();
        _id.name = Util.generateUUID();
    }

    private String _name;
    private DirectoryI _parent;
    private boolean _destroyed;
    private Identity _id;
    private String[] _lines;
}
```

The implementation of the remaining member functions of the `FileI` class is trivial, so we present all of them here:

```
public synchronized String
name(Current c)
{
    if (_destroyed)
        throw new ObjectNotExistException();
    return _name;
}

public Identity
id()
{
    return _id;
}

public synchronized String[]
read(Current c)
{
    if (_destroyed)
        throw new ObjectNotExistException();

    return _lines;
}
```

```
public synchronized void
write(String[] text, Current c)
{
    if (_destroyed)
        throw new ObjectNotExistException();

    _lines = (String[])text.clone();
}

public void
destroy(Current c)
{
    synchronized(this) {
        if (_destroyed)
            throw new ObjectNotExistException();

        c.adapter.remove(id());
        _destroyed = true;
    }

    _parent.removeEntry(_name);
}
```

Concurrency Considerations

The preceding implementation is provably deadlock free. All methods hold only one lock at a time, so they cannot deadlock with each other or themselves. While the locks are held, the methods do not call other methods that acquire locks, so any potential deadlock can only arise by concurrent calls to another mutating method, either on the same node or on different nodes. For concurrent calls on the same node, deadlock is impossible because such calls are strictly serialized on the instance; for concurrent calls to `destroy` on different nodes, each node locks itself, releases itself again, and then acquires and releases a lock on its parent (by calling `removeEntry`), also making deadlock impossible.

34.10 Avoiding Server-Side Garbage

The preceding sections covered the implementation of object life cycle, that is, how to correctly provide clients with the means to create and destroy objects. However, throughout the preceding discussion, we have tacitly assumed that clients actually call `destroy` once they no longer need an object. What if this is

actually not the case? For example, a client might intend to call `destroy` on an object it no longer needs but crash before it can actually make the call.

To see why this is a realistic (and serious) scenario, consider an on-line retail application. Typically, such an application provides a shopping cart to the client, into which the client can place items. Naturally, the cart and the items it contains will be modelled as server-side objects. The expectation is that, eventually, the client will either finalize or cancel the purchase, at which point the shopping cart and its contents can be destroyed. However, the client may never do that, for example, because it crashes or simply loses interest.

The preceding scenario applies to many different applications and shows up in various disguises. For example, the objects might be session objects that encapsulate security credentials of a client, or might be iterator objects that allow a client to iterate over a collection of values. The key point is that the interactions between client and server are stateful: the server creates state on behalf of a client, holds that state for the duration of several client–server interactions, and expects the client to inform the server when it can clean up that state. If the client never informs the server, whatever resources are associated with that client's state are leaked; these resources are termed *garbage*.

The garbage might be memory, file descriptors, network connections, disk space, or any number of other things. Unless the server takes explicit action, eventually, the garbage will accumulate to the point where the server fails because it has run out of memory, file descriptors, network connections, or disk space.

In the context of Ice, the garbage are servants and their associated resources. In this section, we examine strategies that a server can use avoid drowning in that garbage.

34.10.1 Approaches to Garbage Collection

The server is presented with something of a dilemma by garbage objects. The difficulty is not in how to remove the garbage objects (after all, the server knows how to destroy each object), but how to identify whether a particular object is garbage or not. The server knows when a client uses an object (because the server receives an invocation for the object), but the server does not know when an object is no longer of interest to a client (because a dead client is indistinguishable from a slow one).

One approach to dealing with garbage is to avoid creating it in the first place: if all interactions between client and server are stateless, the garbage problem does not arise. Unfortunately, for many applications, implementing this approach is

infeasible. The reason is that, in order to turn interactions that are inherently stateful (such as updating a database) into stateless ones, designers are typically forced to keep all the state on the client side, transmit whatever state is required by the server with each remote procedure call, and return the updated state back to the client. In many situations, this simply does not work: for one, the amount of state that needs to be transmitted with each call is often prohibitively large; second, replicating all the state on the client side creates other problems, such as different clients concurrently making conflicting updates to the same data.

The remainder of this section ignores stateless designs. This is not to say that stateless designs are undesirable: where suitable, they can be very effective. However, because many applications simply cannot use them, we focus instead on other ways to deal with garbage.

Mechanisms that identify and reclaim garbage objects are known as garbage collectors. Garbage collectors are well-understood for non-distributed systems. (For example, many programming languages, such as Java and C#, have built-in garbage collectors.)

Non-distributed garbage collectors keep track of all objects that are created, and perform some form of connectivity analysis to determine which objects are still reachable; any objects that are unreachable (that is, objects to which the application code no longer holds any reference) are garbage and are eventually reclaimed.

Unfortunately, for distributed systems, traditional approaches to garbage collection do not work because the cost of performing the connectivity analysis becomes prohibitively large. For example, DCOM provided a distributed garbage collector that turned out to be its Achilles' heel: the collector did not scale to large numbers of objects, particularly across WANs, and several attempts at making it scale failed.

An alternative to distributed garbage collection is to use timeouts to avoid the cost of doing a full connectivity analysis: if an object has not been used for a certain amount of time, the server assumes that it is garbage and reclaims the object. The drawback of this idea is that it is possible for objects to be collected while they are still in use. For example, a customer may have placed a number of items in a shopping cart and gone out to lunch, only to find on return that the shopping cart has disappeared in the mean time.

Yet another alternative is to use the evictor pattern: the server puts a cap on the total number of objects it is willing to create on behalf of clients and, once the cap is reached, destroys the least-recently used object in order to make room for a new one. This puts an upper limit on the resources used by the server and eventually

gets rid of all unwanted objects. But the drawback is the same as with timeouts: just because an object has not been used for a while does not necessarily mean that it truly is garbage.

Neither timeouts nor evictors are true garbage collectors because they can collect objects that are not really garbage, but they do have the advantage that they reap objects even if the client is alive, but forgets to call `destroy` on some of these objects.

34.10.2 Garbage Collection For Ice Applications

Traditional garbage collection fails in the distributed case for a number of reasons:

- Garbage collectors require connectivity analysis, which is prohibitively expensive. Furthermore, for distributed object systems that permit proxies to be externalized as strings, such as Ice, connectivity analysis is impossible because proxies can exist and travel by means that are invisible to the run time. For example, proxies can exist as records in a database and can travel as strings inside e-mail messages.
- Garbage collectors consider all objects in existence but, for the vast majority of applications, only a small subset of all objects actually ever needs collecting. The work spent in examining objects that can never become garbage is wasted.
- Garbage collectors examine connectivity at the granularity of a single object. However, for many distributed applications, objects are used in groups and, if one object in a group is garbage, all objects in the group are garbage. It would be useful to take advantage of this knowledge, but a garbage collector cannot do this because that knowledge is specific to each application.

In the remainder of this section, we examine a simple mechanism that allows you to get rid of garbage objects cheaply and effectively. The approach has the following characteristics:

- Only those objects that potentially can become garbage are considered for collection.
- Granularity of collection is under control of the application: you can have objects collected as groups of arbitrary size, down to a single object.
- Objects are guaranteed not to be collected prematurely.
- Objects are guaranteed to be collected if the client crashes or suffers loss of connectivity.

- The mechanism is simple to implement and has low run-time overhead.

It is equally important to be aware of the limitations of the approach:

- The approach collects objects if a client crashes, but offers no protection against clients that are still running, but have neglected to destroy objects that they no longer need. In other words, the server is protected against client-side hardware failure and catastrophic client crashes, but it is not protected against faulty programming logic of clients.
- The approach is not transparent at the interface level: it requires changes (albeit minor ones) to the interface definitions for an application.
- The approach requires the client to periodically call the server, thus consuming network resources even if the client is otherwise idle.

Despite the limitations, this approach to garbage collection is applicable to a wide variety of applications and meets the most pragmatic need: how to clean up in case something goes badly wrong (rather than how to clean up in case the client misbehaves).

An Extra Level of Indirection

Object factories are typically singleton objects that create objects on behalf of various clients. It is important for our garbage collector to know which client created what objects, so the collector can reap the objects created by a specific client if that client crashes. We can easily deal with this requirement by adding the proverbial level of indirection: instead of making a factory a singleton object, we provide a singleton object that creates factories. Clients first create a factory and then create all other objects they need using that factory:

```
interface Item { /* ...*/ };

interface Cart
{
    Item* create(/* ... */);
    idempotent string getName();
    void destroy();
    idempotent void refresh();
};

interface CartFactory // Singleton
{
    Cart* create(string name);
};
```

Clients obtain a proxy to the `CartFactory` singleton and call `create` to create a `Cart` object. In turn, the `Cart` object provides a `create` operation to place new objects of type `Item` into the cart. Note that the shopping cart name allows a client to distinguish different carts—the `name` parameter is *not* used to identify clients or to provide a unique identifier for carts. The `getName` operation on the `Cart` object returns the name that was used by the client to create it.

Each `Cart` object remembers which items it created. Because each client uses its own shopping cart, the server knows which objects were created by what client. In normal operation, a client first creates a shopping cart, and then uses the cart to create the items in the cart. Once the client has finished its job, it calls `destroy` on the cart. The implementation of `destroy` destroys both the cart and its items to reclaim resources.

To deal with crashed clients, the server needs to know when a cart is no longer in use. This is the purpose of the `refresh` operation: clients are expected to periodically call `refresh` on their cart objects. For example, the server might decide that, if a client's cart has not been refreshed for more than ten minutes, the cart is no longer in use and reclaim it. As long as the client calls `refresh` at least once every ten minutes, the cart (and the items it contains) remain alive; if more than ten minutes elapse, the server simply calls `destroy` on the cart. Of course, there is no need to hard-wire the timeout value—you can make it part of the application's configuration. However, to keep the implementation simple, it is useful to have the same timeout value for all carts, or to at least restrict the timeouts for different carts to a small number of fixed choices—this considerably simplifies the implementation in both client and server.

Server-Side Implementation

The implementation of the interfaces on the server side almost suggests itself:

- Whenever `refresh` is called on a cart, the cart records the time at which the call was made.
- The server runs a reaper thread that wakes up once every ten minutes. The reaper thread examines the timestamp of all carts and, if it finds a cart last time-stamped more than ten minutes ago, it calls `destroy` on that cart.
- Each cart remembers the items it contains and destroys them as part of its `destroy` implementation.

Here then is the reaper thread in outline. (Note that we have simplified the code to show the essentials. For example, we have omitted the code that is needed to make

the reaper thread terminate cleanly when the server shuts down. See the code in `demo/Ice/session` for more detail.)

```
class ReapThread : public IceUtil::Thread,
                   public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    ReapThread();
    virtual void run();
    void add(const CartPrx&, const CartIPtr&);

private:
    const IceUtil::Time _timeout;
    struct CartProxyPair
    {
        CartProxyPair(const CartPrx& p, const CartIPtr& c) :
              proxy(p), cart(c) { }
        const CartPrx proxy;
        const CartIPtr cart;
    };
    std::list<CartProxyPair> _carts;
};

typedef IceUtil::Handle<ReapThread> ReapThreadPtr;
```

Note that the reaper thread maintains a list of pairs. Each pair stores the proxy of a cart and its servant pointer. We need both the proxy and the pointer because we need to invoke methods on both the Slice interface and the implementation interface of the cart. Whenever a client creates a new cart, the server calls the `add` method on the reaper thread, passing it the new cart:

```
void ReapThread::add(const CartPrx& proxy,
                     const CartIPtr& cart)
{
    Lock sync(*this);
    _carts.push_back(CartProxyPair(proxy, cart));
}
```

The `run` method of the reaper thread is a loop that sleeps for ten minutes and calls `destroy` on any session that has not been refreshed within the preceding ten minutes:

```
void ReapThread::run()
{
    Lock sync(*this);
    while (true) {
```

```
        timedWait(_timeout);
        list<CartProxyPair>::iterator p = _carts.begin();
        while (p != _carts.end()) {
            try {
                //
                // Cart destruction may take some time.
                // Therefore the current time is computed
                // for each iteration.
                //
                if ((IceUtil::Time::now() - p->cart->timestamp())
                       > _timeout) {
                    p->proxy->destroy();
                    p = _carts.erase(p);
                } else {
                    ++p;
                }
            } catch (const Ice::ObjectNotExistException&) {
                p = _carts.erase(p);
            }
        }
    }
}
```

Note that the reaper thread catches `ObjectNotExistException` from the call to `destroy`, and removes the cart from its list in that case. This is necessary because it is possible for a client to call `destroy` explicitly, so a cart may be destroyed already by the time the reaper thread examines it.

The `CartFactory` implementation is trivial:

```
class CartFactoryI : CartFactory
{
public:
    CartFactoryI(const ReapThreadPtr&);
    virtual CartPrx create(const std::string&,
                           const Ice::Current&);

private:
    ReapThreadPtr _reaper;
};
```

The constructor is passed the instantiated reaper thread and remembers that thread in the `_reaper` member.

The `create` method adds each new cart to the reaper thread's list of carts:

```
CartPrx CartFactoryI::create(const string& name,
                             const Ice::Current& c)
{
    CartIPtr cart = new CartI(name);
    CartPrx proxy = CartPrx::uncheckedCast(
                        c.adapter->addWithUUID(cart));
                        _reaper->add(proxy, cart);
    return proxy;
}
```

Note that each cart internally has a unique ID that is unrelated to its name—the name exists purely as a convenience for the application.

The server's `main` function starts the reaper thread and instantiates the cart factory:

```
ReapThreadPtr reaper = new ReapThread();
CartFactory factory = new CartFactoryI(reaper);
reaper->start();
adapter->add(factory,
             Ice::stringToIdentity(CartFactory));
adapter->activate();
```

This completes the implementation on the server side. Note that there is very little code here, and that much of this code is essentially the same for each application. For example, we could easily turn the `ReapThread` class into a template class to permit the same code to be used for something other than shopping carts.

Client-Side Implementation

On the client side, the application code does what it would do with an ordinary factory, except for the extra level of indirection: the client first creates a cart, and then uses the cart as its factory.

As long as the client-side calls `refresh` at least once every ten minutes, the cart remains alive and, with it, all items the client created in that cart. Once the client misses a `refresh` call, the reaper thread in the server cleans up the cart and its items.

To keep the cart alive, you could sprinkle your application code with calls to `refresh` in the hope that at least one of these calls is made at least every ten minutes. However, that is not only error-prone, but also fails if the client blocks for some time. A much better approach is to run a thread in the client that automatically calls `refresh`. That way, the calls are guaranteed to happen even if the client's main thread blocks for some time, and the application code does not get polluted with `refresh` calls. Again, we show a simplified version of the refresh

thread here that does not deal with issues such as clean shutdown and a few other irrelevant details:

```
class CartRefreshThread : public IceUtil::Thread,
                          public IceUtil::Monitor<IceUtil::Mutex>
{
public:
    CartRefreshThread(const IceUtil::Time& timeout,
                      const CartPrx& cart) :
        _cart(cart),
        _timeout(timeout) {}

    virtual void run() {
        Lock sync(*this);
        while(true) {
            timedWait(_timeout);
            try {
                _cart->refresh();
            } catch(const Ice::Exception& ex) {
                return;
            }
        }
    }

private:
    const CartPrx _cart;
    const IceUtil::Time _timeout;
};

typedef IceUtil::Handle<CartRefreshThread> CartRefreshThreadPtr;
```

The client's `main` function instantiates the reaper thread after creating a cart. We assume that the client has a proxy to the cart factory in the `factory` variable:

```
CartPrx cart = factory->create(name);
CartRefreshThreadPtr refresh
    = new CartRefreshThread(IceUtil::Time::seconds(480), cart);
refresh->start();
```

Note that, to be on the safe side and also allow for some network delays, the client calls refresh every eight minutes; this is to ensure that at least one call to `refresh` arrives at the server within each ten-minute interval.

34.11 Summary

It is important to be clear about the meaning of object existence which, as far as the Ice run time is concerned, is defined only within the context of a particular operation invocation. It is also important to be clear about the distinction between proxies, servants, and Ice objects, which have independent life cycles.

While object creation is usually simple, object destruction requires a great deal of attention, particularly for threaded servers: the interactions among life cycle operations and ordinary operations can be complex and require careful consideration with respect to locks.

Frequently, implementing object destruction with reaping instead of callbacks leads to simpler and more maintainable code and eliminates hidden dependencies that can cause deadlock.

Ultimately, the semantics of object existence are supplied by the application code. Depending on how an application is structured, object identities may need to be globally unique; alternatively, if the life times of proxies that use the same identity for different objects cannot overlap, object identities can be safely re-used. It is important to be aware of the respective semantics in order to create correct applications.

If interactions between clients and server are stateful, the server must take care to reclaim state to protect itself against resource exhaustion if clients fail to destroy that state.

Chapter 35
Dynamic Ice

35.1 Chapter Overview

In this chapter we present a collection of Ice interfaces for a dynamic invocation and dispatch model. Section 35.2 discusses the streaming interface for serializing Slice types, an essential tool when implementing dynamic invocation and dispatch. The synchronous language mappings for the dynamic model are described in Section 35.3, while Section 35.4 covers the asynchronous language mappings.

35.2 Streaming Interface

Ice provides a convenient interface for streaming Slice types to and from a sequence of bytes. You can use this interface in many situations, such as when serializing types for persistent storage, and when using Ice's dynamic invocation and dispatch interfaces (see Section 35.3).

The streaming interface is not defined in Slice, but rather is a collection of native classes provided by each language mapping[1]. A default implementation of

1. The streaming interface is currently supported in C++, Java, and .NET.

the interface uses the Ice encoding as specified in Section 37.2, but other implementations are possible.

There are two primary abstract classes in the streaming interface: `InputStream` and `OutputStream`. As you might guess, `InputStream` is used to extract Slice types from a sequence of bytes, while `OutputStream` is used to convert Slice types into a sequence of bytes. The classes provide the functions necessary to manipulate all of the core Slice types:

- Primitives (`bool`, `int`, `string`, etc.)
- Sequences of primitives
- Proxies
- Objects

The classes also provide functions that handle various details of the Ice encoding. Using these functions, you can manually insert and extract constructed types, such as dictionaries and structures, but doing so is tedious and error-prone. To make insertion and extraction of constructed types easier, the Slice compilers can optionally generate helper functions that manage the low-level details for you.

The remainder of this section describes the streaming interface for each supported language mapping. To properly use the streaming interface, you should be familiar with the Ice encoding (see Section 37.2). An example that demonstrates the use of the streaming interface is located in `demo/Ice/invoke` in the Ice distribution.

35.2.1 C++ Stream Interface

We discuss the stream classes first, followed by the helper functions, and finish with an advanced use case of the streaming interface.

InputStream

An `InputStream` is created using the following function:

```
namespace Ice {
    InputStreamPtr createInputStream(
        const Ice::CommunicatorPtr& communicator,
        const std::vector<Ice::Byte>& data);
}
```

The `InputStream` class is shown below.

```
namespace Ice {
    class InputStream : ... {
    public:
        virtual CommunicatorPtr communicator() const = 0;

        virtual void sliceObjects(bool slice) = 0;

        virtual void read(bool& v) = 0;
        virtual void read(Byte& v) = 0;
        virtual void read(Short& v) = 0;
        virtual void read(Int& v) = 0;
        virtual void read(Long& v) = 0;
        virtual void read(Float& v) = 0;
        virtual void read(Double& v) = 0;
        virtual void read(std::string& s,
                          bool convert = true) = 0;
        virtual void read(std::wstring& s) = 0;

        template<typename T> inline void read(T& v) {
            StreamReader< StreamTrait<T>::type>::read(this, v);
        }

        virtual void read(std::vector<std::string>& v,
                          bool convert) = 0;

        virtual void read(
                        std::pair<const bool*, const bool*>&,
                        IceUtil::ScopedArray<bool>&) = 0;

        virtual void read(
                        std::pair<const Byte*, const Byte*>&) = 0;

        virtual void read(std::pair<const Short*, const Short*>&,
                        IceUtil::ScopedArray<Short>&) = 0;

        virtual void read(
                        std::pair<const Int*, const Int*>&,
                        IceUtil::ScopedArray<Int>&) = 0;

        virtual void read(
                        std::pair<const Long*, const Long*>&,
                        IceUtil::ScopedArray<Long>&) = 0;

        virtual void read(
                        std::pair<const Float*, const Float*>&,
                        IceUtil::ScopedArray<Float>&) = 0;
```

```
        virtual void read(
                        std::pair<const Double*, const Double*>&,
                        IceUtil::ScopedArray<Double>&) = 0;

        virtual Int readSize() = 0;
        virtual Int readAndCheckSeqSize(int minWireSize) = 0;

        virtual ObjectPrx readProxy() = 0;

        template<typename T> inline void
        read(IceInternal::ProxyHandle<T>& v) {
            // ...
        }

        virtual void readObject(
                        const ReadObjectCallbackPtr& cb) = 0;

        template<typename T> inline void
        read(IceInternal::Handle<T>& v) {
            // ...
        }

        virtual std::string readTypeId() = 0;

        virtual void throwException() = 0;

        virtual void startSlice() = 0;
        virtual void endSlice() = 0;
        virtual void skipSlice() = 0;

        virtual void startEncapsulation() = 0;
        virtual void endEncapsulation() = 0;
        virtual void skipEncapsulation() = 0;

        virtual void readPendingObjects() = 0;

        virtual void rewind() = 0;
    };
    typedef ... InputStreamPtr;
}
```

Extracting Built-In Types

Member functions are provided to extract any of the built-in types. For example, you can extract a double value followed by a string from a stream as follows:

```
vector<Ice::Byte> data = ...;
in = Ice::createInputStream(communicator, data);
double d;
in->read(d);
string s;
in->read(s);
```

Extracting Sequences of Built-In Type

For other types, the following template member function performs the extraction:

```
template<typename T> inline void
read(T& v) {
    StreamReader<StreamTrait<T>::type>::read(this, v);
}
```

For example, you can extract a sequence of integers as follows:

```
vector<Ice::Byte> data = ...;
in = Ice::createInputStream(communicator, data);
// ...
IntSeq s; // Slice: sequence<int> IntSeq;
in->read(s);
```

The Ice run time provides an implementation of the `StreamReader` template whose `read` method reads a sequence of any of the built-in types. Note that, when reading a sequence, this reads both the sequence size that precedes the sequence elements as well as the sequence elements that follow the size.

If you are using a custom container for your sequence of built-in type, you must provide a specialization of the `StreamTrait` template in order to extract your sequence. For example, the following definition allows you to use the `QVector` container from the Qt library:

```
//
// StreamTrait specialization for QVector
//
template<typename T>
struct StreamTrait< QVector<T> >
{
    static const StreamTraitType type = StreamTraitTypeSequence;
    static const int minWireSize = 1;
};
```

Extracting Sequences of Built-In Type Using Zero-Copy

Note that `InputStream` provides a number of overloads that accept a pair of pointers. For example, you can extract a sequence of bytes as follows:

```
vector<Ice::Byte> data = ...;
in = Ice::createInputStream(communicator, data);
std::pair<const Ice::Byte*, const Ice::Byte*> p;
in->read(p);
```

The same extraction technique works for the other built-in integral and floating-point types, such `int` and `double`.

If the extraction is for a byte sequence, the returned pointers always point at memory in the stream's internal marshaling buffer.

For the other built-in types, the pointers refer to the internal marshaling buffer only if the Ice encoding is compatible with the machine and compiler representation of the type, otherwise the pointers refer to a temporary array allocated to hold the unmarshaled data. The overloads for zero-copy extraction accept an additional parameter of type `IceUtil::ScopedArray` (see Appendix F) that holds this temporary array when necessary.

Here is an example to illustrate how to extract a sequence of integers, regardless of whether the machine's encoding of integers matches the on-the-wire representation or not:

```
#include <IceUtil/ScopedArray.h>
...
in = Ice::createInputStream(communicator, data);
std::pair<const Ice::Int*, const Ice::Int*> p;
IceUtil::ScopedArray<Ice::Int> a;
in->read(p, a);

for(const Ice::Int* i = p.first; i != p.second; ++i) {
    cout << *i << endl;
}
```

If the on-the-wire encoding matches that of the machine, and therefore zero-copy is possible, the returned pair of pointers points into the run time's internal marshaling buffer. Otherwise, the run time allocates an array, unmarshals the data into the array, and sets the pair of pointers to point into that array. Use of the `ScopedArray` helper template ensures that the array is deallocated once you let the `ScopedArray` go out of scope, so there is no need to call `delete[]`. (`ScopedArray` is conceptually the same as the `Ptr` smart pointer types for classes. See Appendix F for details.)

Extracting Structures

Without the **`--stream`** option to **`slice2cpp`**, you must extract structures member by member according to the rules in Chapter 37. Otherwise, with

`--stream`, **`slice2cpp`** generates code that allows you to extract the structure directly. For example, here is how you can extract a Slice structure called `MyStruct` from a stream:

```
in = Ice::createInputStream(communicator, data);
MyStruct myStruct;
in->read(myStruct);
```

Extracting Dictionaries

Without the **`--stream`** option to **`slice2cpp`**, you can extract any dictionary whose key and value types are built-in types; for any other dictionary, you must extract it as a size followed by its entries according to the rules in Chapter 37. If you are using a custom container for your dictionary of built-in types, you must provide a specialization of the `StreamTrait` template in order to extract your dictionary. For example, the following definition allows you to use the `QMap` container from the Qt library:

```
//
// StreamTrait specialization for QMap
//
template<typename K, typename V>
struct StreamTrait< QMap<K, V> >
{
    static const StreamTraitType type = StreamTraitTypeDictionary;
    static const int minWireSize = 1;
};
```

With the **`--stream`** option, **`slice2cpp`** generates code that allows you to extract any dictionary directly, for example:

```
in = Ice::createInputStream(communicator, data);
MyDict myDict; // Slice: dictionary<string, SomeType> MyDict;
in->read(myDict);
```

Extracting Sequences of User-Defined Type

Without the **`--stream`** option to **`slice2cpp`**, you must extract sequences of user-defined type as a size followed by the element type according to the rules in Chapter 37. Otherwise, with **`--stream`**, **`slice2cpp`** generates code that allows you to extract a sequence directly, for example:

```
in = Ice::createInputStream(communicator, data);
MyEnumS myEnumS; // Slice: sequence<MyEnum> myEnumS;
in->read(myEnumS);
```

Other Member Functions

The remaining member functions of `InputStream` have the following semantics:

- `void sliceObjects(bool slice)`

  Determines the behavior of the stream when extracting Ice objects. An Ice object is "sliced" when a factory cannot be found for a Slice type id (see Section 37.2.11 for more information), resulting in the creation of an object of a less-derived type. Slicing is typically disabled when the application expects all object factories to be present, in which case the exception `NoObjectFactoryException` is raised. The default behavior is to allow slicing.

- `void read(std::string& v, bool convert = true)`
  `void read(std::vector<std::string>& v,`
  `bool convert = true)`

  The optional boolean argument determines whether the strings unmarshaled by these methods are processed by the string converter, if one is installed. The default behavior is to convert the strings. See Section 32.24 for more information on string converters.

- `Ice::Int readSize()`

  The Ice encoding has a compact representation to indicate size (see Section 37.2.1). This function extracts a size and returns it as an integer.

- `Ice::Int readAndCheckSeqSize(int minWireSize)`

  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element (see Chapter 37). If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.

- `Ice::ObjectPrx readProxy()`

  This function returns an instance of the base proxy type, `ObjectPrx`.

- `template<typename T> inline void`
  `read(IceInternal::ProxyHandle<T>& v)`

  This template function behaves like `readProxy` but avoids the need to down-cast the return value. You can pass a proxy of any type as the parameter `v`.

- `void readObject(const Ice::ReadObjectCallbackPtr &)`

  The Ice encoding for class instances requires extraction to occur in stages (see Section 37.2.11). The `readObject` function accepts a callback object of type `ReadObjectCallback`, whose definition is shown below:

  ```
  namespace Ice {
      class ReadObjectCallback : ... {
      public:
          virtual void invoke(const Ice::ObjectPtr&) = 0;
      };
      typedef ... ReadObjectCallbackPtr;
  }
  ```

  When the object instance is available, the callback object's `invoke` member function is called. The application must call `readPendingObjects` to ensure that all instances are properly extracted.

  If you are not interested in receiving a callback when the object is extracted, it is easier to use the `read(IceInternal::Handle<T>&)` template function instead (see below).

- `template<typename T> inline void read(IceInternal::Handle<T>& v)`

  This template function behaves like `readObject` but avoids the need to supply a callback. You can pass a smart pointer of any type as the parameter `v`. Note that, if you want to intercept object extraction, you must use `readObject` instead.

- `std::string readTypeId()`

  A table of Slice type ids is used to save space when encoding Ice objects (see Section 37.2.11). This function returns the type id at the stream's current position.

- `void throwException()`

  This function extracts a user exception from the stream and throws it. If the stored exception is of an unknown type, the function attempts to extract and throw a less-derived exception. If that also fails, an `UnmarshalOutOf-BoundsException` is thrown.

- `void startSlice()`
  `void endSlice()`
  `void skipSlice()`

  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of an Ice object or user exception. See Section 37.2.11 for more information.

- `void startEncapsulation()`
  `void endEncapsulation()`
  `void skipEncapsulation()`

  Start, end, and skip an encapsulation, respectively. See Section 37.2.2 for more information.

- `void readPendingObjects()`

  An application must call this function after all other data has been extracted, but only if Ice objects were encoded. This function extracts the state of Ice objects and invokes their corresponding callback objects (see `readObject`).

- `void rewind()`

  Resets the position of the stream to the beginning.

OutputStream

An `OutputStream` is created using the following function:

```
namespace Ice {
    OutputStreamPtr createOutputStream(
        const Ice::CommunicatorPtr& communicator);
}
```

The `OutputStream` class is shown below.

```
namespace Ice {
    class OutputStream : ... {
    public:
        virtual Ice::CommunicatorPtr communicator() const = 0;

        virtual void write(bool v) = 0;
        virtual void write(Byte v) = 0;
        virtual void write(Short v) = 0;
        virtual void write(Int v) = 0;
        virtual void write(Long v) = 0;
        virtual void write(Float v) = 0;
        virtual void write(Double v) = 0;
        virtual void write(const std::string& v,
```

```
                    bool convert = true) = 0;
virtual void write(const char* v,
                    bool convert = true) = 0;
virtual void write(const std::wstring& v) = 0;

virtual void write(const bool* begin,
                    const bool* end) = 0;
virtual void write(const Byte* begin,
                    const Byte* end) = 0;
virtual void write(const Short* begin,
                    const Short* end) = 0;
virtual void write(const Int* begin,
                    const Int* end) = 0;
virtual void write(const Long* begin,
                    const Long* end) = 0;
virtual void write(const Float* begin,
                    const Float* end) = 0;
virtual void write(const Double* begin,
                    const Double* end) = 0;

virtual void write(const std::vector<std::string>& v,
                    bool convert) = 0;

template<typename T> inline void
write(const T& v) {
    StreamWriter<StreamTrait<T>::type>::write(this, v);
}

virtual void writeSize(Ice::Int sz) = 0;

virtual void writeProxy(const Ice::ObjectPrx& v) = 0;

template<typename T> inline void
write(const IceInternal::ProxyHandle<T>& v) {
    // ...
}

virtual void writeObject(const Ice::ObjectPtr& v) = 0;

template<typename T> inline void
write(const IceInternal::Handle<T>& v) {
    // ...
}

virtual void writeTypeId(const std::string& id) = 0;
```

```
        virtual void writeException(
                        const Ice::UserException& e) = 0;

        virtual void startSlice() = 0;
        virtual void endSlice() = 0;

        virtual void startEncapsulation() = 0;
        virtual void endEncapsulation() = 0;

        virtual void writePendingObjects() = 0;

        virtual void finished(std::vector<Ice::Byte>& v) = 0;

        virtual void reset(bool) = 0;
    };
}
```

Inserting Built-In Types

Member functions are provided to insert any of the built-in types. For example, you can insert a double value followed by a string into a stream as follows:

```
out = Ice::createOutputStream(communicator);
Ice::Double d = 3.14;
out->write(d);
string s = "Hello";
out->write(s);
```

Inserting Sequences of Built-In Type

For other types, the following template member function performs the insertion:

```
template<typename T> inline void
write(const T& v) {
    StreamWriter<StreamTrait<T>::type>::write(this, v);
}
```

For example, you can insert a sequence of integers as follows:

```
out = Ice::createOutputStream(communicator);
IntSeq s = ...;
out->write(s);
```

The Ice run time provides an implementation of the `StreamWriter` template whose `write` method writes a sequence of any of the built-in types. Note that, when writing a sequence, this writes both the sequence size that precedes the sequence elements and the sequence elements that follow the size.

If you are using a custom container for your sequence of built-in type, you must provide a specialization of the `StreamTrait` template in order to insert your sequence. For example, the following definition allows you to use the `QVector` container from the Qt library:

```
//
// StreamTrait specialization for QVector
//
template<typename T>
struct StreamTrait< QVector<T> >
{
    static const StreamTraitType type = StreamTraitTypeSequence;
    static const int minWireSize = 1;
};
```

Inserting Sequences of Built-In Type Using Zero-Copy

Note that `OutputStream` provides a number of overloads that accept a pair of pointers. For example, you can insert a sequence of bytes as follows:

```
out = Ice::createOutputStream(communicator);
vector<Ice::Byte> data = ...;
out->write(&v[0], &v[v.size()]);
```

The same insertion technique works for the other built-in integral and floating-point types, such `int` and `double`. Insertion in this way can avoid an additional data copy during marshaling if the internal representation of the data in memory is the same as the on-the-wire representation. (Note that the two pointers must point at a contiguous block of memory.)

Inserting Structures

Without the **`--stream`** option to **`slice2cpp`**, you must insert structures member by member according to the rules in Chapter 37. Otherwise, with **`--stream`**, **`slice2cpp`** generates code that allows you to insert the structure directly. For example, here is how you can insert a Slice structure called `MyStruct` into a stream:

```
out = Ice::createOutputStream(communicator);
MyStruct myStruct;
// Initialize myStruct...
out->write(myStruct);
```

Inserting Dictionaries

Without the **--stream** option to **slice2cpp**, you can insert any dictionary whose key and value types are built-in types; for any other dictionary, you must insert it as a size followed by its entries according to the rules in Chapter 37. If you are using a custom container for your dictionary of built-in types, you must provide a specialization of the `StreamTrait` template in order to insert your dictionary. For example, the following definition allows you to use the `QMap` container from the Qt library:

```
//
// StreamTrait specialization for QMap
//
template<typename K, typename V>
struct StreamTrait< QMap<K, V> >
{
    static const StreamTraitType type = StreamTraitTypeDictionary;
    static const int minWireSize = 1;
};
```

With the **--stream** option, **slice2cpp** generates code that allows you to insert any dictionary directly, for example:

```
out = Ice::createOutputStream(communicator);
MyDict myDict; // Slice: dictionary<int, SomeType> MyDict;
// Initialize myDict...
out->write(myDict);
```

Inserting Sequences of User-Defined Type

Without the **--stream** option to **slice2cpp**, you must insert sequences of user-defined type as a size followed by the element type according to the rules in Chapter 37. Otherwise, with **--stream**, **slice2cpp** generates code that allows you to insert a sequence directly, for example:

```
out = Ice::createOutputStream(communicator);
MyEnumS myEnumS; // Slice: sequence<MyEnum> myEnumS;
// Initialize myEnumS...
out->write(myEnumS);
```

Other Member Functions

The remaining member functions of `OutputStream` have the following semantics:

- `void write(const std::string& v,`
  `bool convert = true)`

```
void write(const char* v, bool convert = true)
void write(const std::vector<std::string>&,
           bool convert = true)
```

The optional boolean argument determines whether the strings marshaled by these methods are processed by the string converter, if one is installed. The default behavior is to convert the strings. See Section 32.24 for more information on string converters.

- `void writeSize(Ice::Int sz)`

  The Ice encoding has a compact representation to indicate size (see Section 37.2.1). This function converts the given non-negative integer into the proper encoded representation.

- `void writeProxy(const Ice::ObjectPtr & v)`

  Inserts a proxy.

- `template<typename T> inline void write(const IceInternal::ProxyHandle<T>& v)`

  This template function behaves like `writeProxy`. You can pass a proxy of any type as the parameter `v`.

- `void writeObject(const Ice::ObjectPtr & v)`

  Inserts an Ice object. The Ice encoding for class instances (see Section 37.2.11) may cause the insertion of this object to be delayed, in which case the stream retains a reference to the given object and the stream does not insert its state it until `writePendingObjects` is invoked on the stream.

- `template<typename T> inline void write(const IceInternal::Handle<T>& v)`

  This template function behaves like `writeObject`. You can pass a smart pointer of any type as the parameter `v`.

- `void writeTypeId(const std::string & id)`

  A table of Slice type ids is used to save space when encoding Ice objects (see Section 37.2.11). This function adds the given type id to the table and encodes the type id. `writeTypeId` may only be invoked in the context of a call to `writePendingObjects` (see below).

- `void writeException(const Ice::UserException & ex)`

  Inserts a user exception. You can also use the template member function `write(const T&)` to insert a user exception.

- `void startSlice()`
  `void endSlice()`

  Starts and ends a slice of object or exception member data (see Section 37.2.11).
- `void startEncapsulation()`
  `void endEncapsulation()`

  Starts and ends an encapsulation, respectively (see Section 37.2.2).
- `void writePendingObjects()`

  Encodes the state of Ice objects whose insertion was delayed during `writeObject`. This member function must only be called once.
- `void finished(std::vector< Ice::Byte > & data)`

  Indicates that marshaling is complete. The given byte sequence is filled with the encoded data. This member function must only be called once.
- `void reset(bool clearBuffer)`

  Resets the writing position of the stream to the beginning. If `clearBuffer` is true, the stream releases the memory it has allocated to hold the encoded data.

Intercepting Object Insertion and Extraction

In some situations it may be necessary to intercept the insertion and extraction of Ice objects. For example, the Ice extension for PHP (see Chapter 28) is implemented using Ice for C++ but represents Ice objects as native PHP objects. The PHP extension accomplishes this by manually encoding and decoding Ice objects as directed by Section 37.2. However, the extension obviously cannot pass a native PHP object to the C++ stream function `writeObject`. To bridge this gap between object systems, Ice supplies the classes `ObjectReader` and `ObjectWriter`:

```
namespace Ice {
    class ObjectReader : public Ice::Object {
    public:
        virtual void read(const InputStreamPtr&, bool) = 0;
        // ...
    };
    typedef ... ObjectReaderPtr;

    class ObjectWriter : public Ice::Object {
    public:
        virtual void write(const OutputStreamPtr&) const = 0;
```

```
        // ...
    };
    typedef ... ObjectWriterPtr;
}
```

A foreign Ice object is inserted into a stream using the following technique:

1. A C++ "wrapper" class is derived from `ObjectWriter`. This class wraps the foreign object and implements the `write` member function.
2. An instance of the wrapper class is passed to `writeObject`. (This is possible because `ObjectWriter` derives from `Ice::Object`.) Eventually, the `write` member function is invoked on the wrapper instance.
3. The implementation of `write` encodes the object's state as directed by Section 37.2.11.

It is the application's responsibility to ensure that there is a one-to-one mapping between foreign Ice objects and wrapper objects. This is necessary in order to ensure the proper encoding of object graphs.

Extracting the state of a foreign Ice object is more complicated than insertion:

1. A C++ "wrapper" class is derived from `ObjectReader`. An instance of this class represents a foreign Ice object.
2. An object factory is installed that returns instances of the wrapper class. Note that a single object factory can be used for all Slice types if it is registered with an empty Slice type id (see Section 6.14.5).
3. A C++ callback class is derived from `ReadObjectCallback`. The implementation of `invoke` expects its argument to be either nil or an instance of the wrapper class as returned by the object factory.
4. An instance of the callback class is passed to `readObject`.
5. When the stream is ready to extract the state of an object, it invokes `read` on the wrapper class. The implementation of `read` decodes the object's state as directed by Section 37.2.11. The boolean argument to read indicates whether the function should invoke `readTypeId` on the stream; it is possible that the type id of the current slice has already been read, in which case this argument is `false`.
6. The callback object passed to `readObject` is invoked, passing the instance of the wrapper object. All other callback objects representing the same instance in the stream (in case of object graphs) are invoked with the same wrapper object.

Intercepting User Exception Insertion

Similar to the discussion of Ice objects in the previous section, a Dynamic Ice application may represent user exceptions in a native format that is not directly compatible with the Ice API. If the application needs to raise such a user exception to the Ice run time, the exception must be wrapped in a subclass of `Ice::UserException`. The Dynamic Ice API provides a class to simplify this process:

```
namespace Ice {
    class UserExceptionWriter : public UserException {
    public:
        UserExceptionWriter(const CommunicatorPtr&);

        virtual void write(const OutputStreamPtr&) const = 0;
        virtual bool usesClasses() const = 0;

        virtual std::string ice_name() const = 0;
        virtual Ice::Exception* ice_clone() const = 0;
        virtual void ice_throw() const = 0;

        // ...
    };
    typedef ... UserExceptionWriterPtr;
}
```

A subclass of `UserExceptionWriter` is responsible for supplying a communicator to the constructor, and for implementing the following methods:

- `void write(const OutputStreamPtr&) const`

  This method is invoked when the Ice run time is ready to marshal the exception. The subclass must marshal the exception using the encoding rules specified in Section 37.2.10.

- `bool usesClasses() const`

  Return true if the exception, or any base exception, contains a data member for an object by value.

- `std::string ice_name() const`

  Return the Slice name of the exception.

- `Ice::Exception* ice_clone() const`

  Return a copy of the exception.

- `void ice_throw() const`

  Raise the exception by calling `throw *this`.

35.2.2 Java Stream Interface

We discuss the stream classes first, followed by the helper functions, and finish with an advanced use case of the streaming interface.

InputStream

An `InputStream` is created using the following function:

```
package Ice;

public class Util {
    public static InputStream
    createInputStream(Communicator communicator, byte[] data);
}
```

The `InputStream` interface is shown below.

```
package Ice;

public interface InputStream {
    Communicator communicator();

    void sliceObjects(boolean slice);

    boolean readBool();
    boolean[] readBoolSeq();

    byte readByte();
    byte[] readByteSeq();

    short readShort();
    short[] readShortSeq();

    int readInt();
    int[] readIntSeq();

    long readLong();
    long[] readLongSeq();

    float readFloat();
    float[] readFloatSeq();

    double readDouble();
    double[] readDoubleSeq();

    String readString();
```

```
    String[] readStringSeq();

    int readSize();
    int readAndCheckSeqSize(int minSizeWireSize);

    ObjectPrx readProxy();

    void readObject(ReadObjectCallback cb);

    String readTypeId();

    void throwException() throws UserException;

    void startSlice();
    void endSlice();
    void skipSlice();

    void startEncapsulation();
    void endEncapsulation();
    void skipEncapsulation();

    void readPendingObjects();

    java.io.Serializable readSerializable();

    void rewind()_;

    void destroy();
}
```

Member functions are provided for extracting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void sliceObjects(boolean slice)`

  Determines the behavior of the stream when extracting Ice objects. An Ice object is "sliced" when a factory cannot be found for a Slice type id (see Section 37.2.11 for more information), resulting in the creation of an object of a less-derived type. Slicing is typically disabled when the application expects all object factories to be present, in which case the exception `NoObjectFactoryException` is raised. The default behavior is to allow slicing.

- `int readSize()`

  The Ice encoding has a compact representation to indicate size (see Section 37.2.1). This function extracts a size and returns it as an integer.

- `int readAndCheckSeqSize(int minWireSize)`

  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element (see Chapter 37). If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.

- `Ice.ObjectPrx readProxy()`

  This function returns an instance of the base proxy type, `ObjectPrx`. The Slice compiler optionally generates helper functions to extract proxies of user-defined types (see page 1224).

- `void readObject(ReadObjectCallback cb)`

  The Ice encoding for class instances requires extraction to occur in stages (see Section 37.2.11). The `readObject` function accepts a callback object of type `ReadObjectCallback`, whose definition is shown below:

  ```
  package Ice;

  public interface ReadObjectCallback {
      void invoke(Ice.Object obj);
  }
  ```

  When the object instance is available, the callback object's `invoke` member function is called. The application must call `readPendingObjects` to ensure that all instances are properly extracted.

  Note that applications rarely need to invoke this member function directly; the helper functions generated by the Slice compiler are easier to use (see page 1234).

- `String readTypeId()`

  A table of Slice type ids is used to save space when encoding Ice objects (see Section 37.2.11). This function returns the type id at the stream's current position.

- `void throwException() throws UserException`

  This function extracts a user exception from the stream and throws it. If the stored exception is of an unknown type, the function attempts to extract and throw a less-derived exception. If that also fails, an `UnmarshalOutOfBoundsException` is thrown.

- `void startSlice()`
  `void endSlice()`
  `void skipSlice()`

  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of an Ice object or user exception. See Section 37.2.11 for more information.

- `void startEncapsulation()`
  `void endEncapsulation()`
  `void skipEncapsulation()`

  Start, end, and skip an encapsulation, respectively. See Section 37.2.2 for more information.

- `void readPendingObjects()`

  An application must call this function after all other data has been extracted, but only if Ice objects were encoded. This function extracts the state of Ice objects and invokes their corresponding callback objects (see `readObject`).

- `java.io.Serializable readSerializable()`

  Reads a serializable Java object from the stream (see Section 10.15).

- `void destroy()`

  Applications must call this function in order to reclaim resources.

Here is a simple example that demonstrates how to extract a boolean and a sequence of strings from a stream:

```
byte[] data = ...
Ice.InputStream in =
    Ice.Util.createInputStream(communicator, data);
try {
    boolean b = in.readBool();
    String[] seq = in.readStringSeq();
} finally {
    in.destroy();
}
```

OutputStream

An `OutputStream` is created using the following function:

```
package Ice;

public class Util {
    public static OutputStream createOutputStream(
        Communicator communicator);
}
```

The `OutputStream` class is shown below.

```
package Ice;

public interface OutputStream {
    Communicator communicator();

    void writeBool(boolean v);
    void writeBoolSeq(boolean[] v);

    void writeByte(byte v);
    void writeByteSeq(byte[] v);

    void writeShort(short v);
    void writeShortSeq(short[] v);

    void writeInt(int v);
    void writeIntSeq(int[] v);

    void writeLong(long v);
    void writeLongSeq(long[] v);

    void writeFloat(float v);
    void writeFloatSeq(float[] v);

    void writeDouble(double v);
    void writeDoubleSeq(double[] v);

    void writeString(String v);
    void writeStringSeq(String[] v);

    void writeSize(int sz);

    void writeProxy(ObjectPrx v);

    void writeObject(Ice.Object v);

    void writeTypeId(String id);
```

```
    void writeException(UserException ex);

    void startSlice();
    void endSlice();

    void startEncapsulation();
    void endEncapsulation();

    void writePendingObjects();

    byte[] finished();

    void reset(boolean clearBuffer);

    void writeSerializable(java.io.Serializable o);

    void destroy();
}
```

Member functions are provided for inserting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void writeSize(int sz)`

  The Ice encoding has a compact representation to indicate size (see Section 37.2.1). This function converts the given non-negative integer into the proper encoded representation.

- `void writeProxy(Ice.ObjectPrx v)`

  Inserts a proxy.

- `void writeObject(Ice.Object v)`

  Inserts an Ice object. The Ice encoding for class instances (see Section 37.2.11) may cause the insertion of this object to be delayed, in which case the stream retains a reference to the given object and does not insert its state it until `writePendingObjects` is invoked on the stream.

- `void writeTypeId(String id)`

  A table of Slice type ids is used to save space when encoding Ice objects (see Section 37.2.11). This function adds the given type id to the table and encodes the type id. `writeTypeId` may only be invoked in the context of a call to `writePendingObjects` (see below).

- `void writeException(UserException ex)`

  Inserts a user exception.

- `void startSlice()`
  `void endSlice()`

  Starts and ends a slice of object or exception member data (see Section 37.2.11).

- `void startEncapsulation()`
  `void endEncapsulation()`

  Starts and ends an encapsulation, respectively (see Section 37.2.2).

- `void writePendingObjects()`

  Encodes the state of Ice objects whose insertion was delayed during `writeObject`. This member function must only be called once.

- `byte[] finished()`

  Indicates that marshaling is complete and returns the encoded byte sequence. This member function must only be called once.

- `void reset(boolean clearBuffer)`

  Resets the writing position of the stream to the beginning. The boolean argument `clearBuffer` determines whether the stream releases the internal buffer it allocated to hold the encoded data. If `clearBuffer` is true, the stream releases the buffer in order to make it eligible for garbage collection. If `clearBuffer` is false, the stream retains the buffer to avoid generating unnecessary garbage.

- `void writeSerializable(java.io.Serializable v)`

  Writes a serializable Java object to the stream (see Section 10.15).

- `void destroy()`

  Applications must call this function in order to reclaim resources.

Here is a simple example that demonstrates how to insert a boolean and a sequence of strings into a stream:

```
final String[] seq = { "Ice", "rocks!" };
Ice.OutputStream out = Ice.Util.createOutputStream(communicator);
try {
    out.writeBool(true);
    out.writeStringSeq(seq);
```

```
    byte[] data = out.finished();
} finally {
    out.destroy();
}
```

Helper Functions

The stream classes provide all of the low-level functions necessary for encoding and decoding Ice types. However, it would be tedious and error-prone to manually encode complex Ice types such as classes, structs, and dictionaries using these low-level functions. For this reason, the Slice compiler (see Section 10.18.1) optionally generates helper functions for streaming complex Ice types.

We will use the following Slice definitions to demonstrate the language mapping:

```
module M {
    sequence<...> Seq;
    dictionary<...> Dict;
    struct S {
        ...
    };
    enum E { ... };
    class C {
        ...
    };
};
```

The Slice compiler generates the corresponding helper functions shown below:

```
package M;

public class SeqHelper {
    public static T[] read(Ice.InputStream in);
    public static void write(Ice.OutputStream out, T[] v);
}

public class DictHelper {
    public static java.util.Map read(Ice.InputStream in);
    public static void write(Ice.OutputStream out,
                             java.util.Map v);
}

public class SHelper {
    public static S read(Ice.InputStream in);
    public static void write(Ice.OutputStream out, S v);
}
```

```
public class EHelper {
    public static E read(Ice.InputStream in);
    public static void write(Ice.OutputStream out, E v);
}

public class CHelper {
    public static void read(Ice.InputStream in, CHolder h);
    public static void write(Ice.OutputStream out, C v);
}

public class CPrxHelper {
    public static CPrx read(Ice.InputStream in);
    public static void write(Ice.OutputStream out, CPrx v);
}
```

In addition, the Slice compiler generates the following member functions for struct and enum types:

```
public class S ... {
    ...
    public void ice_read(Ice.InputStream in);
    public void ice_write(Ice.OutputStream out);
};
public class E... {
    ...
    public void ice_read(Ice.InputStream in);
    public void ice_write(Ice.OutputStream out);
}
```

Be aware that a call to `CHelper.read` does not result in the immediate extraction of an Ice object. The `value` member of the given `CHolder` object is updated when `readPendingObjects` is invoked on the input stream.

Intercepting Object Insertion and Extraction

In some situations it may be necessary to intercept the insertion and extraction of Ice objects. For example, the Ice extension for PHP (see Chapter 28) is implemented using Ice for C++ but represents Ice objects as native PHP objects. The PHP extension accomplishes this by manually encoding and decoding Ice objects as directed by Section 37.2. However, the extension obviously cannot pass a native PHP object to the C++ stream function `writeObject`. To bridge this gap between object systems, Ice supplies the classes `ObjectReader` and `ObjectWriter`:

```
package Ice;

public abstract class ObjectReader extends ObjectImpl {
    public abstract void read(InputStream in, boolean rid);
    // ...
}

public abstract class ObjectWriter extends ObjectImpl {
    public abstract void write(OutputStream out);
    // ...
}
```

A foreign Ice object is inserted into a stream using the following technique:

1. A Java "wrapper" class is derived from `ObjectWriter`. This class wraps the foreign object and implements the `write` member function.
2. An instance of the wrapper class is passed to `writeObject`. (This is possible because `ObjectWriter` derives from `Ice.Object`.) Eventually, the `write` member function is invoked on the wrapper instance.
3. The implementation of `write` encodes the object's state as directed by Section 37.2.11.

It is the application's responsibility to ensure that there is a one-to-one mapping between foreign Ice objects and wrapper objects. This is necessary in order to ensure the proper encoding of object graphs.

Extracting the state of a foreign Ice object is more complicated than insertion:

1. A Java "wrapper" class is derived from `ObjectReader`. An instance of this class represents a foreign Ice object.
2. An object factory is installed that returns instances of the wrapper class. Note that a single object factory can be used for all Slice types if it is registered with an empty Slice type id (see Section 10.14.4).
3. A Java callback class implements the `ReadObjectCallback` interface. The implementation of `invoke` expects its argument to be either null or an instance of the wrapper class as returned by the object factory.
4. An instance of the callback class is passed to `readObject`.
5. When the stream is ready to extract the state of an object, it invokes `read` on the wrapper class. The implementation of `read` decodes the object's state as directed by Section 37.2.11. The boolean argument to read indicates whether the function should invoke `readTypeId` on the stream; it is possible that the type id of the current slice has already been read, in which case this argument is `false`.

6. The callback object passed to `readObject` is invoked, passing the instance of the wrapper object. All other callback objects representing the same instance in the stream (in case of object graphs) are invoked with the same wrapper object.

Intercepting User Exception Insertion

Similar to the discussion of Ice objects in the previous section, a Dynamic Ice application may represent user exceptions in a native format that is not directly compatible with the Ice API. If the application needs to raise such a user exception to the Ice run time, the exception must be wrapped in a subclass of `Ice::UserException`. The Dynamic Ice API provides a class to simplify this process:

```
package Ice;

public abstract class UserExceptionWriter extends UserException {

    public UserExceptionWriter(Communicator communicator);

    public abstract void write(Ice.OutputStream os);
    public abstract boolean usesClasses();

    // ...
}
```

A subclass of `UserExceptionWriter` is responsible for supplying a communicator to the constructor, and for implementing the following methods:

- `void write(OutputStream os)`

  This method is invoked when the Ice run time is ready to marshal the exception. The subclass must marshal the exception using the encoding rules specified in Section 37.2.10.

- `boolean usesClasses()`

  Return true if the exception, or any base exception, contains a data member for an object by value.

35.2.3 C# Stream Interface

We discuss the stream classes first, followed by the helper functions, and finish with an advanced use case of the streaming interface.

InputStream

An `InputStream` is created using the following function:

```
namespace Ice
{
    public sealed class Util
    {
        public static InputStream createInputStream(
                                    Communicator communicator,
                                    byte[] bytes);
    }
}
```

The `InputStream` interface is shown below.

```
namespace Ice
{
    public interface InputStream
    {
        Communicator communicator();

        void sliceObjects(bool slice);

        bool readBool();
        bool[] readBoolSeq();

        byte readByte();
        byte[] readByteSeq();

        short readShort();
        short[] readShortSeq();

        int readInt();
        int[] readIntSeq();

        long readLong();
        long[] readLongSeq();

        float readFloat();
        float[] readFloatSeq();

        double readDouble();
        double[] readDoubleSeq();

        string readString();
        string[] readStringSeq();
```

```
        int readSize();
        int readAndCheckSeqSize(int minSize);

        ObjectPrx readProxy();

        void readObject(ReadObjectCallback cb);

        string readTypeId();

        void throwException();

        void startSlice();
        void endSlice();
        void skipSlice();

        void startEncapsulation();
        void endEncapsulation();
        void skipEncapsulation();
        int getEncapsulationSize();

        void readPendingObjects();

        object readSerializable();

        void rewind();

        void destroy();
    }
}
```

Member functions are provided for extracting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void sliceObjects(boolean slice)`

  Determines the behavior of the stream when extracting Ice objects. An Ice object is "sliced" when a factory cannot be found for a Slice type id (see Section 37.2.11 for more information), resulting in the creation of an object of a less-derived type. Slicing is typically disabled when the application expects all object factories to be present, in which case the exception `NoObjectFactoryException` is raised. The default behavior is to allow slicing.

- `int readSize()`

  The Ice encoding has a compact representation to indicate size (see Section 37.2.1). This function extracts a size and returns it as an integer.

- `int readAndCheckSeqSize(int minWireSize)`

  Like `readSize`, this function reads a size and returns it, but also verifies that there is enough data remaining in the unmarshaling buffer to successfully unmarshal the elements of the sequence. The `minWireSize` parameter indicates the smallest possible on-the-wire representation of a single sequence element (see Chapter 37). If the unmarshaling buffer contains insufficient data to unmarshal the sequence, the function throws `UnmarshalOutOfBoundsException`.

- `Ice.ObjectPrx readProxy()`

  This function returns an instance of the base proxy type, `ObjectPrx`. The Slice compiler optionally generates helper functions to extract proxies of user-defined types (see page 1245).

- `void readObject(ReadObjectCallback cb)`

  The Ice encoding for class instances requires extraction to occur in stages (see Section 37.2.11). The `readObject` function accepts a callback object of type `ReadObjectCallback`, whose definition is shown below:

  ```
  namespace Ice
  {
      public interface ReadObjectCallback
      {
          void invoke(Ice.Object obj);
      }
  }
  ```

  When the object instance is available, the callback object's `invoke` member function is called. The application must call `readPendingObjects` to ensure that all instances are properly extracted.

  Note that applications rarely need to invoke this member function directly; the helper functions generated by the Slice compiler are easier to use (see page 1245).

- `string readTypeId()`

  A table of Slice type ids is used to save space when encoding Ice objects (see Section 37.2.11). This function returns the type id at the stream's current position.

- `void throwException()`

  This function extracts a user exception from the stream and throws it. If the stored exception is of an unknown type, the function attempts to extract and throw a less-derived exception. If that also fails, an `UnmarshalOutOf-BoundsException` is thrown.

- `void startSlice()`
  `void endSlice()`
  `void skipSlice()`

  Start, end, and skip a slice of member data, respectively. These functions are used when manually extracting the slices of an Ice object or user exception. See Section 37.2.11 for more information.

- `void startEncapsulation()`
  `void endEncapsulation()`
  `void skipEncapsulation()`

  Start, end, and skip an encapsulation, respectively. See Section 37.2.2 for more information.

- `int getEncapsulationSize()`

  Returns the size of the current encapsulation in bytes.

- `void readPendingObjects()`

  An application must call this function after all other data has been extracted, but only if Ice objects were encoded. This function extracts the state of Ice objects and invokes their corresponding callback objects (see `readObject`).

- `object readSerializable()`

  Reads a serializable .NET object from the stream (see Section 14.14).

- `void rewind()`

  Resets the position of the stream to the beginning.

- `void destroy()`

  Applications must call this function in order to reclaim resources.

Here is a simple example that demonstrates how to extract a boolean and a sequence of strings from a stream:

```
byte[] data = ...
Ice.InputStream inStream =
    Ice.Util.createInputStream(communicator, data);
try {
    bool b = inStream.readBool();
```

```
    string[] seq = inStream.readStringSeq();
} finally {
    inStream.destroy();
}
```

OutputStream

An `OutputStream` is created using the following function:

```
namespace Ice
{
    public sealed class Util
    {
        public static OutputStream createOutputStream(
                                    Communicator communicator);
    }
}
```

The `OutputStream` class is shown below.

```
namespace Ice
{
    public interface OutputStream
    {
        Communicator communicator();

        void writeBool(bool v);
        void writeBoolSeq(bool[] v);

        void writeByte(byte v);
        void writeByteSeq(byte[] v);

        void writeShort(short v);
        void writeShortSeq(short[] v);

        void writeInt(int v);
        void writeIntSeq(int[] v);

        void writeLong(long v);
        void writeLongSeq(long[] v);

        void writeFloat(float v);
        void writeFloatSeq(float[] v);

        void writeDouble(double v);
        void writeDoubleSeq(double[] v);
```

```
        void writeString(string v);
        void writeStringSeq(string[] v);

        void writeSize(int sz);

        void writeProxy(ObjectPrx v);

        void writeObject(Ice.Object v);

        void writeTypeId(string id);

        void writeException(UserException ex);

        void startSlice();
        void endSlice();

        void startEncapsulation();
        void endEncapsulation();

        void writePendingObjects();

        byte[] finished();

        void reset(bool clearBuffer);

        void writeSerializable(object v);

        void destroy();
    }
}
```

Member functions are provided for inserting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void writeSize(int sz)`

  The Ice encoding has a compact representation to indicate size (see Section 37.2.1). This function converts the given non-negative integer into the proper encoded representation.

- `void writeObject(Ice.Object v)`

  Inserts an Ice object. The Ice encoding for class instances (see Section 37.2.11) may cause the insertion of this object to be delayed, in which case the stream retains a reference to the given object and does not insert its state it until `writePendingObjects` is invoked on the stream.

- `void writeProxy(Ice.ObjectPrx v)`

  Inserts a proxy.

- `void writeTypeId(string id)`

  A table of Slice type ids is used to save space when encoding Ice objects (see Section 37.2.11). This function adds the given type id to the table and encodes the type id. `writeTypeId` may only be invoked in the context of a call to `writePendingObjects` (see below).

- `void writeException(UserException ex)`

  Inserts a user exception.

- `void startSlice()`
  `void endSlice()`

  Starts and ends a slice of object or exception member data (see Section 37.2.11).

- `void startEncapsulation()`
  `void endEncapsulation()`

  Starts and ends an encapsulation, respectively (see Section 37.2.2).

- `void writePendingObjects()`

  Encodes the state of Ice objects whose insertion was delayed during `writeObject`. This member function must only be called once.

- `byte[] finished()`

  Indicates that marshaling is complete and returns the encoded byte sequence. This member function must only be called once.

- `void reset(boolean clearBuffer)`

  Resets the writing position of the stream to the beginning. The boolean argument `clearBuffer` determines whether the stream releases the internal buffer it allocated to hold the encoded data. If `clearBuffer` is true, the stream releases the buffer in order to make it eligible for garbage collection. If `clearBuffer` is false, the stream retains the buffer to avoid generating unnecessary garbage.

- `void writeSerializable(object v)`

  Writes a serializable .NET object to the stream (see Section 14.14).

- `void destroy()`

  Applications must call this function in order to reclaim resources.

Here is a simple example that demonstrates how to insert a boolean and a sequence of strings into a stream:

```
string[] seq = { "Ice", "rocks!" };
Ice.OutputStream outStream
    = Ice.Util.createOutputStream(communicator);
try {
    outStream.writeBool(true);
    outStream.writeStringSeq(seq);
    byte[] data = outStream.finished();
} finally {
    outStream.destroy();
}
```

Helper Functions

The stream classes provide all of the low-level functions necessary for encoding and decoding Ice types. However, it would be tedious and error-prone to manually encode complex Ice types such as classes, structs, and dictionaries using these low-level functions. For this reason, the Slice compiler (see Section 14.17) optionally generates helper functions for streaming complex Ice types.

We will use the following Slice definitions to demonstrate the language mapping:

```
module M {
    sequence<...> Seq;
    dictionary<...> Dict;
    struct S {
        ...
    };
    enum E { ... };
    class C {
        ...
    };
};
```

The Slice compiler generates the corresponding helper functions shown below:

```
namespace M
{
    public sealed class SeqHelper
    {
        public static int[] read(Ice.InputStream _in);
        public static void write(Ice.OutputStream _out, int[] _v);
    }
```

```
    public sealed class DictHelper
    {
        public static Dictionary<...> read(Ice.InputStream _in);
        public static void write(Ice.OutputStream _out,
                                 Dictionary<...> _v);
    }

    public sealed class SHelper
    {
        public static S read(Ice.InputStream _in);
        public static void write(Ice.OutputStream _out, S _v);
    }

    public sealed class EHelper
    {
        public static M.E read(Ice.InputStream _in);
        public static void write(Ice.OutputStream _out, M.E _v);
    }

    public sealed class CHelper
    {
        public CHelper(Ice.InputStream _in);
        public void read();
        public static void write(Ice.OutputStream _out, C _v);
        public M.C value
        {
            get;
        }
        // ...
    }

    public sealed class CPrxHelper : Ice.ObjectPrxHelperBase, CPrx
    {
        public static CPrx read(Ice.InputStream _in);
        public static void write(Ice.OutputStream _out, CPrx _v);
    }
}
```

In addition, the Slice compiler generates the following member functions for struct types:

```
public struct S {
    ...
    public void ice_read(Ice.InputStream in);
    public void ice_write(Ice.OutputStream out);
}
```

Be aware that a call to `CHelper.read` does not result in the immediate extraction of an Ice object. The `value` property of the given `CHelper` object is updated when `readPendingObjects` is invoked on the input stream.

Intercepting Object Insertion and Extraction

In some situations it may be necessary to intercept the insertion and extraction of Ice objects. For example, the Ice extension for PHP (see Chapter 28) is implemented using Ice for C++ but represents Ice objects as native PHP objects. The PHP extension accomplishes this by manually encoding and decoding Ice objects as directed by Section 37.2. However, the extension obviously cannot pass a native PHP object to the C++ stream function `writeObject`. To bridge this gap between object systems, Ice supplies the classes `ObjectReader` and `ObjectWriter`:

```
namespace Ice
{
    public abstract class ObjectReader : ObjectImpl
    {
        public abstract void read(InputStream inStream, bool rid);
        // ...
    }

    public abstract class ObjectWriter : ObjectImpl
    {
        public abstract void write(OutputStream outStream);
        // ...
    }
}
```

A foreign Ice object is inserted into a stream using the following technique:

1. A C# "wrapper" class is derived from `ObjectWriter`. This class wraps the foreign object and implements the `write` member function.
2. An instance of the wrapper class is passed to `writeObject`. (This is possible because `ObjectWriter` derives from `Ice.Object`.) Eventually, the `write` member function is invoked on the wrapper instance.
3. The implementation of `write` encodes the object's state as directed by Section 37.2.11.

It is the application's responsibility to ensure that there is a one-to-one mapping between foreign Ice objects and wrapper objects. This is necessary in order to ensure the proper encoding of object graphs.

Extracting the state of a foreign Ice object is more complicated than insertion:

1. A C# "wrapper" class is derived from `ObjectReader`. An instance of this class represents a foreign Ice object.
2. An object factory is installed that returns instances of the wrapper class. Note that a single object factory can be used for all Slice types if it is registered with an empty Slice type id (see Section 10.14.4).
3. A C# callback class implements the `ReadObjectCallback` interface. The implementation of `invoke` expects its argument to be either null or an instance of the wrapper class as returned by the object factory.
4. An instance of the callback class is passed to `readObject`.
5. When the stream is ready to extract the state of an object, it invokes `read` on the wrapper class. The implementation of `read` decodes the object's state as directed by Section 37.2.11. The boolean argument to read indicates whether the function should invoke `readTypeId` on the stream; it is possible that the type id of the current slice has already been read, in which case this argument is `false`.
6. The callback object passed to `readObject` is invoked, passing the instance of the wrapper object. All other callback objects representing the same instance in the stream (in case of object graphs) are invoked with the same wrapper object.

Intercepting User Exception Insertion

Similar to the discussion of Ice objects in the previous section, a Dynamic Ice application may represent user exceptions in a native format that is not directly compatible with the Ice API. If the application needs to raise such a user exception to the Ice run time, the exception must be wrapped in a subclass of `Ice::UserException`. The Dynamic Ice API provides a class to simplify this process:

```
namespace Ice
{
    public abstract class UserExceptionWriter : UserException
    {
        public UserExceptionWriter(Communicator communicator);

        public abstract void write(OutputStream os);
        public abstract bool usesClasses();

        // ...
    }
}
```

A subclass of `UserExceptionWriter` is responsible for supplying a communicator to the constructor, and for implementing the following methods:

- `void write(OutputStream os)`

  This method is invoked when the Ice run time is ready to marshal the exception. The subclass must marshal the exception using the encoding rules specified in Section 37.2.10.

- `bool usesClasses()`

  Return true if the exception, or any base exception, contains a data member for an object by value.

35.3 Dynamic Invocation and Dispatch

Ice applications generally use the static invocation model, in which the application invokes a Slice operation by calling a member function on a generated proxy class. In the server, the static dispatch model behaves similarly: the request is dispatched to the servant as a statically-typed call to a member function. Underneath this statically-typed facade, the Ice run times in the client and server are exchanging sequences of bytes representing the encoded request arguments and results. These interactions are illustrated in Figure 35.1.

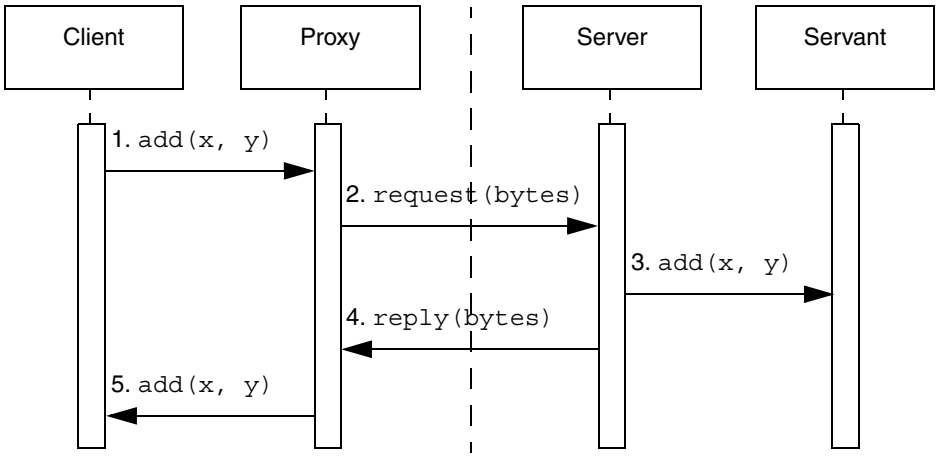


Figure 35.1. Interactions in a static invocation.

1. The client initiates a call to the Slice operation `add` by calling the member function `add` on a proxy.
2. The generated proxy class marshals the arguments into a sequence of bytes and transmits them to the server.
3. In the server, the generated servant class unmarshals the arguments and calls `add` on the subclass.
4. The servant marshals the results and returns them to the client.
5. Finally, the client's proxy unmarshals the results and returns them to the caller.

The application is blissfully unaware of this low-level machinery, and in the majority of cases that is a distinct advantage. In some situations, however, an application can leverage this machinery to accomplish tasks that are not possible in a statically-typed environment. Ice provides the dynamic invocation and dispatch models for these situations, allowing applications to send and receive requests as encoded sequences of bytes instead of statically-typed arguments.

The dynamic invocation and dispatch models offer several unique advantages to Ice services that forward requests from senders to receivers, such as Glacier2 (Chapter 42) and IceStorm (Chapter 44). For these services, the request arguments are an opaque byte sequence that can be forwarded without the need to unmarshal and remarshal the arguments. Not only is this significantly more efficient than a statically-typed implementation, it also allows intermediaries such as Glacier2 and IceStorm to be ignorant of the Slice types in use by senders and receivers.

Another use case for the dynamic invocation and dispatch models is scripting language integration. The Ice extensions for Python, PHP, and Ruby invoke Slice operations using the dynamic invocation model; the request arguments are encoded using the streaming interfaces from Section 35.2.

It may be difficult to resist the temptation of using a feature like dynamic invocation or dispatch, but we recommend that you carefully consider the risks and complexities of such a decision. For example, an application that uses the streaming interface to manually encode and decode request arguments has a high risk of failure if the argument signature of an operation changes. In contrast, this risk is greatly reduced in the static invocation and dispatch models because errors in a strongly-typed language are found early, during compilation. Therefore, we caution you against using this capability except where its advantages significantly outweigh the risks.

35.3.1 Dynamic Invocation using `ice_invoke`

Dynamic invocation is performed using the proxy member function `ice_invoke`, defined in the proxy base class `ObjectPrx`. If we were to define the function in Slice, it would look like this:

```
sequence<byte> ByteSeq;

bool ice_invoke(
    string operation,
    Ice::OperationMode mode,
    ByteSeq inParams,
    out ByteSeq outParams
);
```

The first argument is the name of the Slice operation[2]. The second argument is an enumerator from the Slice type `Ice::OperationMode`; the possible values are `Normal` and `Idempotent`. The third argument, `inParams`, represents the encoded `in` parameters of the operation.

A return value of `true` indicates a successful invocation, in which case the marshaled form of the operation's results (if any) is provided in `outParams`. A return value of `false` signals the occurrence of a user exception whose marshaled form is provided in `outParams`. The caller must also be prepared to catch local exceptions, which are thrown directly.

Note that the Ice run time currently does not support the use of collocation optimization in dynamic invocations. Attempting to call `ice_invoke` on a proxy that is configured to use collocation optimization raises `CollocationOptimizationException`. See Section 32.21 for more information on this optimization and instructions for disabling it.

35.3.2 Dynamic Dispatch using `Blobject`

A server enables dynamic dispatch by creating a subclass of `Blobject` (the name is derived from *blob*, meaning a blob of bytes). The Slice equivalent of `Blobject` is shown below:

2. This is the Slice name of the operation, not the name as it might be mapped to any particular language. For example, the string `"while"` is the name of the Slice operation `while`, and not `"_cpp_while"` (C++) or `"_while"` (Java).

```
sequence<byte> ByteSeq;

interface Blobject {
    bool ice_invoke(ByteSeq inParams, out ByteSeq outParams);
};
```

The `inParams` argument supplies the encoded `in` parameters. The contents of the `outParams` argument depends on the outcome of the invocation: if the operation succeeded, `ice_invoke` must return `true` and place the encoded results in `outParams`; if a user exception occurred, `ice_invoke` must return `false`, in which case `outParams` contains the encoded exception. The operation may also raise local exceptions such as `OperationNotExistException`.

The language mappings add a trailing argument of type `Ice::Current` to `ice_invoke`, and this provides the implementation with the name of the operation being dispatched. See Section 32.6 for more information on `Ice::Current`.

Because `Blobject` derives from `Object`, an instance is a regular Ice object just like instances of the classes generated for user-defined Slice interfaces. The primary difference is that all operation invocations on a `Blobject` instance are dispatched through the `ice_invoke` member function.

If a `Blobject` subclass intends to decode the `in` parameters (and not simply forward the request to another object), then the implementation obviously must know the signatures of all operations it supports. How a `Blobject` subclass determines its type information is an implementation detail that is beyond the scope of this book.

Note that a `Blobject` servant is also useful if you want to create a message forwarding service, such as Glacier2 (see Chapter 42). In this case, there is no need to decode any parameters; instead, the implementation simply forwards each request unchanged to a new destination. You can register a `Blobject` servant as a default servant (see Section 32.8) to easily achieve this.

35.3.3 C++ Mapping

This section describes the C++ mapping for the `ice_invoke` proxy function and the `Blobject` class.

`ice_invoke`

The mapping for `ice_invoke` is shown below:

```
bool ice_invoke(
    const std::string& operation,
    Ice::OperationMode mode,
    const std::vector< Ice::Byte >& inParams,
    std::vector< Ice::Byte >& outParams
);
```

Another overloading of `ice_invoke` (not shown) adds a trailing argument of type `Ice::Context` (see Section 32.12).

As an example, the code below demonstrates how to invoke the operation `op`, which takes no `in` parameters:

```
Ice::ObjectPrx proxy = ...
try {
    std::vector<Ice::Byte> inParams, outParams;
    if (proxy->ice_invoke("op", Ice::Normal, inParams,
                          outParams)) {
        // Handle success
    } else {
        // Handle user exception
    }
} catch (const Ice::LocalException& ex) {
    // Handle exception
}
```

Using Streams with `ice_invoke`

The streaming interface described in Section 35.2 provides the tools an application needs to dynamically invoke operations with arguments of any Slice type. Consider the following Slice definition:

```
module Calc {
    exception Overflow {
        int x;
        int y;
    };
    interface Compute {
        idempotent int add(int x, int y)
            throws Overflow;
    };
};
```

Now let's write a client that dynamically invokes the `add` operation:

```
Ice::ObjectPrx proxy = ...
try {
    std::vector< Ice::Byte > inParams, outParams;

    Ice::OutputStreamPtr out =
        Ice::createOutputStream(communicator);
    out->writeInt(100); // x
    out->writeInt(-1);  // y
    out->finished(inParams);

    if (proxy->ice_invoke("add", Ice::Idempotent, inParams,
                          outParams)) {
        // Handle success
        Ice::InputStreamPtr in =
            Ice::createInputStream(communicator, outParams);
        int result = in->readInt();
        assert(result == 99);
    } else {
        // Handle user exception
    }
} catch (const Ice::LocalException& ex) {
    // Handle exception
}
```

We neglected to handle the case of a user exception in this example, so let's implement that now. We assume that we have compiled our program with the Slice-generated code, therefore we can call `throwException` on the input stream and catch `Overflow` directly[3]:

```
    if (proxy->ice_invoke("add", Ice::Idempotent, inParams,
                          outParams)) {
        // Handle success
        // ...
    } else {
        // Handle user exception
        Ice::InputStreamPtr in =
            Ice::createInputStream(communicator, outParams);
        try {
            in->throwException();
        } catch (const Calc::Overflow& ex) {
```

3. This is obviously a contrived example: if the Slice-generated code is available, why bother using dynamic dispatch? In the absence of Slice-generated code, the caller would need to manually unmarshal the user exception, which is outside the scope of this book.

```
            cout << "overflow while adding " << ex.x
                 << " and " << ex.y << endl;
        } catch (const Ice::UserException& ex) {
            // Handle unexpected user exception
        }
    }
```

As a defensive measure, the code traps `Ice::UserException`. This could be raised if the Slice definition of `add` is modified to include another user exception but this segment of code did not get updated accordingly.

Subclassing `Blobject`

Implementing the dynamic dispatch model requires writing a subclass of `Ice::Blobject`. We continue using the `Compute` interface from page 1253 to demonstrate a `Blobject` implementation:

```
class ComputeI : public Ice::Blobject {
public:
    virtual bool ice_invoke(
        const std::vector<Ice::Byte>& inParams,
        std::vector<Ice::Byte>& outParams,
        const Ice::Current& current);
};
```

An instance of `ComputeI` is an Ice object because `Blobject` derives from `Object`, therefore an instance can be added to an object adapter like any other servant (see Chapter 32 for more information on object adapters).

For the purposes of this discussion, the implementation of `ice_invoke` handles only the `add` operation and raises `OperationNotExistException` for all other operations. In a real implementation, the servant must also be prepared to receive invocations of the following operations:

- `string ice_id()`

  Returns the Slice type id of the servant's most-derived type.

- `StringSeq ice_ids()`

  Returns a sequence of strings representing all of the Slice interfaces supported by the servant, including `"::Ice::Object"`.

- `bool ice_isA(string id)`

  Returns `true` if the servant supports the interface denoted by the given Slice type id, or `false` otherwise. This operation is invoked by the proxy function `checkedCast`.

- `void ice_ping()`

  Verifies that the object denoted by the identity and facet contained in `Ice::Current` is reachable.

With that in mind, here is our simplified version of `ice_invoke`:

```
bool ComputeI::ice_invoke(
    const std::vector<Ice::Byte>& inParams,
    std::vector<Ice::Byte>& outParams,
    const Ice::Current& current)
{
    if (current.operation == "add") {
        Ice::CommunicatorPtr communicator =
            current.adapter->getCommunicator();
        Ice::InputStreamPtr in =
            Ice::createInputStream(communicator, inParams);
        int x = in->readInt();
        int y = in->readInt();
        Ice::OutputStreamPtr out =
            Ice::createOutputStream(communicator);
        if (checkOverflow(x, y)) {
            Calc::Overflow ex;
            ex.x = x;
            ex.y = y;
            out->writeException(ex);
            out->finished(outParams);
            return false;
        } else {
            out->writeInt(x + y);
            out->finished(outParams);
            return true;
        }
    } else {
        Ice::OperationNotExistException ex(__FILE__, __LINE__);
        ex.id = current.id;
        ex.facet = current.facet;
        ex.operation = current.operation;
        throw ex;
    }
}
```

If an overflow is detected, the code “raises” the `Calc::Overflow` user exception by calling `writeException` on the output stream and returning `false`, otherwise the return value is encoded and the function returns `true`.

Array Mapping

Ice for C++ supports an alternative mapping for `sequence` input parameters that avoids the overhead of extra copying. Since the `ice_invoke` functions treat the encoded input parameters as a value of type `sequence<byte>`, the dynamic invocation and dispatch facility includes interfaces that use the array mapping for the input parameter blob.

Ice provides two overloaded versions of the proxy function `ice_invoke` that use the array mapping. The version that omits the trailing `Ice::Context` argument is shown below:

```
bool ice_invoke(
    const std::string& operation,
    Ice::OperationMode mode,
    const std::pair< const Ice::Byte*, const Ice::Byte* >& in,
    std::vector< Ice::Byte >& out
);
```

A Blobject servant uses the array mapping by deriving its implementation class from `Ice::BlobjectArray` and overriding its `ice_invoke` function:

```
class BlobjectArray {
public:
    virtual bool ice_invoke(
        const std::pair<const Ice::Byte*, const Ice::Byte*>& in,
        std::vector<Ice::Byte>& out,
        const Ice::Current& current) = 0;
};
```

See Section 6.7.4 for more information on the array mapping.

35.3.4 Java Mapping

This section describes the Java mapping for the `ice_invoke` proxy function and the `Blobject` class.

`ice_invoke`

The mapping for `ice_invoke` is shown below:

```
boolean ice_invoke(
    String operation,
    Ice.OperationMode mode,
    byte[] inParams,
    Ice.ByteSeqHolder outParams
);
```

Another overloading of `ice_invoke` (not shown) adds a trailing argument of type `Ice.Context` (see Section 32.12).

As an example, the code below demonstrates how to invoke the operation `op`, which takes no `in` parameters:

```
Ice.ObjectPrx proxy = ...
try {
    Ice.ByteSeqHolder outParams = new Ice.ByteSeqHolder();
    if (proxy.ice_invoke("op", Ice.OperationMode.Normal, null,
                         outParams)) {
        // Handle success
    } else {
        // Handle user exception
    }
} catch (Ice.LocalException ex) {
    // Handle exception
}
```

Using Streams with `ice_invoke`

The streaming interface described in Section 35.2 provides the tools an application needs to dynamically invoke operations with arguments of any Slice type. Consider the following Slice definition:

```
module Calc {
    exception Overflow {
        int x;
        int y;
    };
    interface Compute {
        idempotent int add(int x, int y)
            throws Overflow;
    };
};
```

Now let's write a client that dynamically invokes the `add` operation:

```
Ice.ObjectPrx proxy = ...
try {
    Ice.OutputStream out =
        Ice.Util.createOutputStream(communicator);
    out.writeInt(100); // x
    out.writeInt(-1);  // y
    byte[] inParams = out.finished();
    Ice.ByteSeqHolder outParams = new Ice.ByteSeqHolder();
    if (proxy.ice_invoke("add", Ice.OperationMode.Idempotent,
```

```
                        inParams, outParams)) {
        // Handle success
        Ice.InputStream in =
            Ice.Util.createInputStream(communicator,
                                       outParams.value);
        int result = in.readInt();
        assert(result == 99);
    } else {
        // Handle user exception
    }
} catch (Ice.LocalException ex) {
    // Handle exception
}
```

We neglected to handle the case of a user exception in this example, so let's implement that now. We assume that we have compiled our program with the Slice-generated code, therefore we can call `throwException` on the input stream and catch `Overflow` directly[4]:

```
    if (proxy.ice_invoke("add", Ice.OperationMode.Idempotent,
                        inParams, outParams)) {
        // Handle success
        // ...
    } else {
        // Handle user exception
        Ice.InputStream in =
            Ice.Util.createInputStream(communicator,
                                       outParams.value);
        try {
            in.throwException();
        } catch (Calc.Overflow ex) {
            System.out.println("overflow while adding " + ex.x +
                               " and " + ex.y);
        } catch (Ice.UserException ex) {
            // Handle unexpected user exception
        }
    }
```

4. This is obviously a contrived example: if the Slice-generated code is available, why bother using dynamic dispatch? In the absence of Slice-generated code, the caller would need to manually unmarshal the user exception, which is outside the scope of this book.

As a defensive measure, the code traps `Ice.UserException`. This could be raised if the Slice definition of `add` is modified to include another user exception but this segment of code did not get updated accordingly.

Subclassing `Blobject`

Implementing the dynamic dispatch model requires writing a subclass of `Ice.Blobject`. We continue using the `Compute` interface from page 1258 to demonstrate a `Blobject` implementation:

```
public class ComputeI extends Ice.Blobject {
    public boolean ice_invoke(
        byte[] inParams,
        Ice.ByteSeqHolder outParams,
        Ice.Current current)
    {
        // ...
    }
}
```

An instance of `ComputeI` is an Ice object because `Blobject` derives from `Object`, therefore an instance can be added to an object adapter like any other servant (see Chapter 32 for more information on object adapters).

For the purposes of this discussion, the implementation of `ice_invoke` handles only the `add` operation and raises `OperationNotExistException` for all other operations. In a real implementation, the servant must also be prepared to receive invocations of the following operations:

- `string ice_id()`

  Returns the Slice type id of the servant's most-derived type.

- `StringSeq ice_ids()`

  Returns a sequence of strings representing all of the Slice interfaces supported by the servant, including `"::Ice::Object"`.

- `bool ice_isA(string id)`

  Returns `true` if the servant supports the interface denoted by the given Slice type id, or `false` otherwise. This operation is invoked by the proxy function `checkedCast`.

- `void ice_ping()`

  Verifies that the object denoted by the identity and facet contained in `Ice.Current` is reachable.

With that in mind, here is our simplified version of `ice_invoke`:

```
public boolean ice_invoke(
    byte[] inParams,
    Ice.ByteSeqHolder outParams,
    Ice.Current current)
{
    if (current.operation.equals("add")) {
        Ice.Communicator communicator =
            current.adapter.getCommunicator();
        Ice.InputStream in =
            Ice.Util.createInputStream(communicator,
                                       inParams);
        int x = in.readInt();
        int y = in.readInt();
        Ice.OutputStream out =
            Ice.Util.createOutputStream(communicator);
        try {
            if (checkOverflow(x, y)) {
                Calc.Overflow ex = new Calc.Overflow();
                ex.x = x;
                ex.y = y;
                out.writeException(ex);
                outParams.value = out.finished();
                return false;
            } else {
                out.writeInt(x + y);
                outParams.value = out.finished();
                return true;
            }
        } finally {
            out.destroy();
        }
    } else {
        Ice.OperationNotExistException ex =
            new Ice.OperationNotExistException();
        ex.id = current.id;
        ex.facet = current.facet;
        ex.operation = current.operation;
        throw ex;
    }
}
```

If an overflow is detected, the code "raises" the `Calc::Overflow` user exception by calling `writeException` on the output stream and returning `false`, otherwise the return value is encoded and the function returns `true`.

35.3.5 C# Mapping

This section describes the C# mapping for the `ice_invoke` proxy function and the `Blobject` class.

`ice_invoke`

The mapping for `ice_invoke` is shown below:

```
namespace Ice
{
    public interface ObjectPrx
    {
        bool ice_invoke(string operation,
                        OperationMode mode,
                        byte[] inParams,
                        out byte[] outParams);
        // ...
    }
}
```

Another overloading of `ice_invoke` (not shown) adds a trailing argument of type `Ice.Context` (see Section 32.12).

As an example, the code below demonstrates how to invoke the operation `op`, which takes no `in` parameters:

```
Ice.ObjectPrx proxy = ...
try {
    byte[] outParams;
    if (proxy.ice_invoke("op", Ice.OperationMode.Normal, null,
                         outParams)) {
        // Handle success
    } else {
        // Handle user exception
    }
} catch (Ice.LocalException ex) {
    // Handle exception
}
```

Using Streams with `ice_invoke`

The streaming interface described in Section 35.2 provides the tools an application needs to dynamically invoke operations with arguments of any Slice type. Consider the following Slice definition:

```
module Calc {
    exception Overflow {
        int x;
        int y;
    };
    interface Compute {
        idempotent int add(int x, int y)
            throws Overflow;
    };
};
```

Now let's write a client that dynamically invokes the `add` operation:

```
Ice.ObjectPrx proxy = ...
try {
    Ice.OutputStream outStream =
        Ice.Util.createOutputStream(communicator);
    outStream.writeInt(100); // x
    outStream.writeInt(-1);  // y
    byte[] inParams = outStream.finished();
    byte[] outParams;
    if (proxy.ice_invoke("add", Ice.OperationMode.Idempotent,
                         inParams, out outParams)) {
        // Handle success
        Ice.InputStream inStream =
            Ice.Util.createInputStream(communicator, outParams);
        int result = inStream.readInt();
        System.Diagnostics.Debug.Assert(result == 99);
    } else {
        // Handle user exception
    }
} catch (Ice.LocalException ex) {
    // Handle exception
}
```

We neglected to handle the case of a user exception in this example, so let's implement that now. We assume that we have compiled our program with the Slice-generated code, therefore we can call `throwException` on the input stream and catch `Overflow` directly[5]:

5. This is obviously a contrived example: if the Slice-generated code is available, why bother using dynamic dispatch? In the absence of Slice-generated code, the caller would need to manually unmarshal the user exception, which is outside the scope of this book.

```
    if (proxy.ice_invoke("add", Ice.OperationMode.Idempotent,
                         inParams, out outParams)) {
        // Handle success
        ...
    } else {
        // Handle user exception
        Ice.InputStream inStream =
            Ice.Util.createInputStream(communicator, outParams);
        try {
            inStream.throwException();
        } catch (Calc.Overflow ex) {
            System.Console.WriteLine("overflow while adding " +
                                     ex.x + " and " + ex.y);
        } catch (Ice.UserException) {
            // Handle unexpected user exception
        }
    }
```

As a defensive measure, the code traps `Ice.UserException`. This could be raised if the Slice definition of `add` is modified to include another user exception but this segment of code did not get updated accordingly.

Subclassing `Blobject`

Implementing the dynamic dispatch model requires writing a subclass of `Ice.Blobject`. We continue using the `Compute` interface from page 1262 to demonstrate a `Blobject` implementation:

```
public class ComputeI : Ice.Blobject {
    public bool ice_invoke(
                    byte[] inParams,
                    out byte[] outParams,
                    Ice.Current current);
    {
        ...
    }
}
```

An instance of `ComputeI` is an Ice object because `Blobject` derives from `Object`, therefore an instance can be added to an object adapter like any other servant (see Chapter 32 for more information on object adapters).

For the purposes of this discussion, the implementation of `ice_invoke` handles only the `add` operation and raises `OperationNotExistException` for all other operations. In a real implementation, the servant must also be prepared to receive invocations of the following operations:

- `string ice_id()`

  Returns the Slice type id of the servant's most-derived type.
- `StringSeq ice_ids()`

  Returns a sequence of strings representing all of the Slice interfaces supported by the servant, including `"::Ice::Object"`.
- `bool ice_isA(string id)`

  Returns `true` if the servant supports the interface denoted by the given Slice type id, or `false` otherwise. This operation is invoked by the proxy function `checkedCast`.
- `void ice_ping()`

  Verifies that the object denoted by the identity and facet contained in `Ice.Current` is reachable.

With that in mind, here is our simplified version of `ice_invoke`:

```
public bool ice_invoke(
                byte[] inParams,
                out byte[] outParams,
                Ice.Current current);
{
    if (current.operation.Equals("add")) {
        Ice.Communicator communicator =
            current.adapter.getCommunicator();
        Ice.InputStream inStream =
            Ice.Util.createInputStream(communicator,
                                       inParams);
        int x = inStream.readInt();
        int y = inStream.readInt();
        Ice.OutputStream outStream =
            Ice.Util.createOutputStream(communicator);
        try {
            if (checkOverflow(x, y)) {
                Calc.Overflow ex = new Calc.Overflow();
                ex.x = x;
                ex.y = y;
                outStream.StreamwriteException(ex);
                outParams = outStream.finished();
                return false;
            } else {
                outStream.writeInt(x + y);
                outParams = outStream.finished();
                return true;
            }
```

```
            } finally {
                outStream.destroy();
            }
        } else {
            Ice.OperationNotExistException ex =
                new Ice.OperationNotExistException();
            ex.id = current.id;
            ex.facet = current.facet;
            ex.operation = current.operation;
            throw ex;
        }
    }
```

If an overflow is detected, the code "raises" the `Calc::Overflow` user exception by calling `writeException` on the output stream and returning `false`, otherwise the return value is encoded and the function returns `true`.

35.4 Asynchronous Dynamic Invocation and Dispatch

Ice provides asynchronous support for the dynamic invocation and dispatch models described in Section 35.3. The mappings for the `ice_invoke` proxy function and the `Blobject` class adhere to the asynchronous mapping rules.

35.4.1 C++ Mapping

This section describes the asynchronous C++ mapping for the `ice_invoke` proxy function and the `Blobject` class.

ice_invoke_async

The asynchronous mapping for `ice_invoke` produces a function named `ice_invoke_async`, as shown below:

```
bool ice_invoke_async(
    const Ice::AMI_Object_ice_invokePtr& cb,
    const std::string& operation,
    Ice::OperationMode mode,
    const std::vector<Ice::Byte>& inParams
);
```

Another overloading of `ice_invoke_async` (not shown) adds a trailing argument of type `Ice::Context` (see Section 32.12).

The return value and the parameters `operation`, `mode`, and `inParams` have the same semantics as for `ice_invoke` (see Section 35.3.1). As with any other asynchronous invocation, the first argument to the proxy function is always a callback object. In this case, the callback object must derive from the class `Ice::AMI_Object_ice_invoke`, shown here:

```
namespace Ice {
    class AMI_Object_ice_invoke : ... {
    public:
        virtual void ice_response(
            bool result,
            const std::vector<Ice::Byte>& outParams) = 0;

        virtual void ice_exception(const Ice::Exception& ex) = 0;

        // ...
    };
}
```

The `ice_response` function is invoked for a successful completion or when a user exception occurs. A value of `true` for the first argument signals success; results from the operation are encoded in the second argument `outParams`. A value of `false` for the first argument indicates a user exception was raised, and the encoded form of the exception is provided in `outParams`.

The `ice_exception` function is invoked only when the operation raises a local exception such as `OperationNotExistException`.

BlobjectAsync

`BlobjectAsync` is the name of the asynchronous counterpart to `Blobject`:

```
namespace Ice {
    class BlobjectAsync : virtual public Ice::Object {
    public:
        virtual void ice_invoke_async(
            const AMD_Object_ice_invokePtr& cb,
            const std::vector<Ice::Byte>& inParams,
            const Ice::Current& current) = 0;
    };
}
```

To implement asynchronous dynamic dispatch, a server must subclass `BlobjectAsync` and override `ice_invoke_async`.

As with any other asynchronous operation, the first argument to the servant's member function is always a callback object. In this case, the callback object is of type `Ice::AMD_Object_ice_invoke`, shown here:

```
namespace Ice {
    class AMD_Object_ice_invoke : ... {
    public:
        virtual void ice_response(
            bool result,
            const std::vector<Ice::Byte>& outParams) = 0;
        virtual void ice_response(
            bool result,
            const std::pair<const Ice::Byte*,
                            const Ice::Byte*>& outParams) = 0;
        virtual void ice_exception(const std::exception&) = 0;
        virtual void ice_exception() = 0;
    };
}
```

Upon a successful invocation, the servant must invoke one of the `ice_response` methods on the callback object, passing `true` as the first argument and encoding the operation results into `outParams`. To report a user exception, the servant invokes `ice_response` with `false` as the first argument and the encoded form of the exception in `outParams`. Note that the second overloading of `ice_response` uses the array mapping (see the next section below).

The various overloadings of `ice_exception` are discussed in Appendix K. Note however that in the dynamic dispatch model, the `ice_exception` function must not be used to report user exceptions; doing so results in the caller receiving `UnknownUserException`.

Array Mapping

The discussion of the synchronous interfaces on page 1257 presented the array mapping for the `ice_invoke` proxy functions and the `BlobjectArray` base class. The array mapping is also supported in the asynchronous interfaces for dynamic invocation and dispatch.

Ice provides two overloaded versions of the proxy function `ice_invoke_async` that use the array mapping. The version that omits the trailing `Ice::Context` argument is shown below:

```
bool ice_invoke_async(
    const Ice::AMI_Array_Object_ice_invokePtr& cb,
    const std::string& operation,
    Ice::OperationMode mode,
    const std::pair<const Ice::Byte*, const Ice::Byte*>& in
);
```

The AMI callback class also supports the array mapping for the encoded `out` parameter blob, as you can see in its definition of `ice_response`:

```
class AMI_Array_Object_ice_invoke : ... {
public:
    virtual void ice_response(
        bool result,
        const std::pair<const Ice::Byte*, const Ice::Byte*>& out)
        = 0;

    virtual void ice_exception(const Ice::Exception& ex) = 0;

    // ...
};
```

To implement an asynchronous Blobject servant that uses the array mapping, derive your implementation class from `Ice::BlobjectArrayAsync` and override the `ice_invoke_async` function:

```
class BlobjectArrayAsync : virtual public Ice::Object {
public:
    virtual void ice_invoke_async(
        const AMD_Object_ice_invokePtr& cb,
        const std::pair<const Ice::Byte*, const Ice::Byte*>& in,
        const Ice::Current& current) = 0;
};
```

As shown in the previous section, the AMD callback class provides an overloaded `ice_response` method that supports the array mapping for the encoded `out` parameter blob.

See Section 6.7.4 for more information on the array mapping.

35.4.2 Java Mapping

This section describes the asynchronous Java mapping for the `ice_invoke` proxy function and the `Blobject` class.

ice_invoke_async

The asynchronous mapping for `ice_invoke` produces a function named `ice_invoke_async`, as shown below:

```
public abstract boolean ice_invoke_async(
    Ice.AMI_Object_ice_invoke cb,
    String operation,
    Ice.OperationMode mode,
    byte[] inParams
);
```

Another overloading of `ice_invoke_async` (not shown) adds a trailing argument of type `Ice.Context` (see Section 32.12).

The return value and the parameters `operation`, `mode`, and `inParams` have the same semantics as for `ice_invoke` (see Section 35.3.1). As with any other asynchronous invocation, the first argument to the proxy function is always a callback object. In this case, the callback object must derive from the class `Ice.AMI_Object_ice_invoke`, shown here:

```
package Ice;

public abstract class AMI_Object_ice_invoke ... {
    public abstract void ice_response(boolean result,
                                      byte[] outParams);
    public abstract void ice_exception(LocalException ex);
}
```

The `ice_response` function is invoked for a successful completion or when a user exception occurs. A value of `true` for the first argument signals success; results from the operation are encoded in the second argument `outParams`. A value of `false` for the first argument indicates a user exception was raised, and the encoded form of the exception is provided in `outParams`.

The `ice_exception` function is invoked only when the operation raises a local exception such as `OperationNotExistException`.

BlobjectAsync

`BlobjectAsync` is the name of the asynchronous counterpart to `Blobject`:

```
package Ice;

public abstract class BlobjectAsync extends Ice.ObjectImpl {
    public abstract void ice_invoke_async(
        Ice.AMD_Object_ice_invoke cb,
        byte[] inParams,
```

```
        Ice.Current current
    );

    // ...
}
```

To implement asynchronous dynamic dispatch, a server must subclass `BlobjectAsync` and override `ice_invoke_async`.

As with any other asynchronous operation, the first argument to the servant's member function is always a callback object. In this case, the callback object is of type `Ice.AMD_Object_ice_invoke`, shown here:

```
package Ice;

public interface AMD_Object_ice_invoke {
    void ice_response(boolean result, byte[] outParams);
    void ice_exception(java.lang.Exception ex);
}
```

Upon a successful invocation, the servant must invoke `ice_response` on the callback object, passing `true` as the first argument and encoding the operation results into `outParams`. To report a user exception, the servant invokes `ice_response` with `false` as the first argument and the encoded form of the exception in `outParams`.

The `ice_exception` function is discussed in Appendix K. Note however that in the dynamic dispatch model, the `ice_exception` function must not be used to report user exceptions; doing so results in the caller receiving `UnknownUserException`.

35.4.3 C# Mapping

This section describes the asynchronous C# mapping for the `ice_invoke` proxy function and the `Blobject` class.

ice_invoke_async

The asynchronous mapping for `ice_invoke` produces a function named `ice_invoke_async`, as shown below:

```
namespace Ice {
    public interface ObjectPrx {
        bool ice_invoke_async(AMI_Object_ice_invoke cb,
                              string operation,
```

```
                                OperationMode mode,
                                byte[] inParams);
    }
}
```

Another overloading of `ice_invoke_async` (not shown) adds a trailing argument of type `Ice.Context` (see Section 32.12).

The return value and the parameters `operation`, `mode`, and `inParams` have the same semantics as for `ice_invoke` (see Section 35.3.1). As with any other asynchronous invocation, the first argument to the proxy function is always a callback object. In this case, the callback object must derive from the class `Ice.AMI_Object_ice_invoke`, shown here:

```
namespace Ice {

    public abstract class AMI_Object_ice_invoke : ...
    {
        public abstract void ice_response(
                                bool ok, byte[] outParams);
        public abstract override void ice_exception(
                                Ice.Exception ex);
    }

}
```

The `ice_response` function is invoked for a successful completion or when a user exception occurs. A value of `true` for the first argument signals success; results from the operation are encoded in the second argument `outParams`. A value of `false` for the first argument indicates a user exception was raised, and the encoded form of the exception is provided in `outParams`.

The `ice_exception` function is invoked only when the operation raises a local exception such as `OperationNotExistException`.

BlobjectAsync

`BlobjectAsync` is the name of the asynchronous counterpart to `Blobject`:

```
public abstract class BlobjectAsync
    : Ice.ObjectImpl
{
    public abstract void ice_invoke_async(
                        AMD_Object_ice_invoke cb,
                        byte[] inParams,
                        Current current);
}
```

To implement asynchronous dynamic dispatch, a server must subclass `BlobjectAsync` and override `ice_invoke_async`.

As with any other asynchronous operation, the first argument to the servant's member function is always a callback object. In this case, the callback object is of type `Ice.AMD_Object_ice_invoke`, shown here:

```
namespace Ice
{
    public interface AMD_Object_ice_invoke
    {
        void ice_response(bool ok, byte[] outParams);
        void ice_exception(System.Exception ex);
    }
}
```

Upon a successful invocation, the servant must invoke `ice_response` on the callback object, passing `true` as the first argument and encoding the operation results into `outParams`. To report a user exception, the servant invokes `ice_response` with `false` as the first argument and the encoded form of the exception in `outParams`.

The `ice_exception` function is discussed in Appendix K. Note however that in the dynamic dispatch model, the `ice_exception` function must not be used to report user exceptions; doing so results in the caller receiving `UnknownUserException`.

35.5 Summary

The Ice streaming API allows you to serialize and deserialize Slice types using either the Ice encoding or an encoding of your choice (for example, XML). This is useful, for example, if you want to store Slice types in a data base.

The dynamic invocation and dispatch interfaces allow you to write generic clients and servers that need not have compile-time knowledge of the Slice types used by an application. This is useful for applications such as object browsers, protocol analyzers, or protocol bridges. In addition, the dynamic invocation and dispatch interfaces permit services such as IceStorm to be implemented without the need to unmarshal and remarshal every message, with considerable performance improvements.

Keep in mind that applications that use dynamic invocation and dispatch are tedious to implement and harder to prove correct (because what normally would be a compile-time error appears only as a run-time error with dynamic invocation

and dispatch). Therefore, you should use the dynamic interfaces only if your application truly benefits from this trade-off.

Chapter 36
Connection Management

36.1 Chapter Overview

In this chapter we describe the semantics of Ice connections. Section 36.3 defines the rules for connection establishment, including the consequences of connection failure and the effects of timeouts. Section 36.4 provides details about active connection management and when to use it. Section 36.5 demonstrates how Ice applications gain access to connections, while Section 36.6 describes how a connection is closed. Finally, Section 36.7 presents bidirectional connections and describes their use cases and configuration.

36.2 Introduction

The Ice run time establishes connections automatically and transparently as a side effect of using proxies. There are well-defined rules that determine when a new connection is established (see Section 36.3). If necessary, you can influence connection management activities (see Section 36.4).

Connection management becomes increasingly important as network environments grow more complex. In particular, if you need to make callbacks from a server to a client through a firewall, you must use a bidirectional connection. In most cases, you can use a Glacier2 router (see Chapter 42) to automatically take

advantage of bidirectional connections. However, the Ice run time also provides direct access to connections, allowing you to explicitly control establishment and closure of both unidirectional and bidirectional connections.

The discussion that follows assumes that you are familiar with proxies and endpoints (see Section 32.11 and Appendix E).

36.3 Connection Establishment

Connections are established as a side effect of using proxies. The first invocation on a proxy causes the Ice run time to search for an existing connection to one of the proxy's endpoints (see Section 32.11.3); only if no suitable connection exists does the Ice run time establish a new connection to one of the proxy's endpoints.

36.3.1 Endpoint Selection

A proxy performs a number of operations on its endpoints before it asks the Ice run time to supply a connection. These operations produce a list of zero or more endpoints that satisfy the proxy's configuration. If the resulting list is empty, the application receives `NoEndpointException` to indicate that no suitable endpoints could be found. For example, this situation can arise when a twoway proxy contains only a UDP endpoint; the UDP endpoint is eliminated from consideration because it cannot be used for twoway invocations.

The proxy performs the following steps to derive its endpoint list:

1. Remove the endpoints of unknown transports. For instance, SSL endpoints are removed if the SSL plug-in is not installed.
2. Remove endpoints that are not suitable for the proxy's invocation mode. For example, datagram endpoints are removed for twoway, oneway and batch oneway proxies. Similarly, non-datagram endpoints are removed for datagram and batch datagram proxies.
3. Perform DNS queries to convert host names into IP addresses, if necessary. For a multi-homed host name, the proxy adds a new endpoint for each address returned by the DNS query.
4. Sort the endpoints according to the configured selection type, which is established using the `ice_endpointSelection` factory method. The default value is `Random`, meaning the endpoints are randomly shuffled. Alternatively, the value `Ordered` maintains the existing order of the endpoints.

5. Satisfy the proxy's security requirements:
   1. If `Ice.Override.Secure` is defined, remove all non-secure endpoints.
   2. Otherwise, if the proxy is configured to prefer secure endpoints (e.g., by calling the `ice_preferSecure` factory method), move all secure endpoints to the beginning of the list. Note that this setting still allows non-secure endpoints to be included.
   3. Otherwise, move all non-secure endpoints to the beginning of the list.

If connection caching is enabled (see Section 36.3.4) and the Ice run already has a compatible connection (see Section 36.3.3), it reuses the cached connection. Otherwise, the run time attempts to connect to each endpoint in the list until it succeeds or exhausts the list; the order in which endpoints are selected for connection attempts depends on the endpoint selection policy (see page 1862).

36.3.2 Error Handling

If a failure occurs during a connection attempt, the Ice run time tries to connect to all of the proxy's remaining endpoints until either a connection is successfully established or all attempts have failed. At that point, the behavior of the Ice run time depends on the value of the `Ice.RetryIntervals` configuration property (see Appendix D). The default value of this property is `0`, which causes the Ice run time to try connecting to all of the endpoints one more time[1]. If no connection can be established on this second attempt, the Ice run time raises an exception that indicates the reason for the final failed attempt (typically `ConnectFailedException`). Similarly, if a connection was lost during a request and could not be reestablished (assuming the request can be retried), the Ice run time raises an exception that indicates the reason for the final failed attempt.

See Section 32.22 for more information on automatic retries.

36.3.3 Connection Reuse

When establishing a connection for a proxy, the Ice run time reuses an existing connection under the following conditions:

- The remote endpoint matches one of the proxy's endpoints.
- The connection was established by the communicator that created the proxy.

1. Define the property `Ice.Trace.Retry=2` to monitor these attempts.

- The connection matches the proxy's configuration. Timeout values play an important role here, as an existing connection is only reused if its timeout value (i.e., the timeout used when the connection was established) matches the new proxy's timeout (see Section 36.3.5). Similarly, a proxy configured with a connection id only reuses a connection if it was established by a proxy with the same connection id.

Multiple Endpoints

Applications must exercise caution when using proxies containing multiple endpoints, especially endpoints using different transports. For example, suppose a proxy has multiple endpoints, such as one each for TCP, SSL, and UDP. When establishing a connection for this proxy, the Ice run time will open a new connection only if it cannot reuse an existing connection to any of the endpoints (that is, if connection caching is enabled). Furthermore, the proxy in its default (that is, non-secure) configuration gives higher priority to non-secure endpoints. If you want to ensure that a particular transport is used by a proxy, you must configure the proxy appropriately, such as by calling the `ice_secure` or `ice_datagram` factory methods described in Section 32.11.2.

Compression

The Ice run time does not consider compression settings when searching for existing connections to reuse; proxies whose compression settings differ can share the same connection (assuming all other selection criteria are satisfied).

Application Control

The default behavior of the Ice run time, which reuses connections whenever possible, is appropriate for many applications because it conserves resources and typically has little or no impact on performance. However, when a server implementation attaches semantics to a connection, the client often must be designed to cooperate, despite the tighter coupling it causes. For example, a server might use a serialized thread pool (see Section 32.10) to preserve the order of requests received over each connection. If the client wants to execute several requests simultaneously, it must be able to force the Ice run time to establish new connections at will.

For those situations that require more control over connection reuse, the Ice run time allows you to form arbitrary groups of proxies that share a connection by configuring them with the same connection identifier. The factory function `ice_connectionId` (see Section 32.11.2) returns a new proxy configured

with the given connection id. Once configured, the Ice run time ensures that the proxy only reuses a connection that was established by a proxy with the same connection id (assuming all other criteria for connection reuse are also satisfied). A new connection is created if none with a matching id is found, which means each proxy could conceivably have its own connection if each were assigned a unique connection id.

As an example, consider the following code fragment:

```
// C++
Ice::ObjectPrx prx = comm->stringToProxy("ident:tcp -p 10000");
Ice::ObjectPrx g1 = prx->ice_connectionId("group1");
Ice::ObjectPrx g2 = prx->ice_connectionId("group2");
prx->ice_ping(); // Opens a new connection
g1->ice_ping(); // Opens a new connection
g2->ice_ping(); // Opens a new connection
MyInterfacePrx i1 = MyInterfacePrx::checkedCast(g1);
i1->ice_ping(); // Reuses g1's connection
MyInterfacePrx i2 = MyInterfacePrx::checkedCast(
    prx->ice_connectionId("group2"));
i2->ice_ping(); // Reuses g2's connection
```

A total of three connections are established by this example:

1. The proxy `prx` establishes a new connection. This proxy has the default connection id (an empty string).
2. The proxy `g1` establishes a new connection because the only existing connection, the one established by `prx`, has a different connection id.
3. Similarly, the proxy `g2` establishes a new connection because none of the existing connections have a matching connection id.

The proxy `i1` inherits its connection id from `g1`, and therefore shares the connection for `group1`; `i2` explicitly configured its connection id and shares the `group2` connection with proxy `g2`.

36.3.4 Connection Caching

When we refer to a proxy's connection, we actually mean the connection that the proxy is *currently* using. This connection can change over time, such that a proxy might use several connections during its lifetime. For example, an idle connection may be closed automatically and then transparently replaced by a new connection when activity resumes (see Section 36.4).

After establishing a connection in response to proxy activities, the Ice run time adds the connection to an internal pool for subsequent reuse by other proxies (see

Section 36.3.3). The Ice run time manages the lifetime of the connection; it is eventually closed as described in Section 36.6. The connection is not affected by the life cycle of the proxies that use it, except that the lack of activity may prompt the Ice run time to close the connection after a while.

Once a proxy has been associated with a connection, the proxy's default behavior is to continue using that connection for all subsequent requests. In effect, the proxy caches the connection and attempts to use it for as long as possible in order to minimize the overhead of creating new connections. If the connection is later closed and the proxy is used again, the proxy repeats the connection-establishment procedure described in Section 36.3.1.

There are situations in which this default caching behavior is undesirable, such as when a client has a proxy with multiple endpoints and wishes to balance the load among the servers at those endpoints. The client can disable connection caching by passing an argument of `false` to the proxy factory method `ice_connectionCached`. The new proxy returned by this method repeats the connection-establishment procedure outlined in Section 36.3.1 before each request, thereby achieving request load balancing at the expense of potentially higher latency.

This type of load balancing is performed solely by the client using whatever endpoints are contained in the proxy. More sophisticated forms of load balancing are also possible, as described in Chapter 38.

36.3.5 Timeouts

A proxy's default configuration has a timeout value of `-1`, meaning that network activity initiated by this proxy does not time out. The timeout value affects both connection establishment and remote invocations. If a different timeout value is specified and the connection cannot be established within the allotted time, a `ConnectTimeoutException` is raised.

You can set a timeout on a proxy using the `ice_timeout` factory method (see Section 32.11.2). To use the same timeout period for all proxies, you can define the `Ice.Override.Timeout` property; in this case, any timeout established using the `ice_timeout` factory method is ignored. Finally, if you want to specify a separate timeout value that affects only connection establishment and takes precedence over a proxy's configured timeout value, you can define the `Ice.Override.ConnectTimeout` property. (See Appendix D for more information on these properties, and Section 32.13 for more information on invocation timeouts.)

The timeout in effect when a connection is established is bound to that connection and affects all requests on that connection. If a request times out, all other outstanding requests on the same connection also time out, and the connection is closed forcefully (see Section 36.5.5). The Ice run time automatically retries these requests on a new connection, assuming that automatic retries are enabled and would not violate at-most-once semantics (see Section 32.22).

36.4 Active Connection Management

Active Connection Management (*ACM*) is enabled by default and helps to improve scalability and conserve application resources by closing idle connections.

36.4.1 Configuring ACM

ACM is configured separately for client (outgoing) and server (incoming) connections using the properties `Ice.ACM.Client` and `Ice.ACM.Server`, respectively. The default value of `Ice.ACM.Client` is `60`, meaning an outgoing connection is closed if it has not been used for sixty seconds. The default value of `Ice.ACM.Server` is zero, which disables ACM for incoming connections. Ice disables server ACM by default because it can cause incoming oneway requests to be silently discarded, as discussed in Section 32.14.

You can also selectively enable or disable ACM for individual object adapters by setting the property `<adapter>.ACM`.

The decision to close a connection is not based only on a lack of network activity. For example, a request may take longer to complete than the configured idle time. Therefore, ACM does not close a connection if there are outgoing or incoming requests pending on that connection, or if a batch request is being accumulated for that connection.

When it is safe to close the connection, it is done gracefully as described in Section 36.6. The closure is usually transparent to the client and server applications because the connection is reestablished automatically when necessary. We say connection closure is "usually transparent" because it is possible that the Ice run time will be unable to reestablish a connection for a variety of reasons (see Section 36.4.2). In such a situation, the application receives a local exception for new requests (usually a `ConnectFailedException`).

It is important that you choose an idle time that does not result in excessive connection closure and reestablishment. The default value of sixty seconds is a

reasonable default, but your requirements may determine a more appropriate value.

The run time periodically scans through all open connections to close those that have exceeded their idle time. By default, the scanning interval is 10% of the smallest configured ACM timeout, with a minimum of 5 seconds and a maximum of 5 minutes. You can choose a different scanning interval by setting the property `Ice.MonitorConnections` to the desired interval in seconds. Once a connection exceeds its idle time, it is closed by the next scan. This means that, if you set a scanning interval of 120 seconds, and have an ACM timeout of 60 seconds, idle connections will be closed once they have been idle for between one and three minutes. For servers with many (thousands) of incoming connections, you should set the scanning interval to the longest amount of time you can afford connections to remain open past their idle limit; this reduces the overhead of the scans and makes it more likely that each scan will actually find and close idle connections.

36.4.2 Disabling ACM

Since server ACM is disabled by default, you only need to set `Ice.ACM.Client` to 0 to disable ACM for all connections. In this configuration a connection is not closed until its communicator is destroyed or it is closed explicitly by the application (see Section 36.5.5). It is important to note that disabling ACM in a process does not prevent a remote peer from closing a connection; all peers must be properly configured in order to truly disable ACM.

There are certain situations in which it is necessary to disable ACM. For example, oneway requests can be silently discarded when a connection is closed (see Section 32.14). As another example, ACM must be disabled when using bidirectional connections. A bidirectional connection is enabled by using a router such as Glacier2 (see Chapter 42) or by configuring a connection explicitly (see Section 36.7). If you do not disable ACM in such cases, ACM can prematurely close a bidirectional connection and thereby cause callbacks to fail unexpectedly.

36.5 Obtaining a Connection

Applications can gain access to an Ice object representing an established connection.

36.5.1 The `Ice::Connection` Interface

The Slice definition of the `Connection` interface is shown below:

```
module Ice {
    local class ConnectionInfo {
        bool incoming;
        string adapterName;
    };

    local interface Connection {
        void close(bool force);
        Object* createProxy(Identity id);
        void setAdapter(ObjectAdapter adapter);
        ObjectAdapter getAdapter();
        Endpoint getEndpoint();
        void flushBatchRequests();
        string type();
        int timeout();
        string toString();
        ConnectionInfo getInfo();
    };

    local class IPConnectionInfo extends ConnectionInfo {
        string localAddress;
        int localPort;
        string remoteAddress;
        int remotePort;
    };

    local class TCPConnectionInfo extends IPConnectionInfo {};

    local class UDPConnectionInfo extends IPConnectionInfo {
        string mcastAddress;
        int mcastPort;
    };
};

module IceSSL {
    local class ConnectionInfo extends Ice::IPConnectionInfo {
        string cipher;
        Ice::StringSeq certs;
    };
};
```

As indicated in the Slice definition, a connection is a local interface, similar to a communicator or an object adapter. A connection therefore is only usable within the process and cannot be accessed remotely.

The `Connection` interface supports the following operations:

- `void close(bool force)`

  Explicitly closes the connection. The connection is closed gracefully if `force` is false, otherwise the connection is closed forcefully. See Section 36.5.5 for more information.

- `Object* createProxy(Identity id)`

  Creates a special proxy that only uses this connection. See Section 36.7 for more information.

- `void setAdapter(ObjectAdapter adapter)`

  Enables callbacks over this connection. See Section 36.7 for more information.

- `ObjectAdapter getAdapter()`

  Returns the object adapter associated with this connection, or nil if no association has been made.

- `Endpoint getEndpoint()`

  The operation returns an `Endpoint` interface (see Section 36.5.2).

- `void flushBatchRequests()`

  Flushes any pending batch requests for this connection. See page 1285 for more information on this operation.

- `string type()`

  Returns the connection type as a string, such as "`tcp`".

- `int timeout()`

  Returns the timeout value used when the connection was established.

- `string toString()`

  Returns a readable description of the connection.

- `ConnectionInfo getInfo()`

  This operation returns a `ConnectionInfo` class defined as follows:

```
local class ConnectionInfo {
    bool incoming;
    string adapterName;
};
```

The `incoming` member is true if the connection is an incoming connection and false, otherwise. If `incoming` is true, `adapterName` provides the name of the object adapter that accepted the connection.

Note that the actual class that is returned derives from `ConnectionInfo`, depending on the connection type. You can down-cast the returned class instance and access the connection-specific information according to the type of the connection.

Flushing Batch Requests

The `flushBatchRequests` operation blocks the calling thread until any batch requests that are queued for a connection have been successfully written to the local transport. To avoid the risk of blocking, you can also invoke this operation asynchronously using the `begin_flushBatchRequests` method (in those language mappings that support it).

Since batch requests are inherently oneway invocations, the `begin_flushBatchRequests` method does not support a request callback. However, you can use the exception callback to handle any errors that might occur while flushing, and the sent callback to receive notification that the batch request has been flushed successfully.

For example, the code below demonstrates how to flush batch requests asynchronously in C++:

```
class FlushCallback : public IceUtil::Shared
{
public:

    void exception(const Ice::Exception& ex)
    {
        cout << "Flush failed: " << ex << endl;
    }

    void sent(bool sentSynchronously)
    {
        cout << "Batch sent!" << endl;
    }
};
typedef IceUtil::Handle<FlushCallback> FlushCallbackPtr;
```

```
void flushConnection(const Ice::ConnectionPtr& conn)
{
    FlushCallbackPtr f = new FlushCallback;
    Ice::Callback_Connection_flushBatchRequestsPtr cb =
        Ice::newCallback_Connection_flushBatchRequests(
            f, &FlushCallback::exception, &FlushCallback::sent);
    conn->begin_flushBatchRequests(cb);
}
```

For more information on asynchronous invocations, please see the relevant language mapping chapter.

36.5.2 The `Ice::Endpoint` Interface

The `Connection::getEndpoint` operation returns an interface of type `Endpoint`:

```
module Ice {
    const short TCPEndpointType = 1;
    const short UDPEndpointType = 3;

    local class EndpointInfo {
        int timeout;
        bool compress;
        short type();
        bool datagram();
        bool secure();
    };

    local interface Endpoint {
        EndpointInfo getInfo();
        string toString();
    };

    local class IPEndpointInfo extends EndpointInfo {
        string host;
        int port;
    };

    local class TCPEndpointInfo extends IPEndpointInfo {};

    local class UDPEndpointInfo extends IPEndpointInfo {
        byte protocolMajor;
        byte protocolMinor;
        byte encodingMajor;
```

```
        byte encodingMinor;
        string mcastInterface;
        int mcastTtl;
    };

    local class OpaqueEndpointInfo extends EndpointInfo {
        Ice::ByteSeq rawBytes;
    };
};

module IceSSL {
    const short EndpointType = 2;

    local class EndpointInfo extends Ice::IPEndpointInfo {};
};
```

The `getInfo` operation returns an `EndpointInfo` instance. Note that the actual class that is returned derives from `EndpointInfo`, depending on the endpoint type. You can down-cast the returned class instance and access the endpoint-specific information according to the type of the endpoint, as returned by the `type` operation.

The `timeout` member provides the timeout in milliseconds. The `compress` member is true if the endpoint uses compression (if available). The `datagram` operation returns true if the endpoint is for a datagram transport, and the `secure` operation returns true if the endpoint uses SSL.

The derived classes provide further detail about the endpoint according to its type.

Opaque Endpoints

An application may receive a proxy that contains an endpoint whose type is unrecognized by the Ice run time. In this situation, Ice preserves the endpoint in its encoded (*opaque*) form so that the proxy remains intact, but Ice ignores the endpoint for all connection-related activities. Preserving the endpoint allows an application to later forward that proxy with all of its original endpoints to a different program that might support the endpoint type in question.

Although a connection will never return an opaque endpoint, it is possible for a program to encounter an opaque endpoint when iterating over the endpoints returned by the proxy method `ice_getEndpoints`.

As a practical example, consider a program for which the IceSSL plug-in is not configured. If this program receives a proxy containing an SSL endpoint, Ice

treats it as an opaque endpoint such that calling `getInfo` on the endpoint object returns an instance of `OpaqueEndpointInfo`.

Note that the `type` operation of the `OpaqueEndpointInfo` object returns the *actual* type of the endpoint. For example, the operation returns the value `2` if the object encodes an SSL endpoint. As a result, your program cannot assume that an `EndpointInfo` object whose type is `2` can be safely down-cast to `IceSSL::EndpointInfo`; if the IceSSL plug-in is not configured, such a down-cast will fail because the object is actually an instance of `OpaqueEndpointInfo`.

36.5.3 Clients

Clients obtain a connection by calling `ice_getConnection` or `ice_getCachedConnection` on a proxy (see Section 32.11.2). If the proxy does not yet have a connection, the `ice_getConnection` method attempts to establish one. As a result, the caller must be prepared to handle connection failure exceptions as described in Section 36.3.2. Furthermore, if the proxy denotes a collocated object and collocation optimization is enabled, calling `ice_getConnection` results in a `CollocationOptimizationException`.

If you wish to obtain the proxy's connection without the potential for triggering connection establishment, call `ice_getCachedConnection`; this method returns null if the proxy is not currently associated with a connection or if connection caching is disabled for the proxy.

As an example, the C++ code below illustrates how to obtain a connection from a proxy and print its type:

```
Ice::ObjectPrx proxy = ...
try
{
    Ice::ConnectionPtr conn = proxy->ice_getConnection();
    cout << conn->type() << endl;
}
catch(const Ice::CollocationOptimizationException&)
{
    cout << "collocated" << endl;
}
```

36.5.4 Servers

Servers can access a connection via the `con` member of the `Ice::Current` parameter passed to every operation (see Section 32.6). For collocated invocations, `con` has a nil value.

For example, this Java code shows how to invoke `toString` on the connection:

```
public int add(int a, int b, Ice.Current curr)
{
    if (curr.con != null)
    {
        System.out.println("Request received on connection:\n" +
                           curr.con.toString());
    }
    else
    {
        System.out.println("collocated invocation");
    }
    return a + b;
}
```

Although the mapping for the Slice operation `toString` results in a Java method named `_toString`, the Ice run time implements `toString` to return the same value.

36.5.5 Closing a Connection

Applications should rarely need to close a connection explicitly, but those that do must be aware of its implications. Since there are two ways to close a connection, we discuss them separately.

Graceful Closure

Passing an argument of `false` to the `close` operation initiates graceful connection closure, as discussed in Section 36.6. The operation blocks until all pending outgoing requests on the connection have completed.

Forceful Closure

A forceful closure is initiated by passing an argument of `true` to the `close` operation, causing the peer to receive a `ConnectionLostException`.

A client must use caution when forcefully closing a connection. Any outgoing requests that are pending on the connection when `close` is invoked will fail with a `ForcedCloseConnectionException`. Furthermore, requests that fail with this exception are not automatically retried.

In a server context, forceful closure can be useful as a defense against hostile clients.

The Ice run time interprets a `CloseConnectionException` to mean that it is safe to retry the request without violating at-most-once semantics (see Section 32.22). If automatic retries are enabled, a client must only initiate a graceful close when it knows that there are no outgoing requests in progress on that connection, or that any pending requests can be safely retried.

36.6 Connection Closure

The Ice run time may close a connection for many reasons, including the situations listed below:

- When deactivating an object adapter or shutting down a communicator
- As required by active connection management (see Section 36.4)
- When initiated by an application (see Section 36.5.5)
- After a request times out (see Section 36.3.5)
- In response to an exception, such as a socket failure or protocol error.

In most cases, the Ice run time closes a connection gracefully as required by the Ice protocol (see Section 37.3.6). The Ice run time only closes a connection forcefully when a timeout occurs or when the application explicitly requests it.

36.6.1 Graceful Connection Closure

Gracefully closing a connection occurs in stages:

- In the process that initiates closure, incoming and outgoing requests that are in progress are allowed to complete, and then a close connection message is sent to the peer. Any incoming requests received after closure is initiated are silently discarded (but may be retried, as discussed in the next bullet). An attempt to make a new outgoing request on the connection results in a `CloseConnectionException` and an automatic retry (if enabled).
- Upon receipt of a close connection message, the Ice run time in the peer closes its end of the connection. Any outgoing requests still pending on that connection fail with a `CloseConnectionException`. This exception indicates to the Ice run time that it is safe to retry those requests without violating at-most-once semantics (see Section 32.22), assuming automatic retries have not been disabled.

- After detecting that the peer has closed the connection, the initiating Ice run time closes the connection.

36.6.2 Impact on Oneway Invocations

As discussed in Section 32.14, oneway invocations are generally considered reliable because they are sent over a stream-oriented transport. However, it is quite possible for oneway requests to be silently discarded if a server has initiated graceful connection closure (see Section 36.6.1). Whereas graceful closure causes a discarded twoway request to receive a `CloseConnectionException` and eventually be retried, the sender receives no notice about a discarded oneway request.

If an application makes assumptions about the reliability of oneway requests, it may be necessary to control the events surrounding connection closure as much as possible, for example by disabling active connection management (see Section 36.4) and avoiding explicit connection closures.

36.7 Bidirectional Connections

An Ice connection normally allows requests to flow in only one direction. If an application's design requires the server to make callbacks to a client, the server normally establishes a new connection to that client in order to send callback requests, as shown in Figure 36.1.

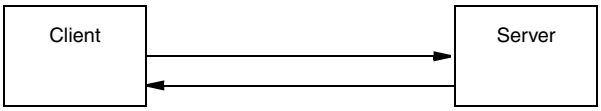


Figure 36.1. Callbacks in an open network.

Unfortunately, network restrictions often prevent a server from being able to create a separate connection to the client, such as when the client resides behind a firewall as shown in Figure 36.2.

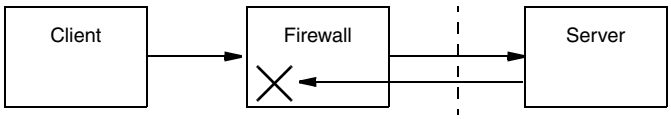


Figure 36.2. Callbacks with a firewall.

In this scenario, the firewall blocks any attempt to establish a connection directly to the client.

For situations such as these, a bidirectional connection offers a solution. Requests may flow in both directions over a bidirectional connection, enabling a server to send callback requests to a client over the client's existing connection to the server.

There are two ways to make use of a bidirectional connection. First, you can use a Glacier2 router, in which case bidirectional connections are used automatically. If you do not require the functionality offered by Glacier2 or you do not want an intermediary service between clients and servers, you can configure bidirectional connections manually.

The remainder of this section discusses manual configuration of bidirectional connections. For more information on using a Glacier2 router, see Chapter 42.

36.7.1 Client Configuration

A client needs to perform the following steps in order to configure a bidirectional connection:

1. Create an object adapter to receive callback requests. This adapter does not require a name or endpoints if its only purpose is to receive callbacks over bidirectional connections.
2. Register the callback object with the object adapter.
3. Activate the object adapter.
4. Obtain the `Ice::Connection` object (see Section 36.5.1) by calling `ice_getConnection` on the proxy.

5. Invoke `setAdapter` on the connection, passing the callback object adapter. This associates an object adapter with the connection and enables callback requests to be dispatched.

6. Pass the identity of the callback object to the server.

The C++ code below illustrates these steps:

```
Ice::ObjectAdapterPtr adapter =
    communicator->createObjectAdapter("");
Ice::Identity ident;
ident.name = IceUtil::generateUUID();
ident.category = "";
CallbackPtr cb = new CallbackI;
adapter->add(cb, ident);
adapter->activate();
proxy->ice_getConnection()->setAdapter(adapter);
proxy->addClient(ident);
```

The last step may seem unusual because a client would typically pass a proxy to the server, not just an identity. For example, you might be tempted to give the proxy returned by the adapter operation `add` to the server, but this would not have the desired effect: if the callback object adapter is configured with endpoints, the server would attempt to establish a separate connection to one of those endpoints, which defeats the purpose of a bidirectional connection. It is just as likely that the callback object adapter has no endpoints, in which case the proxy is of no use to the server.

Similarly, you might try invoking `createProxy` on the connection to obtain a proxy that the server can use for callbacks. This too would fail, because the proxy returned by the connection is for local use only and cannot be used by another process.

As you will see in the next section, the server must create its own callback proxy.

36.7.2 Server Configuration

A server needs to take the following steps in order to make callbacks over a bidirectional connection:

1. Obtain the identity of the callback object, which is typically supplied by the client.

2. Create a proxy for the callback object by calling `createProxy` on the connection. As discussed in Section 36.5.4, the connection object is accessible as a member of the `Ice::Current` parameter to an operation implementation.

These steps are illustrated in the C++ code below:

```
void addClient(const Ice::Identity& ident,
               const Ice::Current& curr)
{
    CallbackPrx client =
        CallbackPrx::uncheckedCast(curr.con->createProxy(ident));
    client->notify();
}
```

36.7.3 Fixed Proxies

The proxy returned by a connection's `createProxy` operation is called a *fixed proxy*. It can only be used in the server process and cannot be marshaled or stringified by `proxyToString`; attempts to do so raise `FixedProxyException`.

The fixed proxy is bound to the connection that created it, and ceases to work once that connection is closed. If the connection is closed prematurely, either by active connection management or by explicit action on the part of the application (see Section 36.4), the server can no longer make callback requests using that proxy. Any attempt to use the proxy again usually results in a `CloseConnectionException`.

Many aspects of a fixed proxy cannot be changed. For example, it is not possible to change the proxy's endpoints or timeout. Attempting to invoke a method such as `ice_timeout` on a fixed proxy raises `FixedProxyException`.

36.7.4 Limitations

Bidirectional connections have certain limitations:

- They can only be configured for connection-oriented transports such as TCP and SSL.
- Most proxy factory methods are not relevant for a proxy created by a connection's `createProxy` operation. The proxy is bound to an existing connection, therefore the proxy reflects the connection's configuration. Attempting to

change settings such as the proxy's timeout value causes the Ice run time to raise `FixedProxyException`.

It is legal to configure a fixed proxy for using oneway or twoway invocations. You may also invoke `ice_secure` on a fixed proxy if its security configuration is important; a fixed proxy configured for secure communication raises `NoEndpointException` on the first invocation if the connection is not secure.

- A connection established from a Glacier2 router to a server is not configured for bidirectional use. Only the connection from a client to the router is bidirectional. However, the client must not attempt to manually configure a bidirectional connection to a router, as this is handled internally by the Ice run time.
- Bidirectional connections are not compatible with active connection management (see Section 36.4).

36.7.5 Threading Considerations

The Ice run time normally creates two thread pools for processing network traffic on connections: the client thread pool manages outgoing connections and the server thread pool manages incoming connections. All of the object adapters in a server share the same thread pool by default, but an object adapter can also be configured to have its own thread pool. The default size of the client and server thread pools is one.

The client thread pool is normally waiting for the replies to pending requests. When a client configures an outgoing connection for bidirectional requests, the client thread pool also becomes responsible for processing requests received over that connection.

Similarly, the server thread pool normally dispatches requests from clients. If a server uses a connection to send callback requests, then the server thread pool must also process the replies to those requests.

You must increase the size of the appropriate thread pool if you need the ability to dispatch multiple requests in parallel, or if you need to make nested twoway invocations. For example, a client that receives a callback request over a bidirectional connection and makes nested invocations must increase the size of the *client* thread pool. See Section 32.10.5 for more information on nested invocations, and Section 32.10 for details on the Ice threading model.

36.7.6 Example

An example that demonstrates how to configure a bidirectional connection is provided in the directory `demo/Ice/bidir`.

36.8 Summary

Most Ice applications benefit from active connection management and transparent connection establishment and thus need not concern themselves with the details of connections. Not all Ice applications can be so fortunate, and for those applications Ice provides convenient access to connections that enables developers to address the realities of today's deployment environments.

Chapter 37
The Ice Protocol

37.1 Chapter Overview

The Ice protocol definition consists of three major parts:

- a set of data encoding rules that determine how the various data types are serialized
- a number of message types that are interchanged between client and server, together with rules as to what message is to be sent under what circumstances
- a set of rules that determine how client and server agree on a particular protocol and encoding version

Section 37.2 describes the encoding rules, Section 37.3 describes the various protocol messages, Section 37.4 describes compression, and Section 37.5 explains how the protocol and encoding are versioned and how client and server agree on a common version. (Both encoding and protocol specifications are currently at version 1.0.)

37.2 Data Encoding

The key goals of the Ice data encoding are simplicity and efficiency. In keeping with these principles, the encoding does not align primitive types on word bound-

aries and therefore eliminates the wasted space and additional complexity that alignment requires. The Ice data encoding simply produces a stream of contiguous bytes; data contains no padding bytes and need not be aligned on word boundaries.

Data is always encoded using little-endian byte order for numeric types. (Most machines use a little-endian byte order, so the Ice data encoding is "right" more often than not.) Ice does not use a "receiver makes it right" scheme because of the additional complexity this would introduce. Consider, for example, a chain of receivers that merely forward data along the chain until that data arrives at an ultimate receiver. (Such topologies are common for event distribution services.) The Ice protocol permits all the intermediates to forward the data without requiring it to be unmarshaled: the intermediates can forward requests by simply copying blocks of binary data. With a "receiver makes it right" scheme, the intermediates would have to unmarshal and remarshal the data whenever the byte order of the next receiver in the chain differs from the byte order of the sender, which is inefficient.

Ice requires clients and servers that run on big-endian machines to incur the extra cost of byte swapping data into little-endian layout, but that cost is insignificant compared to the overall cost of sending or receiving a request.

37.2.1 Sizes

Many of the types involved in the data encoding, as well as several protocol message components, have an associated size or count. A size is a non-negative number. Sizes and counts are encoded in one of two ways:

1. If the number of elements is less than 255, the size is encoded as a single `byte` indicating the number of elements.

2. If the number of elements is greater than or equal to 255, the size is encoded as a `byte` with value `255`, followed by an `int` indicating the number of elements.

Using this encoding to indicate sizes is significantly cheaper than always using an `int` to store the size, especially when marshaling sequences of short strings: counts of up to 254 require only a single byte instead of four. This comes at the expense of counts greater than 254, which require five bytes instead of four. However, for sequences or strings of length greater than 254, the extra byte is insignificant.

37.2.2 Encapsulations

An encapsulation is used to contain variable-length data that an intermediate receiver may not be able to decode, but that the receiver can forward to another recipient for eventual decoding. An encapsulation is encoded as if it were the following structure:

```
struct Encapsulation {
    int size;
    byte major;
    byte minor;
    // [... size - 6 bytes ...]
};
```

The `size` member specifies the size of the encapsulation in bytes (including the `size`, `major`, and `minor` fields). The `major` and `minor` fields specify the encoding version of the data contained in the encapsulation (see Section 37.5.2). The version information is followed by `size-6` bytes of encoded data.

All the data in an encapsulation is context-free, that is, nothing inside an encapsulation can refer to anything outside the encapsulation. This property allows encapsulations to be forwarded among address spaces as a blob of data.

Encapsulations can be nested, that is, contain other encapsulations.

An encapsulations can be empty, in which case its byte count is 6.

37.2.3 Slices

Exceptions and classes are subject to slicing if the receiver of a value only partially understands the received value (that is, only has knowledge of a base type, but not of the actual run-time derived type). To allow the receiver of an exception or class to ignore those parts of a value that it does not understand, exception and class values are marshaled as a sequence of slices (one slice for each level of the inheritance hierarchy). A slice is a byte count encoded as a fixed-length four-byte integer, followed by the data for the slice. (The byte count includes the four bytes occupied by the count itself, so an empty slice has a byte count of four and no data.) The receiver of a value can skip over a slice by reading the byte count b, and then discarding the next b–4 bytes in the input stream.

37.2.4 Basic Types

The basic types are encoded as shown in Table 37.1. Integer types (`short`, `int`, `long`) are represented as two's complement numbers, and floating point types

(`float`, `double`) use the IEEE standard formats [6]. All numeric types use a little-endian byte order.

Table 37.1. Encoding for basic types.

Type	Encoding
`bool`	A single byte with value `1` for `true`, `0` for `false`
`byte`	An uninterpreted byte
`short`	Two bytes (LSB, MSB)
`int`	Four bytes (LSB .. MSB)
`long`	Eight bytes (LSB .. MSB)
`float`	Four bytes (23-bit fractional mantissa, 8-bit exponent, sign bit)
`double`	Eight bytes (52-bit fractional mantissa, 11-bit exponent, sign bit)

37.2.5 Strings

Strings are encoded as a size (see Section 37.2.1), followed by the string contents in UTF-8 format [23]. Strings are not NUL-terminated. An empty string is encoded with a size of zero.

37.2.6 Sequences

Sequences are encoded as a size (see Section 37.2.1) representing the number of elements in the sequence, followed by the elements encoded as specified for their type.

37.2.7 Dictionaries

Dictionaries are encoded as a size (see Section 37.2.1) representing the number of key–value pairs in the dictionary, followed by the pairs. Each key–value pair is encoded as if it were a `struct` containing the key and value as members, in that order.

37.2.8 Enumerators

Enumerated values are encoded depending on the number of enumerators:

- If the enumeration has 1 - 127 enumerators, the value is marshaled as a `byte`.
- If the enumeration has 128 - 32767 members, the value is marshaled as a `short`.
- If the enumeration has more than 32767 members, the value is marshaled as an `int`.

The value is the ordinal value of the corresponding enumerator, with the first enumerator value encoded as zero.

37.2.9 Structures

The members of a structure are encoded in the order they appear in the `struct` declaration, as specified for their types.

37.2.10 Exceptions

Exceptions are marshaled as shown in Figure 37.1

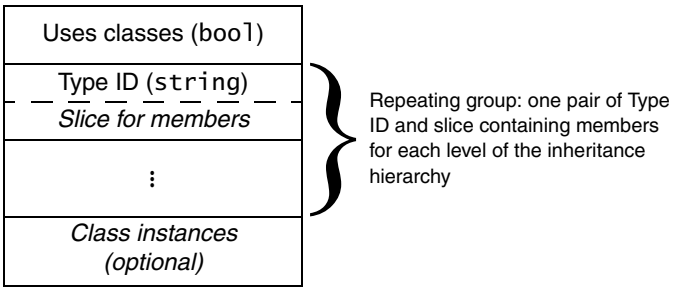


Figure 37.1. Marshaling format for exceptions.

Every exception instance is preceded by a single byte that indicates whether the exception uses class members: the byte value is `1` if any of the exception members are classes (or if any of the exception members, recursively, contain class members) and `0`, otherwise.

Following the header byte, the exception is marshaled as a sequence of pairs: the first member of each pair is the type ID for an exception slice, and the second member of the pair is a slice containing the marshaled members of that slice. The

sequence of pairs is marshaled in derived-to-base order, with the most-derived slice first, and ending with the least-derived slice. Within each slice, data members are marshaled as for structures: in the order in which they are defined in the Slice definition.

Following the sequence of pairs, any class instances that are used by the members of the exception are marshaled. This final part is optional: it is present only if the header byte is `1`. (See Section 37.2.11 for a detailed explanation of how class instances are marshaled.)

To illustrate the marshaling, consider the following exception hierarchy:

```
exception Base {
    int baseInt;
    string baseString;
};

exception Derived extends Base {
    bool derivedBool;
    string derivedString;
    double derivedDouble;
};
```

Assume that the exception members are initialized to the values shown in Table 37.2.

Table 37.2. Member values of an exception of type `Derived`.

Member	Type	Value	Marshaled Size (in bytes)
`baseInt`	`int`	`99`	4
`baseString`	`string`	`"Hello"`	6
`derivedBool`	`bool`	`true`	1
`derivedString`	`string`	`"World!"`	7
`derivedDouble`	`double`	`3.14`	8

From Table 37.2, we can see that the total size of the members of `Base` is 10 bytes, and the total size of the members of `Derived` is 16 bytes. None of the exception members are classes. An instance of this exception has the on-the-wire representa-

tion shown in Table 37.3. (The size, type, and byte offset of the marshaled representation is indicated for each component.)

Table 37.3. Marshaled representation of the exception in Table 37.2.

Marshaled Value	Size in Bytes	Type	Byte offset
`0` *(no class members)*	1	`bool`	0
`"::Derived"` *(type ID)*	10	`string`	1
`20` *(byte count for slice)*	4	`int`	11
`1` *(`derivedBool`)*	1	`bool`	15
`"World!"` *(`derivedString`)*	7	`string`	16
`3.14` *(`derivedDouble`)*	8	`double`	23
`"::Base"` *(type ID)*	7	`string`	31
`14` *(byte count for slice)*	4	`int`	38
`99` *(`baseInt`)*	4	`int`	42
`"Hello"` *(`baseString`)*	6	`string`	46

Note that the size of each string is one larger than the actual string length. This is because each string is preceded by a count of its number of bytes, as explained in Section 37.2.5.

The receiver of this sequence of values uses the header byte to decide whether it eventually must unmarshal any class instances contained in the exception (none in this example) and then examines the first type ID (`::Derived`). If the receiver recognizes that type ID, it can unmarshal the contents of the first slice, followed by the remaining slices; otherwise, the receiver reads the byte count that follows the unknown type (20) and then skips 20–4 bytes in the input stream, which is the start of the type ID for the second slice (`::Base`). If the receiver does not recognize that type ID either, it again reads the byte count following the type ID (14), skips 14–4 bytes, and attempts to read another type ID. (This can happen only if client and server have been compiled with mismatched Slice definitions that disagree in the exception specification of an operation.) In this case, the receiver will eventually encounter an unmarshaling error, which it can report with a `MarshalException`.

If an exception contains class members, these members are marshaled following the exception slices as described in the following section.

37.2.11 Classes

The marshaling for classes is complex, due to the need to deal with the pointer semantics for graphs of classes, as well as the need for the receiver to slice classes of unknown derived type. In addition, the marshaling for classes uses a type ID compression scheme to avoid repeatedly marshaling the same type IDs for large graphs of class instances.

Basic Marshaling Format

Classes are marshaled similar to exceptions: each instance is divided into a number of pairs containing a type ID and a slice (one pair for each level of the inheritance hierarchy) and marshaled in derived-to-base order. Only data members are marshaled—no information is sent that would relate to operations. Unlike exceptions, no header byte precedes a class. Instead, each marshaled class instance is preceded by a (non-zero) positive integer that provides an identity for the instance. The sender assigns this identity during marshaling such that each marshaled instance has a different identity. The receiver uses that identity to correctly reconstruct graphs of classes. The overall marshaling format for classes is shown in Figure 37.2.

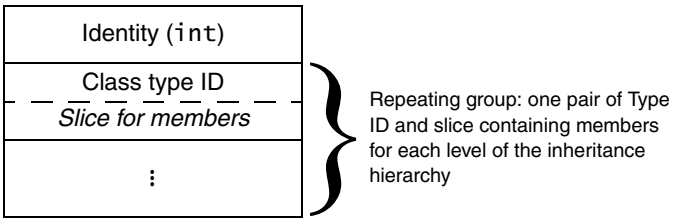


Figure 37.2. Marshaling format for classes.

Class Type IDs

Unlike for exception type IDs, class type IDs are not simple strings. Instead, a class type ID is marshaled as a boolean followed by either a string or a size, to conserve bandwidth. To illustrate this, consider the following class hierarchy:

```
class Base {
    // ...
};

class Derived extends Base {
    // ...
};
```

The type IDs for the class slices are `::Derived` and `::Base`. Suppose the sender marshals three instances of `::Derived` as part of a single request. (For example, two instances could be out-parameters and one instance could be the return value.)

The first instance that is sent on the wire contains the type IDs `::Derived` and `::Base` preceding their respective slices. Because marshaling proceeds in derived-to-base order, the first type ID that is sent is `::Derived`. Every time the sender sends a type ID that it has not sent previously in the same request, it sends the boolean value `false`, followed by the type ID. Internally, the sender also assigns a unique positive number to each type ID. These numbers start at `1` and increment by one for each type ID that has not been marshaled previously. This means that the first type ID is encoded as the boolean value `false`, followed by `::Derived`, and the second type ID is encoded as the boolean value `false`, followed by `::Base`.

When the sender marshals the remaining two instances, it consults a lookup table of previously-marshaled type IDs. Because both type IDs were sent previously in the same request (or reply), the sender encodes all further occurrences of `::Derived` as the value `true` followed by the number `1` encoded as a size (see Section 37.2.1), and it encodes all further occurrences of `::Base` as the value `true` followed by the number `2` encoded as a size.

When the receiver reads a type ID, it first reads its boolean marker:

- If the boolean is `false`, the receiver reads a string and enters that string into a lookup table that maps integers to strings. The first new class type ID received in a request is numbered `1`, the second new class type ID is numbered `2`, and so on.
- If the boolean value is `true`, the receiver reads a number encoded as a size and uses it to retrieve the corresponding class type ID from the lookup table.

Note that this numbering scheme is re-established for each new encapsulation. (As we will see in Section 37.3, parameters, return values, and exceptions are always marshaled inside an enclosing encapsulation.) For subsequent or nested encapsulation, the numbering scheme restarts, with the first new type ID being assigned the value `1`. In other words, each encapsulation uses its own independent

numbering scheme for class type IDs to satisfy the constraint that encapsulations must not depend on their surrounding context.

Encoding class type IDs in this way provides significant savings in bandwidth: whenever an ID is marshaled a second and subsequent time, it is marshaled as a two-byte value (assuming no more than 254 distinct type IDs per request) instead of as a string. Because type IDs can be long, especially if you are using nested modules, the savings are considerable.

Simple Class Marshaling Example

To make the preceding discussion more concrete, consider the following class definitions:

```
interface SomeInterface {
     void op1();
};

class Base {
    int baseInt;
    void op2();
    string baseString;
};

class Derived extends Base implements SomeInterface {
    bool derivedBool;
    string derivedString;
    void op3();
    double derivedDouble;
};
```

Note that `Base` and `Derived` have operations, and that `Derived` also implements the interface `SomeInterface`. Because marshaling of classes is concerned with state, not behavior, the operations `op1`, `op2`, and `op3` are simply ignored during marshaling and the on-the-wire representation is as if the classes had been defined as follows:

```
class Base {
    int baseInt;
    string baseString;
};

class Derived extends Base {
```

```
    bool derivedBool;
    string derivedString;
    double derivedDouble;
};
```

Suppose the sender marshals two instances of `Derived` (for example, as two in-parameters in the same request). The member values are as shown in Table 37.4.

Table 37.4. Member values for two instances of class `Derived`.

	Member	Type	Value	Marshaled Size (in bytes)
First instance	`baseInt`	`int`	`99`	4
	`baseString`	`string`	`"Hello"`	6
	`derivedBool`	`bool`	`true`	1
	`derivedString`	`string`	`"World!"`	7
	`derivedDouble`	`double`	`3.14`	8
Second instance	`baseInt`	`int`	`115`	4
	`baseString`	`string`	`"Cave"`	5
	`derivedBool`	`bool`	`false`	1
	`derivedString`	`string`	`"Canem"`	6
	`derivedDouble`	`double`	`6.32`	8

The sender arbitrarily assigns a non-zero identity (see page 1304) to each instance. Typically, the sender will simply consecutively number the instances starting at `1`. For this example, assume that the two instances have the identities `1`

and `2`. The marshaled representation for the two instances (assuming that they are marshaled immediately following each other) is shown in Table 37.5.

Table 37.5. Marshaled representation of the two instances in Table 37.4.

Marshaled Value	Size in Bytes	Type	Byte offset
`1` *(identity)*	4	`int`	0
`0` *(marker for class type ID)*	1	`bool`	4
`"::Derived"` *(class type ID)*	10	`string`	5
`20` *(byte count for slice)*	4	`int`	15
`1` *(`derivedBool`)*	1	`bool`	19
`"World!"` *(`derivedString`)*	7	`string`	20
`3.14` *(`derivedDouble`)*	8	`double`	27
`0` *(marker for class type ID)*	1	`bool`	35
`"::Base"` *(type ID)*	7	`string`	36
`14` *(byte count for slice)*	4	`int`	43
`99` *(`baseInt`)*	4	`int`	47
`"Hello"` *(`baseString`)*	6	`string`	51
`0` *(marker for class type ID)*	1	`bool`	57
`"::Ice::Object"` *(class type ID)*	14	`string`	58
`5` *(byte count for slice)*	4	`int`	72
`0` *(number of dictionary entries)*	1	`size`	76
`2` *(identity)*	4	`int`	77
`1` *(marker for class type ID)*	1	`bool`	81
`1` *(class type ID)*	1	`size`	82
`19` *(byte count for slice)*	4	`int`	83
`0` *(`derivedBool`)*	1	`bool`	87
`"Canem"` *(`derivedString`)*	6	`string`	88

Table 37.5. Marshaled representation of the two instances in Table 37.4.

Marshaled Value	Size in Bytes	Type	Byte offset
`6.32` *(`derivedDouble`)*	8	`double`	94
`1` *(marker for class type ID)*	1	`bool`	102
`2` *(class type ID)*	1	`size`	103
`13` *(byte count for slice)*	4	`int`	104
`115` *(`baseInt`)*	4	`int`	108
`"Cave"` *(`baseString`)*	5	`string`	112
`1` *(marker for class type ID)*	1	`bool`	117
`3` *(class type ID)*	1	`size`	118
`5` *(byte count for slice)*	4	`int`	119
`0` *(number of dictionary entries)*	1	`size`	123

Note that, because classes (like exceptions) are sent as a sequence of slices, the receiver of a class can slice off any derived parts of a class it does not understand. Also note that (as shown in Table 37.5) each class instance contains three slices. The third slice is for the type `::Ice::Object`, which is the base type of all classes. The class type ID `::Ice::Object` has the number `3` in this example because it is the third distinct type ID that is marshaled by the sender. (See entries at byte offsets 58 and 118 in Table 37.5.) All class instances have this final slice of type `::Ice::Object`.

Marshaling a separate slice for `::Ice::Object` dates back to Ice versions 1.3 and earlier. In those versions, classes carried a facet map that was marshaled as if it were defined as follows:

```
module Ice {
    class Object;

    dictionary<string, Object> FacetMap;

    class Object {
        FacetMap facets; // No longer exists
    };
};
```

As of Ice version 1.4, this facet map is always empty, that is, the count of entries for the dictionary that is marshaled in the `::Ice::Object` slice is always zero. If a receiver receives a class instance with a non-empty facet map, it must throw a `MarshalException`.

Note that if a class has no data members, a type ID and slice for that class is still marshaled. The byte count of the slice will be 4 in this case, indicating that the slice contains no data.

Marshaling Pointers

Classes support pointer semantics, that is, you can construct graphs of classes. It follows that classes can arbitrarily point at each other. The class identity (see page 1304) is used to distinguish instances and pointers as follows:

- A class identity of 0 denotes a null pointer.
- A class identity > 0 precedes the marshaled contents of an instance (see page 1304).
- A class identity < 0 denotes a pointer to an instance.

Identity values less than zero are pointers. For example, if the receiver receives the identity –57, this means that the corresponding class member that is currently being unmarshaled will eventually point at the instance with identity 57.

For structures, classes, exceptions, sequences, and dictionary members that do not contain class members, the Ice protocol uses a simple depth-first traversal algorithm to marshal the members. For example, structure members are marshaled in the order of their Slice definition; if a structure member itself is of complex type, such as a sequence, the sequence is marshaled in toto where it appears inside its enclosing structure. For complex types that contain class members, this depth-first marshaling is suspended: instead of marshaling the actual class instance at this point, a negative identity is marshaled that indicates which class instance that member must eventually denote. For example, consider the following definitions:

```
class C {
    // ...
};

struct S {
    int i;
    C firstC;
    C secondC;
    C thirdC;
    int j;
};
```

Suppose we initialize a structure of type `S` as follows:

```
S myS;
myS.i = 99;
myS.firstC = new C;              // New instance
myS.secondC = 0;                 // null
myS.thirdC = myS.firstC;         // Same instance as previously
myS.j = 100;
```

When this structure is marshaled, the contents of the three class members are not marshaled in-line. Instead, the sender marshals the negative identities of the corresponding instances. Assuming that the sender has assigned the identity 78 to the instance assigned to `myS.firstC`, `myS` is marshaled as shown in Table 37.6.

Table 37.6. Marshaled representation of `myS`.

Marshaled Value	Size in Bytes	Type	Byte offset
`99` *(myS.i)*	4	`int`	0
`-78` *(myS.firstC)*	4	`int`	4
`0` *(myS.secondC)*	4	`int`	8
`-78` *(mys.thirdC)*	4	`int`	12
`100` *(myS.j)*	4	`int`	16

Note that `myS.firstC` and `myS.thirdC` both use the identity –78. This allows the receiver to recognize that `firstC` and `thirdC` point at the same class instance (rather than at two different instances that happen to have the same contents).

Marshaling the negative identities instead of the contents of an instance allows the receiver to accurately reconstruct the class graph that was sent by the sender. However, this begs the question of *when* the actual instances are to be marshaled as described at the beginning of this section. As we will see in Section 37.3, parameters and return values are marshaled as if they were members of a structure. For example, if an operation invocation has five input parameters, the client marshals the five parameters end-to-end as if they were members of a single structure. If any of the five parameters are class instances, or are of complex type (recursively) containing class instances, the sender marshals the parameters in multiple passes: the first pass marshals the parameters end-to-end, using the usual depth-first algorithm:

- If the sender encounters a class member during marshaling, it checks whether it has marshaled the same instance previously for the current request or reply:
  - If the instance has not been marshaled before, the sender assigns a new identity to the instance and marshals the negative identity.
  - Otherwise, if the instance was marshaled previously, the sender sends the same negative identity that is previously sent for that instance.

  In effect, during marshaling, the sender builds an identity table that is indexed by the address of each instance; the lookup value for the instance is its identity.

Once the first pass ends, the sender has marshaled all the parameters, but has not yet marshaled any of the class instances that may be pointed at by various parameters or members. The identity table at this point contains all those instances for which negative identities (pointers) were marshaled, so whatever is in the identity table at this point are the classes that the receiver still needs. The sender now marshals those instances in the identity table, but with positive identities and followed by their contents, as described on page 1306. The outstanding instances are marshaled as a sequence, that is, the sender marshals the number of instances as a size (see Section 37.2.1), followed by the actual instances.

In turn, the instances just sent may themselves contain class members; when those class members are marshaled, the sender assigns an identity to new instances or uses a negative identity for previously marshaled instances as usual. This means that, by the end of the second pass, the identity table may have grown, necessitating a third pass. That third pass again marshals the outstanding class instances as a size followed by the actual instances. The third pass contains all those instances that were not marshaled in the second pass. Of course, the third pass may trigger yet more passes until, finally, the sender has sent all outstanding instances, that is, marshaling is complete. At this point, the sender terminates the sequence of passes by marshaling an empty sequence (the value `0` encoded as a size).

To illustrate this with an example, consider the definitions shown in Section 4.11.7 on page 140 once more:

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {
    idempotent long eval();
};
```

```
class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
    Node operand2;
};

class Operand {
    long val;
};
```

These definitions allow us to construct expression trees. Suppose the client initializes a tree to the shape shown in Figure 37.3, representing the expression $(1 + 6 / 2) * (9 - 3)$. The values outside the nodes are the identities assigned by the client.

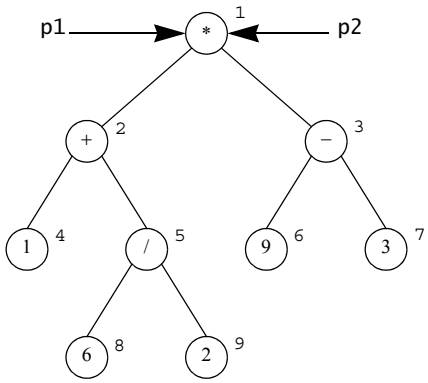


Figure 37.3. Expression tree for the expression $(1 + 6 / 2) * (9 - 3)$. Both `p1` and `p2` denote the root node.

The client passes the root of the tree to the following operation in the parameters `p1` and `p2`, as shown on page 1313. (Even though it does not make sense to pass the same parameter value twice, we do it here for illustration purposes):

```
interface Tree {
    void sendTree(Node p1, Node p2);
};
```

The client now marshals the two parameters `p1` and `p2` to the server, resulting in the value `-1` being sent twice in succession. (The client arbitrarily assigns an identity to each node. The value of the identity does not matter, as long as each node has a unique identity. For simplicity, the Ice implementation numbers instances with a counter that starts counting at `1` and increments by one for each unique instance.) This completes the marshaling of the parameters and results in a single instance with identity 1 in the identity table. The client now marshals a sequence containing a single element, node 1, as described on page 1306. In turn, node 1 results in nodes 2 and 3 being added to the identity table, so the next sequence of nodes contains two elements, nodes 2 and 3. The next sequence of nodes contains nodes 4, 5, 6, and 7, followed by another sequence containing nodes 8 and 9. At this point, no more class instances are outstanding, and the client marshals an empty sequence to indicate to the receiver that the final sequence has been marshaled.

Within each sequence, the order in which class instances are marshaled is irrelevant. For example, the third sequence could equally contain nodes 7, 6, 4, and 5, in that order. What is important here is that each sequence contains nodes that are an equal number of "hops" away from the initial node: the first sequence contains the initial node(s), the second sequence contains all nodes that can be reached by traversing a single link from the initial node(s), the third sequence contains all nodes that can be reached by traversing two links from the initial node(s), and so on.

Now consider the same example once more, but with different parameter values for `sendTree`: `p1` denotes the root of the tree, and `p2` denotes the – operator of the right-hand sub-tree, as shown in Figure 37.4.

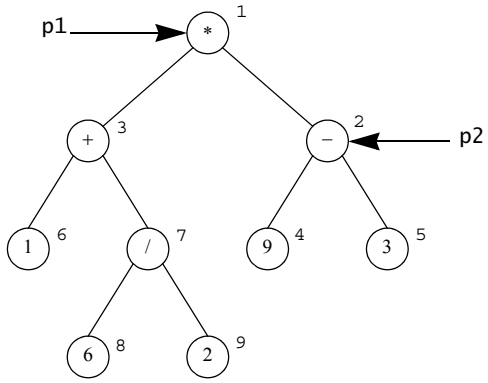


Figure 37.4. The expression tree of Figure 37.3, with `p1` and `p2` denoting different nodes.

The graph that is marshaled is exactly the same, but instances are marshaled in a different order and with different identities:

- During the first pass, the client sends the identities `-1` and `-2` for the parameter values.
- The second pass marshals a sequence containing nodes 1 and 2.
- The third pass marshals a sequence containing nodes 3, 4, and 5.
- The fourth pass marshals a sequence containing nodes 6 and 7.
- The fifth pass marshals a sequence containing nodes 8 and 9.
- The final pass marshals an empty sequence.

In this way, any graph of nodes can be transmitted (including graphs that contain cycles). The receiver reconstructs the graph by filling in a patch table during unmarshaling:

- Whenever the receiver unmarshals a negative identity, it adds that identity to a patch table; the lookup value is the memory address of the parameter or member that eventually will point at the corresponding instance.
- Whenever the receiver unmarshals an actual instance, it adds the instance to an unmarshaled table; the lookup value is the memory address of the instantiated class. The receiver then uses the address of the instance to patch any parameters or members with the actual memory address.

Note that the receiver may receive negative identities that denote class instances that have been unmarshaled already (that is, point "backward" in the unmarshaling stream), as well as instances that are yet to be unmarshaled (that is, point "forward" in the unmarshaling stream). Both scenarios are possible, depending on the order in which instances are marshaled, as well as their in-degree.

To provide another example, consider the following definition:

```
class C {
    // ...
};

sequence<C> CSeq;
```

Suppose the client marshals a sequence of 100 `C` instances to the server, with each instance being distinct. (That is, the sequence contains 100 pointers to 100 different instances, not 100 pointers to the same single instance.) In that case, the sequence is marshaled as a size of 100, followed by 100 negative identities, `-1` to `-100`. Following that, the client marshals a single sequence containing the 100 instances, each instance with its positive identity in the range `1` to `100`, and completes by marshaling an empty sequence.

On the other hand, if the client sends a sequence of 100 elements that all point to the same single class instance, the client marshals the sequence as a size of 100, followed by 100 negative identities, all with the value `-1`. The client then marshals a sequence containing a single element, namely instance `1`, and completes by marshaling an empty sequence.

Class Graphs and Slicing

It is important to note that when a graph of class instances is sent, it always forms a connected graph. However, when the receiver rebuilds the graph, it may end up with a disconnected graph, due to slicing. Consider:

```
class Base {
    // ...
};

class Derived extends Base {
    // ...
    Base b;
};
```

```
interface Example {
    void op(Base p);
};
```

Suppose the client has complete type knowledge, that is, understands both types `Base` and `Derived`, but the server only understands type `Base`, so the derived part of a `Derived` instance is sliced. The client can instantiate classes to be sent as parameter `p` as follows:

```
DerivedPtr p = new Derived;
p->b = new Derived;
ExamplePrx e = ...;
e->op(p);
```

As far as the client is concerned, the graph looks like the one shown in Figure 37.5.

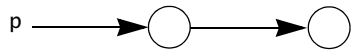


Figure 37.5. Sender-side view of a graph containing derived instances.

However, the server does not understand the derived part of the instances and slices them. Yet, the server unmarshals all the class instances, leading to the situation where the class graph has become disconnected, as shown in Figure 37.6.

Figure 37.6. Receiver-side view of the graph in Figure 37.5.

Of course, more complex situations are possible, such that the receiver ends up with multiple disconnected graphs, each containing many instances.

Exceptions with Class Members

If an exception contains class members, its header byte (see page 1301) is `1` and the exception members are followed by the outstanding class instances as described on the preceding pages, that is, the actual exception members are followed by one or more sequences that contain the outstanding class instances, followed by an empty sequence that serves as an end marker.

37.2.12 Interfaces

Interfaces can be marshaled by value (see Section 4.11.12). For an interface marshaled by value (as opposed to a class instance derived from that interface), only the type ID of the most-derived interface is encoded. Here are the Slice definitions once more:

```
interface Base { /* ... */ };

interface Derived extends Base { /* ... */ };

interface Example {
    void doSomething(Base b);
};
```

If the client passes a class instance to `doSomething` that does not have a Slice definition (but derives from `Derived`), the on-the-wire representation of the interface is as follows:

Table 37.7. Marshaled representation of a `Derived` instance.

Marshaled Value	Size in Bytes	Type	Byte offset
`1` *(identity)*	4	`int`	0
`0` *(marker for class type ID)*	1	`bool`	4
`"::Derived"` *(class type ID)*	10	`string`	5
`4` *(byte count for slice)*	4	`int`	15
`0` *(marker for class type ID)*	1	`bool`	19
`"::Ice::Object"` *(class type ID)*	14	`string`	20
`5` *(byte count for slice)*	4	`int`	34
`0` *(number of dictionary entries)*	1	`size`	38

37.2.13 Proxies

The first component of an encoded proxy is a value of type `Ice::Identity`. If the proxy is a nil value, the `category` and `name` members are empty strings, and no additional data is encoded. The encoding for a non-null proxy consists of general parameters followed by endpoint parameters.

General Proxy Parameters

The general proxy parameters are encoded as if they were members of the following structure:

```
struct ProxyData {
    Ice::Identity id;
    Ice::StringSeq facet;
    byte mode;
    bool secure;
};
```

The general proxy parameters are described in Table 37.8.

Table 37.8. General proxy parameters.

Parameter	Description
`id`	The object identity
`facet`	The facet name (zero- or one-element sequence)
`mode`	The proxy mode (`0`=twoway, `1`=oneway, `2`=batch oneway, `3`=datagram, 4=batch datagram)
`secure`	`true` if secure endpoints are required, otherwise `false`

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a proxy with a `facet` field with more than one element, it must throw a `ProxyUnmarshalException`.

Endpoint Parameters

A proxy optionally contains an endpoint list (see Appendix E) or an adapter identifier, but not both.

- If a proxy contains endpoints, they are encoded immediately following the general parameters. A size specifying the number of endpoints is encoded first (see Section 37.2.1), followed by the endpoints. Each endpoint is encoded as a `short` specifying the endpoint type (`1`=TCP, `2`=SSL, `3`=UDP), followed by an encapsulation (see Section 37.2.2) of type-specific parameters. The type-specific parameters for TCP, UDP, and SSL are presented in the sections that follow.

- If a proxy does not have endpoints, a single byte with value `0` immediately follows the general parameters and a string representing the object adapter identifier is encoded immediately following the zero byte.

Type-specific endpoint parameters are encapsulated because a receiver may not be capable of decoding them. For example, a receiver can only decode SSL endpoint parameters if it is configured with the SSL plug-in (see Chapter 41). However, the receiver must be able to re-encode the proxy with all of its original endpoints, in the order they were received, even if the receiver does not understand the type-specific parameters for an endpoint. Encapsulation of the parameters allows the receiver to do this.

TCP Endpoint Parameters

A TCP endpoint is encoded as an encapsulation containing the following structure:

```
struct TCPEndpointData {
    string host;
    int port;
    int timeout;
    bool compress;
};
```

The endpoint parameters are described in Table 37.9.

Table 37.9. TCP endpoint parameters.

Parameter	Description
`host`	The server host (a host name or IP address)
`port`	The server port (`1-65535`)
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`

See Section 37.4 for more information on compression.

UDP Endpoint Parameters

A UDP endpoint is encoded as an encapsulation containing the following structure:

```
struct UDPEndpointData {
    string host;
    int port;
    byte protocolMajor;
    byte protocolMinor;
    byte encodingMajor;
    byte encodingMinor;
    bool compress;
};
```

The endpoint parameters are described in Table 37.10.

Table 37.10. UDP endpoint parameters.

Parameter	Description
`host`	The server host (a host name or IP address)
`port`	The server port (`1-65535`)
`protocolMajor`	The major protocol version supported by the endpoint
`protocolMinor`	The highest minor protocol version supported by the endpoint
`encodingMajor`	The major encoding version supported by the endpoint
`encodingMinor`	The highest minor encoding version supported by the endpoint
`compress`	`true` if compression should be used (if possible), otherwise `false`

See Section 37.4 for more information on compression.

SSL Endpoint Parameters

An SSL endpoint is encoded as an encapsulation containing the following structure:

```
struct SSLEndpointData {
    string host;
    int port;
    int timeout;
    bool compress;
};
```

The endpoint parameters are described in Table 37.11.

Table 37.11. SSL endpoint parameters.

Parameter	Description
`host`	The server host (a host name or IP address)
`port`	The server port (`1-65535`)
`timeout`	The timeout in milliseconds for socket operations
`compress`	`true` if compression should be used (if possible), otherwise `false`

See Section 37.4 for more information on compression.

37.3 Protocol Messages

The Ice protocol uses five protocol messages:

- Request (from client to server)
- Batch request (from client to server)
- Reply (from server to client)
- Validate connection (from server to client)
- Close connection (client to server or server to client)

Of these messages, validate and close connection only apply to connection-oriented transports.

As with the data encoding described in Section 37.2, protocol messages have no alignment restrictions. Each message consists of a message header and (except for validate and close connection) a message body that immediately follows the header.

37.3.1 Message Header

Each protocol message has a 14-byte header that is encoded as if it were the following structure:

```
struct HeaderData {
    int  magic;
    byte protocolMajor;
    byte protocolMinor;
    byte encodingMajor;
    byte encodingMinor;
    byte messageType;
    byte compressionStatus;
    int  messageSize;
};
```

The members are described in Table 37.12.

Table 37.12. Message header members.

Member	Description
`magic`	A four-byte magic number consisting of the ASCII-encoded values of ‘I’, ‘c’, ‘e’, ‘P’ (0x49, 0x63, 0x65, 0x50)
`protocolMajor`	The protocol major version number
`protocolMinor`	The protocol minor version number
`encodingMajor`	The encoding major version number
`encodingMinor`	The encoding minor version number
`messageType`	The message type
`compressionStatus`	The compression status of the message (see Section 37.4)
`messageSize`	The size of the message in bytes, including the header

Currently, both the protocol and the encoding are at version 1.0. The valid message types are shown in Table 37.13.

Table 37.13. Message types.

Message Type	Encoding
Request	0
Batch request	1
Reply	2
Validate connection	3
Close connection	4

The encoding for these message bodies of each of these message types is described in the sections that follow.

37.3.2 Request Message Body

A request message contains the data necessary to perform an invocation on an object, including the identity of the object, the operation name, and input parameters. A request message is encoded as if it were the following structure:

```
struct RequestData {
    int requestId;
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
    byte mode;
    Ice::Context context;
    Encapsulation params;
};
```

The members are described in Table 37.14.

Table 37.14. Request members.

Member	Description
`requestId`	The request identifier
`id`	The object identity
`facet`	The facet name (zero- or one-element sequence)
`operation`	The operation name
`mode`	A byte representation of `Ice::OperationMode` (`0`=normal, `2`=idempotent)
`context`	The invocation context
`params`	The encapsulated input parameters, in order of declaration

The request identifier zero (`0`) is reserved for use in oneway requests and indicates that the server must not send a reply to the client. A non-zero request identifier must uniquely identify the request on a connection, and must not be reused while a reply for the identifier is outstanding.

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a request with a `facet` field with more than one element, it must throw a `MarshalException`.

37.3.3 Batch Request Message Body

A batch request message contains one or more oneway requests, bundled together for the sake of efficiency. A batch request message is encoded as integer (not a size) that specifies the number of requests in the batch, followed by the corresponding number of requests, encoded as if each request were the following structure:

```
struct BatchRequestData {
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
```

```
    byte mode;
    Ice::Context context;
    Encapsulation params;
};
```

The members are described in Table 37.15.

Table 37.15. Batch request members.

Member	Description
`id`	The object identity
`facet`	The facet name (zero- or one-element sequence)
`operation`	The operation name
`mode`	A byte representation of `Ice::OperationMode`
`context`	The invocation context
`params`	The encapsulated input parameters, in order of declaration

Note that no request ID is necessary for batch requests because only oneway invocations can be batched.

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a batch request with a `facet` field with more than one element, it must throw a `MarshalException`.

37.3.4 Reply Message Body

A reply message body contains the results of a twoway invocation, including any return value, out-parameters, or exception. A reply message body is encoded as if it were the following structure:

```
struct ReplyData {
    int requestId;
    byte replyStatus;
    Encapsulation body; // messageSize - 19 bytes
};
```

The first four bytes of a reply message body contain a request ID. The request ID matches an outgoing request and allows the requester to associate the reply with the original request (see Section 37.3.2).

The byte following the request ID indicates the status of the request; the remainder of the reply message body following the status byte is an encapsulation whose contents depend on the status value. The possible status values are shown in Table 37.16.

Table 37.16. Reply status.

Reply status	Encoding
Success	`0`
User exception	`1`
Object does not exist	`2`
Facet does not exist	`3`
Operation does not exist	`4`
Unknown Ice local exception	`5`
Unknown Ice user exception	`6`
Unknown exception	`7`

Reply Status 0: Success

A successful reply message is encoded as an encapsulation containing out-parameters (in the order of declaration), followed by the return value for the invocation, encoded according to their types as specified in Section 37.2. If an operation declares a `void` return type and no out-parameters, an empty encapsulation is encoded.

Reply Status 1: User exception

A user exception reply message contains an encapsulation containing the user exception, encoded as described in Section 37.2.10.

Reply Status 2: Object does not exist

If the target object does not exist, the reply message is encoded as if it were the following structure inside an encapsulation:

```
struct ReplyData {
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
};
```

The members are described in Table 37.17.

Table 37.17. Invalid object reply members.

Member	Description
`id`	The object identity
`facet`	The facet name (zero- or one-element sequence)
`operation`	The operation name

The `facet` field has either zero elements or one element. An empty sequence denotes the default facet, and a one-element sequence provides the facet name in its first member. If a receiver receives a reply with a `facet` field with more than one element, it must throw a `MarshalException`.

Reply Status 3: Facet does not exist

If the target object does not support the facet encoded in the request message, the reply message is encoded as for reply status 2.

Reply Status 4: Operation does not exist

If the target object does not support the operation encoded in the request message, the reply message is encoded as for reply status 2.

Reply Status 5: Unknown Ice local exception

The reply message for an unknown Ice local exception is encoded as an encapsulation containing a single string that describes the exception.

Reply Status 6: Unknown Ice user exception

The reply message for an unknown Ice user exception is encoded as an encapsulation containing a single string that describes the exception.

Reply Status 7: Unknown exception

The reply message for an unknown exception is encoded as an encapsulation containing a single string that describes the exception.

37.3.5 Validate Connection Message

A server sends a validate connection message when it receives a new connection.[1] The message indicates that the server is ready to receive requests; the client must not send any messages on the connection until it has received the validate connection message from the server. No reply to the message is expected by the server.

The purpose of the validate connection message is two-fold:

- It informs the client of the protocol and encoding versions that are supported by the server (see Section 37.5.3).
- It prevents the client from writing a request message to its local transport buffers until after the server has acknowledged that it can actually process the request. This avoids a race condition caused by the server's TCP/IP stack accepting connections in its backlog while the server is in the process of shutting down: if the client were to send a request in this situation, the request would be lost but the client could not safely re-issue the request because that might violate at-most-once semantics.

  The validate connection message guarantees that a server is not in the middle of shutting down when the server's TCP/IP stack accepts an incoming connection and so avoids the race condition.

The message header described in Section 37.3.1 on page 1323 comprises the entire validate connection message. The compression status of a validate connection message is always `0`.

37.3.6 Close Connection Message

A close connection message is sent when a peer is about to gracefully shutdown a connection.[2] The message header described in Section 37.3.1 comprises the entire close connection message. The compression status of a close connection message is always `0`.

1. Validate connection messages are only used for connection-oriented transports.
2. Close connection messages are only used for connection-oriented transports.

Either client or server can initiate connection closure. On the client side, connection closure is triggered by *Active Connection Management* (*ACM*) (see Section 36.4), which automatically reclaims connections that have been idle for some time.

This means that connection closure can be initiated at will by either end of a connection; most importantly, no state is associated with a connection as far as the object model or application semantics are concerned.

The client side can close a connection whenever no reply for a request is outstanding on the connection. The sequence of events is:

1. The client sends a close connection message.
2. The client closes the writing end of the connection.
3. The server responds to the client's close connection message by closing the connection.

The server side can close a connection whenever no operation invocation is in progress that was invoked via that connection. This guarantees that the server will not violate at-most-once semantics: an operation, once invoked in a servant, is allowed to complete and its results are returned to the client. Note that the server can close a connection even after it has received a request from the client, provided that the request has not yet been passed to a servant. In other words, if the server decides that it wants to close a connection, the sequence of events is:

1. The server discards all incoming requests on the connection.
2. The server waits until all still executing requests have completed and their results have been returned to the client.
3. The server sends a close connection message to the client.
4. The server closes its writing end of the connection.
5. The client responds to the server's close connection message by closing both its reading and writing ends of the connection.
6. If the client has outstanding requests at the time it receives the close connection message, it re-issues these requests on a new connection. Doing so is guaranteed not to violate at-most-once semantics because the server guarantees not to close a connection while requests are still in progress on the server side.

37.3.7 Protocol State Machine

From a client's perspective, the Ice protocol behaves according to the state machine shown in Figure 37.7.

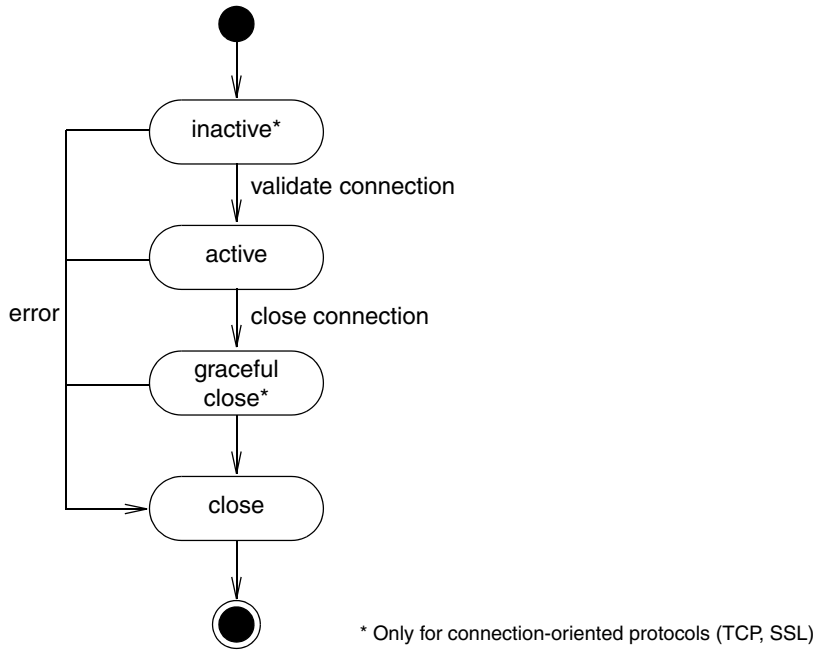


Figure 37.7. Protocol state machine.

To summarize, a new connection is inactive until a validate connection message (see Section 37.3.5) has been received by the client, at which point the active state is entered. The connection remains in the active state until it is shut down, which can occur when there are no more proxies using the connection, or after the connection has been idle for a while. At this point, the connection is gracefully closed, meaning that a close connection message is sent (see Section 37.3.6), and the connection is closed.

37.3.8 Disorderly Connection Closure

Any violation of the protocol or encoding rules results in a disorderly connection closure: the side of the connection that detects a violation unceremoniously closes it (without sending a close connection message or similar). There are many poten-

tial error conditions that can lead to disorderly connection closure; for example, the receiver might detect that a message has a bad magic number or incompatible version, receive a reply with an ID that does not match that of an outstanding request, receive a validate connection message when it should not, or find illegal data in a request (such as a negative size, or a size that disagrees with the actual data that was unmarshaled).

37.4 Compression

Compression is an optional feature of the Ice protocol; whether it is used for a particular message is determined by several factors:

1. Compression may not be supported on all platforms or in all language mappings.
2. Compression can be used in a request or batch request only if the endpoint advertises the ability to accept compressed messages (see Section 37.2.13).
3. For efficiency reasons, the Ice protocol engine does not compress messages smaller than 100 bytes.[3]

If compression is used, the entire message excluding the header is compressed using the bzip2 algorithm [16]. The `messageSize` member of the message header therefore reflects the size of the compressed message, including the uncompressed header, plus an additional four bytes (see page 1334).

3. A compliant implementation of the protocol is free to compress messages that are smaller than 100 bytes—the choice is up to the protocol implementation.

The `compressionStatus` field of the message header (see Section 37.3.1) indicates whether a message is compressed and whether the sender can accept a compressed reply, as shown in Table 37.18.

Table 37.18. Compression status values.

Reply status	Encoding	Applies to
Message is uncompressed, sender cannot accept a compressed reply.	`0`	Request, Batch Request, Reply, Validate Connection, Close Connection
Message is uncompressed, sender can accept a compressed reply.	`1`	Request, Batch Request
Message is compressed and sender can accept a compressed reply	`2`	Request, Batch Request, Reply

- A compression status of `0` indicates that the message is not compressed and, moreover, that the sender of this message cannot accept a compressed reply. A client that does not support compression always uses this value. A client that supports compression sets the value to `0` if the endpoint via which the request is dispatched indicates that it does not support compression.

  A server uses this value for uncompressed replies.
- A compression status of `1` indicates that the message is not compressed, but that the server should return a compressed reply (if any). A client uses this value if the endpoint via which the request is dispatched indicates that it supports compression, but the client has decided not to use compression for this particular request (presumably because the request is too small, so compression does not provide any saving).

  This value applies only to request and batch request messages.
- A compression status of `2` indicates that the message is compressed and that the server is free to reply with a compressed message (but need not reply with a compressed message). A client that supports compression (obviously) sets this value only if the endpoint via which the request is dispatched indicates that it supports compression.

  A server uses this value for compressed replies.

The message body of a compressed request, batch request, or reply message is encoded by first writing the size of the uncompressed message (including its

header) as four-byte integer, followed by the compressed message body (excluding the header). It follows that the size of a compressed message is 14 bytes for the header, plus four bytes to record the size of the uncompressed message, plus the number of bytes occupied by the compressed message body (encoded as specified in Sections 37.3.2 through 37.3.4). Writing the uncompressed message size prior to the body enables the receiver to allocate a buffer that is large enough to accomodate the uncompressed message body.

Note that compression is likely to improve performance only over lower-speed links, for which bandwidth is the overall limiting factor. Over high-speed LAN links, the CPU time spent on compressing and uncompressing messages is longer than the time it takes to just send the uncompressed data.

37.5 Protocol and Encoding Versions

As we saw in the preceding sections, both the Ice protocol and encoding have separate major and minor version numbers. Separate versioning of protocol and encoding has the advantage that neither depends on the other: any version of the Ice protocol can be used with any version of the encoding, so they can evolve independently. (For example, Ice protocol version 1.1 could use encoding version 2.3, and vice versa.)

The Ice versioning mechanism provides the maximum possible amount of interoperability between clients and servers that use different versions of the Ice run time. In particular, older deployed clients can communicate with newer deployed servers and vice versa, provided that the message contents use types that are understandable to both sides.

For an example, assume that a later version of Ice were to introduce a new Slice keyword and data type, such as `complex`, for complex numbers. This would require a new minor version number for the encoding; let us assume that version 1.1 of the encoding is identical to the 1.0 encoding but, in addition,

supports the `complex` type. We now have four possible combinations of client and server encoding versions:

Table 37.19. Interoperability for different versions.

Client Version	Server Version	Operation with `complex` Parameter	Operation without `complex` Parameter
1.0	1.0	N/A	✔
1.1	1.0	N/A	✔
1.0	1.1	N/A	✔
1.1	1.1	✔	✔

As you can see, interoperability is provided to the maximum extent possible. If both client and server are at version 1.1, they can obviously exchange messages and will use encoding version 1.1. For version 1.0 clients and servers, obviously only operations that do not involve `complex` parameters can be invoked (because at least one of client and server do not know about the new `complex` type) and messages are exchanged using encoding version 1.0.

37.5.1 Version Ground Rules

For versioning of the protocol and encoding to be possible, *all* versions (present and future) of the Ice run time adhere to a few ground rules:

1. Encapsulations always have a six-byte header; the first four bytes are the size of the encapsulation (including the size of the header), followed by two bytes that indicate the major and minor version. How to interpret the remainder of the encapsulation depends on the major and minor version.
2. The first eight bytes of a message header always contain the magic number ‘I’, ‘c’, ‘e’, ‘P’, followed by four bytes of version information (two bytes for the protocol major and minor number, and two bytes of the encoding major and minor number). How to interpret the remainder of the header and the message body depends on the major and minor version.

These ground rules ensure that all current and future versions of the Ice run time can at least identify the version and size of an encapsulation and a message. This is particularly important for message switches such as IceStorm (see Chapter 44);

by keeping the version and size information in a fixed format, it is possible to forward messages that are, for example, at version 2.0, even though the message switch itself may still be at version 1.0.

37.5.2 Version Compatibility Rules

To establish whether a particular protocol version is compatible with another protocol version (or a particular encoding version is compatible with another encoding version), the following rules apply:

1. Different major versions are incompatible. There is no obligation on either clients or servers to support more than a single major version. For example, a server with major version 2 is under no obligation to also support major version 1.

   This rule exists to permit the Ice run time to eventually get rid of old versions—without such a rule, all future releases of Ice would have to support all previous major versions forever. In plain language, the rule means that clients and servers that use different major versions simply cannot communicate with each other.

2. A receiver that advertises minor version n guarantees to be able to successfully decode all minor versions less than n. Note that this does *not* imply that messages using version $n-1$ can be decoded as if they were version n: as far as their physical representation is concerned, two adjacent minor versions can be completely incompatible. However, because any receiver advertising version n is also obliged to correctly deal with version $n-1$, minor version upgrades are *semantically* backward compatible, even though their physical representation may be incompatible.

3. A sender that supports minor version n guarantees to be able to send messages using all minor versions less than n. Moreover, the sender guarantees that if it receives a request using minor version k (with $k \leq n$), it will send the reply for that request using minor version k.

37.5.3 Version Negotiation

Client and server must somehow agree on which version to use to exchange messages. Depending on whether the underlying transport is connection-oriented or connection-less, different mechanisms are used to negotiate a common version.

Negotiation for Connection-Oriented Transports

For connection-oriented transports, the client opens a connection to the server and then waits for a validate connection message (see page 1329). The validate connection message sent by the server indicates the server's major and highest supported minor version numbers for both protocol and encoding. If the server's and client's major version numbers do not match, the client side raises an `UnsupportedProtocolException` or `UnsupportedEncodingException`.

Assuming that the client has received a validate connection message from the server that matches the client's major version, the client knows the highest minor version number that is supported by the server. Thereafter, the client is obliged to send no message with a minor version number higher than the server's limit. However, the client is free to send a message with a minor version number that is less than the server's limit.

The server does not have a-priori knowledge of the highest minor version that is supported by the client (because there is no validate connection message from client to server). Instead, the server learns about the client version number in each individual message, by looking at the message header. That minor version indicates the minor version number that the client can accept. The scope of that minor version number is a single request-reply interaction. For example, if the client sends a request with minor version 3, the server must reply to that request with minor version 3 as well. However, the next client request might be with minor version 2, and the server must reply to that request with minor version 2.

For orderly connection closure via a close connection message, the server can use any minor version, but that minor version must not be higher than the highest minor version number that was received from the client while the connection was open.

Negotiation for Connection-Less Transports

For connection-less transports, no validate connection message exists, so the client must learn about the highest supported minor version number of the server via other means. The mechanism for this depends on whether a proxy for a connection-less endpoint is bound directly or indirectly (see page 11):

- For direct proxies, the version information is part of the endpoint contained in the proxy. In this case, the client simply sends its messages with a minor version number that is not greater than the minor version number of the endpoint in the proxy.
- For indirect proxies, the proxy itself contains no version information at all (because the proxy contains no endpoints). Instead, the client obtains the

version information when it resolves the proxy's symbolic information to one or more endpoints (via IceGrid or an equivalent service). The version information of the endpoints determines the highest minor version number that is available to the client.

Part XI

Ice Services

Chapter 38
IceGrid

38.1 Chapter Overview

In this chapter we present IceGrid, an important service for building robust Ice applications. Section 38.3 introduces the IceGrid architecture and its core concepts. Section 38.4 describes a sample IceGrid application that we improve incrementally in Section 38.5. The subject of Section 38.6 is well-known objects, which are a convenient way to decouple clients and servers.

Core IceGrid features such as templates, IceBox integration, object adapter replication, load balancing, resource allocation, and registry replication are covered in individual sections from Section 38.7 through Section 38.12.

In Section 38.13 we learn how IceGrid automates the distribution of server executables. Using administrative sessions is the subject of Section 38.14, while Section 38.15 gives instructions on integrating a Glacier2 router into an application. Section 38.16 discusses how to use an SQL database to store IceGrid's persistent state. Section 38.17 is a reference for the descriptors that define an IceGrid application. Section 38.18 describes the semantics of variables and parameters in descriptors, and property sets are discussed in Section 38.19.

The convenient features that IceGrid supports for XML files are the focus of Section 38.20. Section 38.21 through Section 38.24 address the usage and administration of IceGrid servers. Section 38.25 explains the details of server activation, while Section 38.26 provides troubleshooting advice.

38.2 Introduction

IceGrid is the location and activation service for Ice applications. In prior Ice releases, the IcePack service supplied this functionality. Given the increasing importance of grid computing, IceGrid was introduced in Ice 3.0 to support all of IcePack's responsibilities and extend it with new features to simplify the development and administration of Ice applications on grid networks.

For the purposes of this chapter, we can loosely define *grid computing* as the use of a network of relatively inexpensive computers to perform the computational tasks that once required costly "big iron." Developers familiar with distributed computing technologies may not consider the notion of grid computing to be particularly revolutionary; after all, distributed applications have been running on networks for years, and the definition of grid computing is sufficiently vague that practically any server environment could be considered a "grid."

One possible grid configuration is a homogeneous collection of computers running identical programs. Each computer in the grid is essentially a clone of the others, and all are equally capable of handling a task. As a developer, you need to write the code that runs on the grid computers, and Ice is ideally suited as the infrastructure that enables the components of a grid application to communicate with one another. However, writing the application code is just the first piece of the puzzle. Many other challenges remain:

- How do I install and update this application on all of the computers in the grid?
- How do I keep track of the servers running on the grid?
- How do I distribute the load across all the computers?
- How do I migrate a server from one computer to another one?
- How can I quickly add a new computer to the grid?

Of course, these are issues faced by most distributed applications. As you read this chapter and learn more about IceGrid's capabilities, you will discover that it offers solutions to these challenges. To get you started, we have summarized IceGrid's feature set below:

- Location service

  As an implementation of an Ice location service (see Section 32.17), IceGrid enables clients to bind indirectly to their servers, making applications more flexible and resilient to changing requirements.

- On-demand server activation

  Starting an Ice server process is called *server activation*. IceGrid can be given responsibility for activating a server on demand, that is, when a client attempts to access an object hosted by the server. Activation usually occurs as a side effect of indirect binding, and is completely transparent to the client.

- Application distribution

  IceGrid provides a convenient way to distribute your application to a set of computers, without the need for a shared file system or complicated scripts. Simply configure an IcePatch2 server (see Chapter 45) and let IceGrid download the necessary files and keep them synchronized.

- Replication and load balancing

  IceGrid supports replication by grouping the object adapters of several servers into a single virtual object adapter. During indirect binding, a client can be bound to an endpoint of any of these adapters. Furthermore, IceGrid monitors the load on each computer and can use that information to decide which of the endpoints to return to a client.

- Sessions and resource allocation

  An IceGrid client establishes a session in order to allocate a resource such as an object or a server. IceGrid prevents other clients from using the resource until the client releases it or the session expires. Sessions enhance security through the use of an authentication mechanism that can be integrated with a Glacier2 router.

- Automatic failover

  Ice supports automatic retry and failover in any proxy that contains multiple endpoints. When combined with IceGrid's support for replication and load balancing, automatic failover means that a failed request results in a client transparently retrying the request on the next endpoint with the lowest load.

- Dynamic queries

  In addition to transparent binding, applications can interact directly with IceGrid to locate objects in a variety of ways.

- Status monitoring

  IceGrid supports Slice interfaces that allow applications to monitor its activities and receive notifications about significant events, enabling the development of custom tools or the integration of IceGrid status events into an existing management framework.

- Administration

  IceGrid includes command-line and graphical administration tools. They are available on all supported platforms and allow you to start, stop, monitor, and reconfigure any server managed by IceGrid.

- Deployment

  Using XML files, you can describe the servers to be deployed on each computer. Templates simplify the description of identical servers.

- Database Indepence

  By default, IceGrid uses a Freeze database (Chapter 39) to store its state. However, you can configure IceGrid to use a different dabase, such as MySQL (among others).

As grid computing enters the mainstream and compute servers become commodities, users expect more value from their applications. IceGrid, in cooperation with the Ice run time, relieves you of these low-level tasks to accelerate the construction and simplify the administration of your distributed applications.

38.3 IceGrid Architecture

An IceGrid domain consists of a *registry* and any number of *nodes*. Together, the registry and nodes cooperate to manage the information and server processes that comprise *applications*. Each application assigns servers to particular nodes. The registry maintains a persistent record of this information, while the nodes are responsible for starting and monitoring their assigned server processes. In a typical configuration, one node runs on each computer that hosts Ice servers. The registry does not consume much processor time, so it commonly runs on the same computer as a node; in fact, the registry and a node can run in the same process if desired. If fault tolerance is desired, the registry supports replication using a master-slave design.

38.3.1 Simple Example

As an example, Figure 38.1 shows a very simple IceGrid application running on a network of three computers. The IceGrid registry is the only process of interest on

host `PC1`, while IceGrid nodes are running on the hosts `PC2` and `PC3`. In this sample application, one server has been assigned to each node.

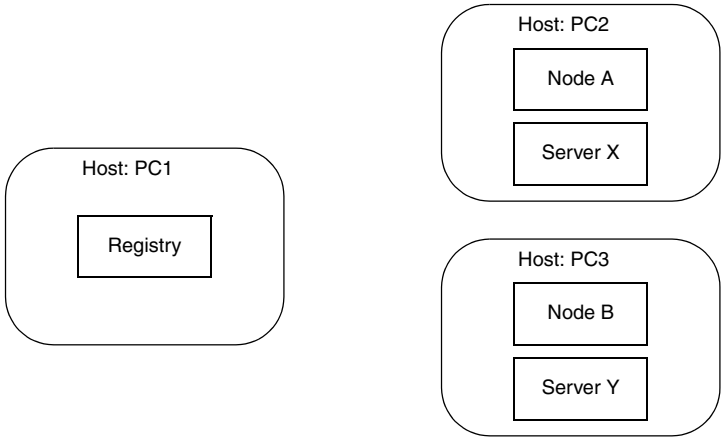


Figure 38.1. Simple IceGrid application.

From a client application's perspective, the primary responsibility of the registry is to resolve indirect proxies as an Ice location service (see Section 32.17). As such, this contribution is largely transparent: when a client first attempts to use an indirect proxy, the Ice run time in the client contacts the registry to convert the proxy's symbolic information into endpoints that allow the client to establish a connection.

Although the registry might sound like nothing more than a simple lookup table, reality is quite different. For example, behind the scenes, a locate request might prompt a node to start the target server automatically, or the registry might select appropriate endpoints based on load statistics from each computer.

This also illustrates the benefits of indirect proxies: the location service can provide a great deal of functionality without any special action by the client and, unlike with direct proxies, the client does not need advance knowledge of the address and port of a server. The extra level of indirection adds some latency to the client's first use of a proxy; however, all subsequent interactions occur directly between client and server, so the cost is negligible. Furthermore, indirection allows servers to migrate to different computers without the need to update proxies held by clients.

38.3.2 Server Replication

IceGrid's flexibility allows an endless variety of configurations. For example, suppose we have a grid network and want to replicate a server on each blade, as shown in Figure 38.2.

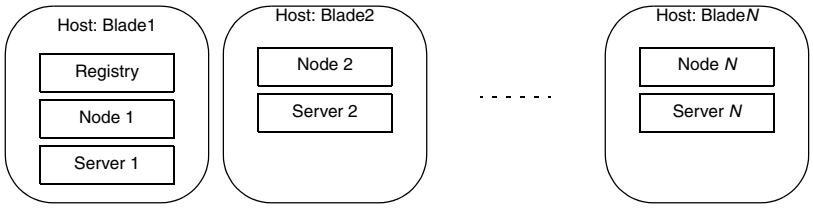


Figure 38.2. Replicated server on grid network.

Replication in Ice is based on object adapters, not servers. Any object adapter in any server could participate in replication, but it is far more likely that all of the replicated object adapters are created by instances of the same server executable that is running on each computer. We are using this configuration in the example shown above, but IceGrid requires each server to have a unique name. `Server 1` and `Server 2` are our unique names for the same executable.

The binding process works somewhat differently when replication is involved, since the registry now has multiple object adapters to choose from. The description of the IceGrid application drives the registry's decision about which object adapter (or object adapters) to use. For example, the registry could consider the system load of each computer (as periodically reported by the nodes) and return the endpoints of the object adapter on the computer with the lowest load. It is also possible for the registry to combine the endpoints of several object adapters, in which case the Ice run time in the client would select the endpoint for the initial connection attempt.

38.3.3 Deployment

In IceGrid, *deployment* is the process of describing an application to the registry. This description includes the following information:

- Replica groups

  A *replica group* is the term for a collection of replicated object adapters. An application can create any number of replica groups. Each group requires a unique identifier.
- Nodes

  An application must assign its servers to one or more nodes.
- Servers

  A server's description includes a unique name and the path to its executable. It also lists the object adapters it creates.
- Object adapters

  Information about an object adapter includes its endpoints and any well-known objects it advertises. If the object adapter is a member of a replica group, it must also supply that group's identifier.
- Objects

  A *well-known object* is one that is known solely by its identity. The registry maintains a global list of such objects for use during locate requests.

IceGrid uses the term *descriptor* to refer to the description of an application and its components; deploying an application involves creating its descriptors in the registry. The are several ways to accomplish this:

- You can use a command-line tool that reads XML descriptors.
- You can create descriptors interactively with the graphical administration tool.
- You can create descriptors programmatically via IceGrid's administrative interface.

The registry server must be running in order to deploy an application, but it is not necessary for nodes to be active. Nodes that are started after deployment automatically retrieve the information they need from the registry. Once deployed, you can update the application at any time.

38.4 Getting Started

This section introduces a sample application that will help us demonstrate IceGrid's capabilities. Our application "rips" music tracks from a compact disc (CD) and encodes them as MP3 files, as shown in Figure 38.3.

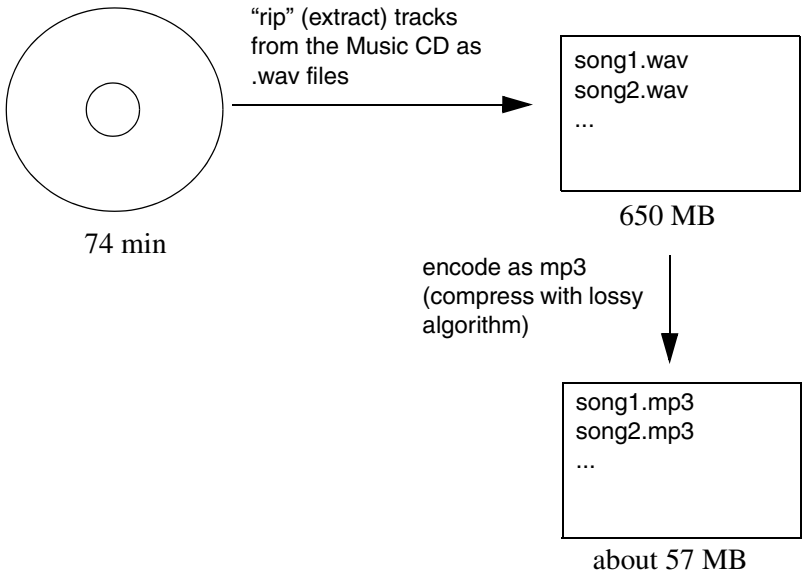


Figure 38.3. Overview of sample application.

Ripping an entire CD usually takes several minutes because the MP3 encoding requires lots of CPU cycles. Our distributed ripper application accelerates this process by taking advantage of powerful CPUs on remote Ice servers, enabling us to process many songs in parallel.

The Slice interface for the MP3 encoder is straightforward:

```
module Ripper {
exception EncodingFailedException {
    string reason;
};

sequence<short> Samples;
```

```
interface Mp3Encoder {
    // Input: PCM samples for left and right channels
    // Output: MP3 frame(s).
    Ice::ByteSeq encode(Samples leftSamples, Samples rightSamples)
        throws EncodingFailedException;

    // You must flush to get the last frame(s). Flush also
    // destroys the encoder object.
    Ice::ByteSeq flush()
        throws EncodingFailedException;
};

interface Mp3EncoderFactory
{
    Mp3Encoder* createEncoder();
};
};
```

The implementation of the encoding algorithm is not relevant for the purposes of this discussion. Instead, we will focus on incrementally improving the application as we discuss IceGrid features.

38.4.1 Architecture

The initial architecture for our application is intentionally simple, consisting of an IceGrid registry and a server that we start manually. Figure 38.4 shows how the client's invocation on its `EncoderFactory` proxy causes an implicit locate request.

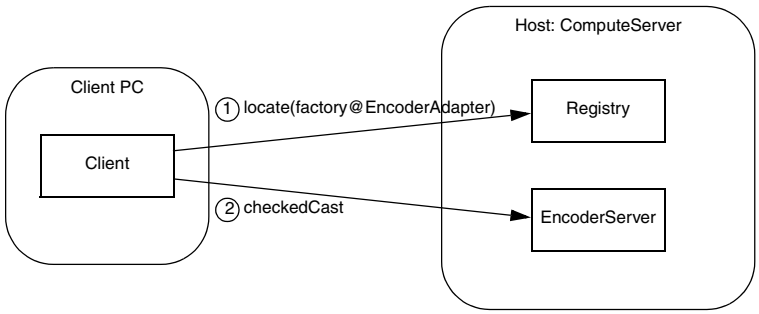


Figure 38.4. Initial architecture for the ripper application.

The corresponding C++ code for the client is presented below:

```
Ice::ObjectPrx proxy =
    communicator->stringToProxy("factory@EncoderAdapter");
Ripper::MP3EncoderFactoryPrx factory =
    Ripper::MP3EncoderFactoryPrx::checkedCast(proxy);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

Notice that the client uses an indirect proxy for the `MP3EncoderFactory` object. This stringified proxy can be read literally as "the object with identity `factory` in the object adapter identified as `EncoderAdapter`." The encoding server creates this object adapter and ensures that the object adapter uses this identifier. Since each object adapter must be uniquely identified, the registry can easily determine the server that created the adapter and return an appropriate endpoint to the client.

The client's call to `checkedCast` is the first remote invocation on the factory object, and therefore the locate request is performed during the completion of this invocation. The subsequent call to `createEncoder` is sent directly to the server without further involvement by IceGrid.

38.4.2 Configuring the Registry

The registry needs a subdirectory in which to create its databases, and we will use `/opt/ripper/registry` for this purpose (the directory must exist before starting the registry). We also need to create an Ice configuration file to hold properties required by the registry. The file `/opt/ripper/registry.cfg` contains the following properties:

```
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsV
erifier
IceGrid.Registry.Data=/opt/ripper/registry
IceGrid.Registry.DynamicRegistration=1
```

Several of the properties define endpoints, but only the value of `IceGrid.Registry.Client.Endpoints` needs a fixed port. This property specifies the endpoints of the IceGrid locator service; IceGrid clients must include these endpoints in their definition of `Ice.Default.Locator`, as discussed in the next section. The TCP port number (4061) used in this example has been reserved by the Internet Assigned Numbers Authority (IANA) for the IceGrid registry, along with SSL port number 4062.

Several other properties are worth mentioning:

- `IceGrid.Registry.AdminPermissionsVerifier`

  This property controls access to the registry's administrative functionality (see Section 38.11.2).

- `IceGrid.Registry.Data`

  This property specifies the registry's database directory.

- `IceGrid.Registry.DynamicRegistration`

  By setting this property to a non-zero value, we allow servers to register their object adapters. Dynamic registration is explained in more detail below.

Dynamic Registration

By default, IceGrid will not permit a server to register its object adapters without using IceGrid's deployment facility (see Section 38.5). In some situations, such as in this sample application, you may want a client to be able to bind indirectly to a server without having to first deploy the server. That is, simply starting the server should be sufficient to make the server register itself with IceGrid and be reachable from clients.

You can achieve this by running the registry with the property `IceGrid.Registry.DynamicRegistration` set to a non-zero value. With this setting, IceGrid permits an adapter to register itself upon activation even if it has not been previously deployed. To force the server to register its adapters, you must define `Ice.Default.Locator` (so the server can find the registry) and, for each adapter that you wish to register, you must set *`<adapter-name>`*`.AdapterId` to an identifier that is unique within the registry. Setting the *`<adapter-name>`*`.AdapterId` property also causes the adapter to no longer create direct proxies but rather to create indirect proxies that must be resolved by the registry.

38.4.3 Configuring the Client

The client requires only minimal configuration, namely a value for the property `Ice.Default.Locator` (see Section 32.17.3). This property supplies the Ice run time with the proxy for the locator service. In IceGrid, the locator service is implemented by the registry, and the locator object is available on the registry's client endpoints. The property `IceGrid.Registry.Client.Endpoints` defined in the previous section provides most of the information we need to construct the proxy. The missing piece is the identity of the locator object, which defaults to `IceGrid/Locator`:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost -p 4061
```

The use of a locator service allows the client to take advantage of indirect binding and avoid static dependencies on server endpoints. However, the locator proxy must have a fixed port, otherwise the client has a bootstrapping problem: it cannot resolve indirect proxies without knowing the endpoints of the locator service.

See Section 38.21.3 for more information on client configuration.

38.4.4 Configuring the Server

We use `/opt/ripper/server.cfg` as the server's configuration file. It contains the following properties:

```
EncoderAdapter.AdapterId=EncoderAdapter
EncoderAdapter.Endpoints=tcp
Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost -p 4061
```

The properties are described below:

- `EncoderAdapter.AdapterId`

  This property supplies the object adapter identifier that the client uses in its indirect proxy (e.g., `factory@EncoderAdapter`).

- `EncoderAdapter.Endpoints`

  This property defines the object adapter's endpoint. Notice that the value does not contain any port information, meaning that the adapter uses a system-assigned port. Without IceGrid, the use of a system-assigned port would pose a significant problem: how would a client create a direct proxy if the adapter's port could change every time the server is restarted? IceGrid solves this problem nicely because clients can use indirect proxies that contain no endpoint dependencies. The registry resolves indirect proxies using the endpoint information supplied by object adapters each time they are activated.

- `Ice.Default.Locator`

  The server requires a value for this property in order to register its object adapter.

38.4.5 Starting the Registry

Now that the configuration file is written and the directory structure is prepared, we are ready to start the IceGrid registry:

```
$ icegridregistry --Ice.Config=/opt/ripper/registry.cfg
```

Additional command line options are supported, including those that allow the registry to run as a Windows service or Unix daemon. See Section 38.21.1 for more information.

38.4.6 Starting the Server

With the registry up and running, we can now start the server. At a command prompt, we run the program and pass an `--Ice.Config` option indicating the location of the configuration file:

```
$ /opt/ripper/bin/server \
   --Ice.Config=/opt/ripper/server.cfg
```

38.4.7 Review

This example demonstrated how to use IceGrid's location service, which is a core component of IceGrid's feature set. By incorporating IceGrid into our application, the client is now able to locate the `MP3EncoderFactory` object using only an indirect proxy and a value for `Ice.Default.Locator`. Furthermore, we can reconfigure the application in any number of ways without modifying the client's code or configuration.

For some applications, the functionality we have already achieved using IceGrid may be entirely sufficient. However, we have only just begun to explore IceGrid's capabilities, and there is much we can still do to improve our application. The next section shows how we can avoid the need to start our server manually by deploying our application onto an IceGrid node.

38.5 Using Deployment

In this section, we examine how to extend the capabilities of our sample application using IceGrid's deployment facility.

38.5.1 Architecture

The revised architecture for our application consists of a single IceGrid node responsible for our encoding server that runs on the computer named `Compute-`

`Server`. Figure 38.5 shows the client's initial invocation on its indirect proxy and the actions that IceGrid takes to make this invocation possible.

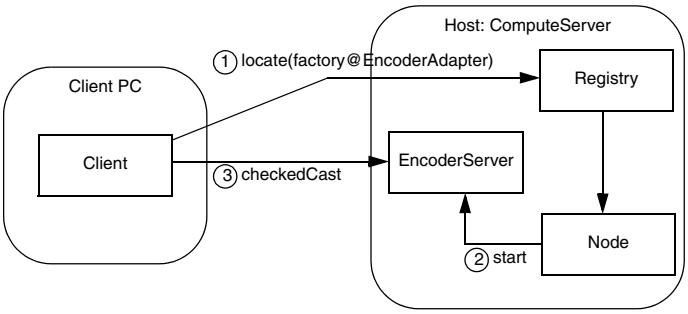


Figure 38.5. Architecture for deployed ripper application.

In contrast to the architecture shown in Section 38.4.1, we no longer need to manually start our server. In this revised application, the client's locate request prompts the registry to query the node about the server's state and start it if necessary. Once the server starts successfully, the locate request completes and subsequent client communication occurs directly with the server.

38.5.2 Descriptors

We can deploy our application using the **`icegridadmin`** command line utility (see Section 38.24.1), but first we must define our descriptors in XML. The descriptors are quite brief:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    id="EncoderAdapter"
                    endpoints="tcp"/>
            </server>
        </node>
    </application>
</icegrid>
```

For IceGrid's purposes, we have named our application `Ripper`. It consists of a single server, `EncoderServer`, assigned to the node `Node1`[1]. The server's `exe` attribute supplies the pathname of its executable, and the `activation` attribute indicates that the server should be activated on demand when necessary.

The object adapter's descriptor is the most interesting. As you can see, the `name` and `id` attributes both specify the value `EncoderAdapter`. The value of `name` reflects the adapter's name in the server process (i.e., the argument passed to `createObjectAdapter`) that is used for configuration purposes, whereas the value of `id` uniquely identifies the adapter within the registry and is used in indirect proxies. These attributes are not required to have the same value. Had we omitted the `id` attribute, IceGrid would have composed a unique value by combining the server name and adapter name to produce the following identifier:

```
EncoderServer.EncoderAdapter
```

The `endpoints` attribute defines one or more endpoints for the adapter. As explained in Section 38.4.4, these endpoints do not require a fixed port.

See Section 38.17 for detailed information on using XML to define descriptors.

38.5.3 Configuring the Registry and Node

In Section 38.4.2, we created the directory `/opt/ripper/registry` for use by the registry. The node also needs a subdirectory for its own purposes, so we will use `/opt/ripper/node`. Again, these directories must exist before starting the registry and node.

We also need to create an Ice configuration file to hold properties required by the registry and node. The file `/opt/ripper/config` contains the following properties:

```
# Registry properties
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsV
erifier
IceGrid.Registry.Data=/opt/ripper/registry
```

1. Since a computer typically runs only one node process, you might be tempted to give the node a name that identifies its host (such as `ComputeServerNode`). However, this naming convention becomes problematic as soon as you need to migrate the node to another host.

```
# Node properties
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=Node1
IceGrid.Node.Data=/opt/ripper/node
IceGrid.Node.CollocateRegistry=1
Ice.Default.Locator=IceGrid/Locator:tcp -p 4061
```

The registry and node can share this configuration file. In fact, by defining `IceGrid.Node.CollocateRegistry=1`, we have indicated that the registry and node should run in the same process.

Section 38.4.2 described the registry properties. One difference, however, is that we no longer define `IceGrid.Registry.DynamicRegistration`. By omitting this property, we force the registry to reject the registration of object adapters that have not been deployed.

The node properties are explained below:

- `IceGrid.Node.Endpoints`

  This property specifies the node's endpoints. A fixed port is not required.

- `IceGrid.Node.Name`

  This property defines the unique name for this node. Its value must match the descriptor shown in Section 38.5.2.

- `IceGrid.Node.Data`

  This property specifies the node's data directory.

- `Ice.Default.Locator`

  This property is defined for use by the **`icegridadmin`** tool. The node would also require this property if the registry is not collocated. Section 38.4.3 provides more information on this setting.

38.5.4 Configuring the Server

Server configuration is accomplished using descriptors. During deployment, the node creates a subdirectory tree for each server. Inside this tree the node creates a configuration file containing properties derived from the server's descriptors. For instance, the adapter's descriptor in Section 38.5.2 generates the following properties in the server's configuration file:

```
# Server configuration
Ice.Admin.ServerId=EncoderServer
Ice.Admin.Endpoints=tcp -h 127.0.0.1
Ice.ProgramName=EncoderServer
# Object adapter EncoderAdapter
EncoderAdapter.Endpoints=tcp
EncoderAdapter.AdapterId=EncoderAdapter
Ice.Default.Locator=IceGrid/Locator:default -p 4061
```

As you can see, the configuration file that IceGrid generates from the descriptor resembles the one we created in Section 38.4.4, with two additional properties:

- `Ice.Admin.ServerId`
- `Ice.Admin.Endpoints`

These properties enable the administrative facility that, among other features, allows an IceGrid node to gracefully deactivate the server. See Section 38.22 for more information.

Using the directory structure we established for our ripper application, the configuration file for `EncoderServer` has the file name shown below:

```
/opt/ripper/node/servers/EncoderServer/config/config
```

Note that this file should not be edited directly because any changes you make are lost the next time the node regenerates the file. The correct way to add properties to the file is to include property definitions in the server's descriptor. For example, we can add the property `Ice.Trace.Network=1` by modifying the server descriptor as follows:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    id="EncoderAdapter"
                    endpoints="tcp"/>
                <property name="Ice.Trace.Network"
                    value="1"/>
            </server>
        </node>
    </application>
</icegrid>
```

When a node activates a server, it passes the location of the server's configuration file using the `--Ice.Config` command-line argument. If you start a server manually from a command prompt, you must supply this argument yourself.

38.5.5 Starting IceGrid

Now that the configuration file is written and the directory structure is prepared, we are ready to start the IceGrid registry and node. Using a collocated registry and node, we only need to use one command:

```
$ icegridnode --Ice.Config=/opt/ripper/config
```

Additional command line options are supported, including those that allow the node to run as a Windows service or Unix daemon. See Section 38.21.2 for more information.

38.5.6 Deploying the Application

With the registry up and running, it is now time to deploy our application. Like our client, the **icegridadmin** utility also requires a definition for the `Ice.Default.Locator` property. We can start the utility with the following command:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
```

After confirming that it can contact the registry, **icegridadmin** provides a command prompt at which we deploy our application. Assuming our descriptor is stored in `/opt/ripper/app.xml`, the deployment command is shown below:

```
>>> application add "/opt/ripper/app.xml"
```

Next, confirm that the application has been deployed:

```
>>> application list
Ripper
```

You can start the server using this command:

```
>>> server start EncoderServer
```

Finally, you can retrieve the current endpoints of the object adapter:

```
>>> adapter endpoints EncoderAdapter
```

If you want to experiment further using **icegridadmin**, issue the **help** command and see Section 38.24.1.

38.5.7 Review

We have deployed our first IceGrid application, but you might be questioning whether it was worth the effort. Even at this early stage, we have already gained several benefits:

- We no longer need to manually start the encoder server before starting the client, because the IceGrid node automatically starts it if it is not active at the time a client needs it. If the server happens to terminate for any reason, such as an IceGrid administrative action or a server programming error, the node restarts it without intervention on our part.
- We can manage the application remotely using one of the IceGrid administration tools. The ability to remotely modify applications, start and stop servers, and inspect every aspect of your configuration is a significant advantage.

Admittedly, we have not made much progress yet in our stated goal of improving the performance of the ripper over alternative solutions that are restricted to running on a single computer. Our client now has the ability to easily delegate the encoding task to a server running on another computer, but we have not achieved the parallelism that we really need. For example, if the client created a number of encoders and used them simultaneously from multiple threads, the encoding performance might actually be *worse* than simply encoding the data directly in the client, as the remote computer would likely slow to a crawl while attempting to task-switch among a number of processor-intensive tasks.

38.5.8 Adding Nodes

Adding more nodes to our environment would allow us to distribute the encoding load to more compute servers. Using the techniques we have learned so far, let us investigate the impact that adding a node would have on our descriptors, configuration, and client application.

Descriptors

The addition of a node is mainly an exercise in cut and paste:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer1"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
```

```
                    endpoints="tcp"/>
            </server>
        </node>
        <node name="Node2">
            <server id="EncoderServer2"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    endpoints="tcp"/>
            </server>
        </node>
    </application>
</icegrid>
```

Note that we now have two `node` elements instead of a single one. You might be tempted to simply use the host name as the node name. However, in general, that is not a good idea. For example, you may want to run several IceGrid nodes on a single machine (for example, for testing). Similarly, you may have to rename a host at some point, or need to migrate a node to a different host. But, unless you also rename the node, that leads to the situation where you have a node with the name of a (possibly obsolete) host when the node in fact is not running on that host. Obviously, this makes for a confusing configuration—it is better to use abstract node names, such as `Node1`.

Aside from the new `node` element, notice that the server identifiers must be unique. The adapter name, however, can remain as `EncoderAdapter` because this name is used only for local purposes within the server process. In fact, using a different name for each adapter would actually complicate the server implementation, since it would somehow need to discover the name it should use when creating the adapter.

We have also removed the `id` attribute from our adapter descriptors; the default values supplied by IceGrid are sufficient for our purposes (see Section 38.17.1).

Configuration

We can continue to use the configuration file we created in Section 38.5.3 for our combined registry-node process. We need a separate configuration file for `Node2`, primarily to define a different value for the property `IceGrid.Node.Name`. However, we also cannot have two nodes configured with `IceGrid.Node.CollocateRegistry=1` because only one master registry is allowed, so we must remove this property:

```
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=Node2
IceGrid.Node.Data=/opt/ripper/node

Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost -p 4061
```

We assume that `/opt/ripper/node` refers to a local file system directory on the computer hosting `Node2`, and not a shared volume, because two nodes must not share the same data directory.

We have also modified the locator proxy to include the address of the host on which the registry is running.

Redeploying

After saving the new descriptors, you need to redeploy the application. Using **`icegridadmin`**, issue the following command:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> application update "/opt/ripper/app.xml"
```

Client

We have added a new node, but we still need to modify our client to take advantage of it. As it stands now, our client can delegate an encoding task to one of the two `MP3EncoderFactory` objects. The client selects a factory by using the appropriate indirect proxy:

- `factory@EncoderServer1.EncoderAdapter`
- `factory@EncoderServer2.EncoderAdapter`

In order to distribute the tasks among both factories, the client could use a random number generator to decide which factory receives the next task:

```
string adapter;
if ((rand() % 2) == 0)
    adapter = "EncoderServer1.EncoderAdapter";
else
    adapter = "EncoderServer2.EncoderAdapter";
Ice::ObjectPrx proxy =
    communicator->stringToProxy("factory@" + adapter);
Ripper::MP3EncoderFactoryPrx factory =
    Ripper::MP3EncoderFactoryPrx::checkedCast(proxy);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

There are a few disadvantages in this design:

- The client application must be modified each time a new compute server is added or removed because it knows all of the adapter identifiers.
- The client cannot distribute the load intelligently; it is just as likely to assign a task to a heavily-loaded computer as it is an idle one.

We describe better solutions in the sections that follow.

38.6 Well-known Objects

There are two types of indirect proxies (see Section 2.2.2): one specifies an identity and an object adapter identifier, while the other contains only an identity. The latter type of indirect proxy is known as a *well-known proxy*. A well-known proxy refers to a well-known object, that is, its identity alone is sufficient to allow the client to locate it. Ice requires all object identities in an application to be unique, but typically only a select few objects are able to be located via only their identities.

In earlier sections we showed the relationship between indirect proxies containing an object adapter identifier and the IceGrid configuration. Briefly, in order for a client to use a proxy such as `factory@EncoderAdapter`, an object adapter must be given the identifier `EncoderAdapter`.

A similar requirement exists for well-known objects. The registry maintains a table of these objects, which can be populated in a number of ways:

- statically in descriptors,
- programmatically using IceGrid's administrative interface,
- dynamically using an IceGrid administration tool.

The registry's database maps an object identity to a proxy. A locate request containing only an identity prompts the registry to consult this database. If a match is found, the registry examines the associated proxy to determine if addi-

tional work is necessary. For example, consider the well-known objects in Table 38.1.

Table 38.1. Well-known objects and their proxies.

Identity	Proxy
`Object1`	`Object1:tcp -p 10001`
`Object2`	`Object2@TheAdapter`
`Object3`	`Object3`

The proxy associated with `Object1` already contains endpoints, so the registry can simply return this proxy to the client.

For `Object2`, the registry notices the adapter id and checks to see whether it knows about an adapter identified as `TheAdapter`. If it does, it attempts to obtain the endpoints of that adapter, which may cause its server to be started. If the registry is successfully able to determine the adapter's endpoints, it returns a direct proxy containing those endpoints to the client. If the registry does not recognize `TheAdapter` or cannot obtain its endpoints, it returns the indirect proxy `Object2@TheAdapter` to the client. Upon receipt of another indirect proxy, the Ice run time in the client will try once more to resolve the proxy, but generally this will not succeed and the Ice run time in the client will raise a `NoEndpointException` as a result.

Finally, `Object3` represents a hopeless situation: how can the registry resolve `Object3` when its associated proxy refers to itself? In this case, the registry returns the proxy `Object3` to the client, which causes the client to raise `NoEndpointException`. Clearly, you should avoid this situation.

38.6.1 Object Types

The registry's database not only associates an identity with a proxy, but also a type. Technically, the "type" is an arbitrary string but, by convention, that string represents the most-derived Slice type of the object. For example, the Slice type of the encoder factory in our ripper application is `::Ripper::MP3EncoderFactory`.

Object types are useful when performing queries, as discussed in Section 38.6.5.

38.6.2 Object Descriptors

The object descriptor (see Section 38.17.14) adds a well-known object to the registry. It must appear within the context of an adapter descriptor, as shown in the XML example below:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    id="EncoderAdapter"
                    endpoints="tcp">
                    <object identity="EncoderFactory"
                        type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
    </application>
</icegrid>
```

During deployment, the registry associates the identity `EncoderFactory` with the indirect proxy `EncoderFactory@EncoderAdapter`. If the adapter descriptor had omitted the adapter id, the registry would have generated a unique identifier using the server id and the adapter name.

In this example, the object's type is specified explicitly. See Section 38.6.1 for more information on object types.

38.6.3 Adding Objects in a Program

The `IceGrid::Admin` interface defines several operations that manipulate the registry's database of well-known objects:

```
module IceGrid {
interface Admin {
    ...
    void addObject(Object* obj)
        throws ObjectExistsException,
               DeploymentException;
```

```
    void updateObject(Object* obj)
        throws ObjectNotRegisteredException,
               DeploymentException;
    void addObjectWithType(Object* obj, string type)
        throws ObjectExistsException,
               DeploymentException;
    void removeObject(Ice::Identity id)
        throws ObjectNotRegisteredException,
               DeploymentException;
    ...
};
};
```

- `addObject`

  The `addObject` operation adds a new object to the database. The proxy argument supplies the identity of the well-known object. If an object with the same identity has already been registered, the operation raises `ObjectExistsException`. Since this operation does not accept an argument supplying the object's type, the registry invokes `ice_id` on the given proxy to determine its most-derived type. The implication here is that the object must be available in order for the registry to obtain its type. If the object is not available, `addObject` raises `DeploymentException`.

- `updateObject`

  The `updateObject` operation supplies a new proxy for the well-known object whose identity is encapsulated by the proxy. If no object with the given identity is registered, the operation raises `ObjectNotRegisteredException`. The object's type is not modified by this operation.

- `addObjectWithType`

  The `addObjectWithType` operation behaves like `addObject`, except the object's type is specified explicitly and therefore the registry does not attempt to invoke `ice_id` on the given proxy (even if the type is an empty string).

- `removeObject`

  The `removeObject` operation removes the well-known object with the given identity from the database. If no object with the given identity is registered, the operation raises `ObjectNotRegisteredException`.

The following C++ example produces the same result as deploying the descriptor in Section 38.6.2:

```
Ice::ObjectAdapterPtr adapter =
    communicator->createObjectAdapter("EncoderAdapter");
Ice::Identity ident =
    communicator->stringToIdentity("EncoderFactory");
FactoryPtr f= new FactoryI;
Ice::ObjectPrx factory = adapter->add(f, ident);
IceGrid::AdminPrx admin = // ...
try {
    admin->addObject(factory); // OOPS!
} catch (const IceGrid::ObjectExistsException &) {
    admin->updateObject(factory);
}
```

After obtaining a proxy for the `IceGrid::Admin` interface (see Section 38.14), the code invokes `addObject`. Notice that the code traps `ObjectExistsException` and calls `updateObject` instead when the object is already registered.

There is one subtle problem in this code: calling `addObject` causes the registry to invoke `ice_id` on our factory object, but we have not yet activated the object adapter. As a result, our program will hang indefinitely at the call to `addObject`. One solution is to activate the adapter prior to the invocation of `addObject`; another solution is to use `addObjectWithType` as shown below:

```
Ice::ObjectAdapterPtr adapter =
    communicator->createObjectAdapter("EncoderAdapter");
Ice::Identity ident =
    communicator->stringToIdentity("EncoderFactory");
FactoryPtr f = new FactoryI;
Ice::ObjectPrx factory = adapter->add(f, ident);
IceGrid::AdminPrx admin = // ...
try {
    admin->addObjectWithType(factory, factory->ice_id());
} catch (const IceGrid::ObjectExistsException &) {
    admin->updateObject(factory);
}
```

38.6.4 Adding Objects with `icegridadmin`

The **`icegridadmin`** utility (see Section 38.24.1) provides commands that are the functional equivalents of the Slice operations shown in Section 38.6.3. We can use the utility to manually register the `EncoderFactory` object from Section 38.6.2:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> object add "EncoderFactory@EncoderAdapter"
```

Use the **`object list`** command to verify that the object was registered successfully:

```
>>> object list
EncoderFactory
IceGrid/Query
IceGrid/Locator
IceGrid/Registry
IceGrid/InternalRegistry-Master
```

To specify the object's type explicitly, append it to the **`object add`** command:

```
>>> object add "EncoderFactory@EncoderAdapter" \
"::Ripper::MP3EncoderFactory"
```

Finally, the object is removed from the registry like this:

```
>>> object remove "EncoderFactory"
```

38.6.5 Queries

The registry's database of well-known objects is not used solely for resolving indirect proxies. The database can also be queried interactively to find objects in a variety of ways. The `IceGrid::Query` interface supplies this functionality:

```
module IceGrid {
enum LoadSample {
    LoadSample1,
    LoadSample5,
    LoadSample15
};

interface Query {
    idempotent Object* findObjectById(Ice::Identity id);
    idempotent Object* findObjectByType(string type);
    idempotent Object* findObjectByTypeOnLeastLoadedNode(
            string type, LoadSample sample);
    idempotent Ice::ObjectProxySeq findAllObjectsByType(
            string type);
    idempotent Ice::ObjectProxySeq findAllReplicas(Object* proxy);
};
};
```

- `findObjectById`

  The `findObjectById` operation returns the proxy associated with the given identity of a well-known object. It returns a null proxy if no match was found.
- `findObjectByType`

  The `findObjectByType` operation returns a proxy for an object registered with the given type. If more than one object has the same type, the registry selects one at random. The operation returns a null proxy if no match was found.
- `findObjectByTypeOnLeastLoadedNode`

  The `findObjectByTypeOnLeastLoadedNode` operation considers the system load when selecting one of the objects with the given type. If the registry is unable to determine which node hosts an object (for example, because the object was registered with a direct proxy and not an adapter id), the object is considered to have a load value of `1` for the purposes of this operation. The sample argument determines the interval over which the loads are averaged (one, five, or fifteen minutes). The operation returns a null proxy if no match was found.
- `findAllObjectsByType`

  The `findAllObjectsByType` operation returns a sequence of proxies representing the well-known objects having the given type. The operation returns an empty sequence if no match was found.
- `findAllReplicas`

  Given an indirect proxy for a replicated object, the `findAllReplicas` operation returns a sequence of proxies representing the individual replicas. An application can use this operation when it is necessary to communicate directly with one or more replicas.

Be aware that the operations accepting a `type` parameter are not equivalent to invoking `ice_isA` on each object to determine whether it supports the given type, a technique that would not scale well as the for a large number of registered objects. Rather, the operations simply compare the given type to the object's registered type or, if the object was registered without a type, to the object's most-derived Slice type as determined by the registry (see Section 38.6.1).

38.6.6 Application Changes

Well-known objects are another IceGrid feature we can incorporate into our ripper application.

Descriptors

First we'll modify the descriptors from Section 38.5.8 to add two well-known objects:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer1"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    endpoints="tcp">
                    <object identity="EncoderFactory1"
                        type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
        <node name="Node2">
            <server id="EncoderServer2"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    endpoints="tcp">
                    <object identity="EncoderFactory2"
                        type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
    </application>
</icegrid>
```

At first glance, the addition of the well-known objects does not appear to simplify our client very much. Rather than selecting which of the two adapters receives the next task, we now need to select one of the well-known objects.

Querying with `findAllObjectsByType`

The `IceGrid::Query` interface provides a way to eliminate the client's dependency on object adapter identifiers and object identities. Since our factories are registered with the same type, we can search for all objects of that type:

```
Ice::ObjectPrx proxy =
    communicator->stringToProxy("IceGrid/Query");
IceGrid::QueryPrx query = IceGrid::QueryPrx::checkedCast(proxy);
Ice::ObjectProxySeq seq;
string type = Ripper::MP3EncoderFactory::ice_staticId();
```

```
seq = query->findAllObjectsByType(type);
if (seq.empty()) {
    // no match
}
Ice::ObjectProxySeq::size_type index = ... // random number
Ripper::MP3EncoderFactoryPrx factory =
    Ripper::MP3EncoderFactoryPrx::checkedCast(seq[index]);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

This example invokes `findAllObjectsByType` and then randomly selects an element of the sequence.

Querying with `findObjectByType`

We can simplify the client further using `findObjectByType` instead, which performs the randomization for us:

```
Ice::ObjectPrx proxy =
    communicator->stringToProxy("IceGrid/Query");
IceGrid::QueryPrx query = IceGrid::QueryPrx::checkedCast(proxy);
Ice::ObjectPrx obj;
string type = Ripper::MP3EncoderFactory::ice_staticId();
obj = query->findObjectByType(type);
if (!obj) {
    // no match
}
Ripper::MP3EncoderFactoryPrx factory =
    Ripper::MP3EncoderFactoryPrx::checkedCast(obj);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

Querying with `findObjectByTypeOnLeastLoadedNode`

So far the use of `IceGrid::Query` has allowed us to simplify our client, but we have not gained any functionality. If we replace the call to `findObjectByType` with `findObjectByTypeOnLeastLoadedNode`, we can improve the client by distributing the encoding tasks more intelligently. The change to the client's code is trivial:

```
Ice::ObjectPrx proxy =
    communicator->stringToProxy("IceGrid/Query");
IceGrid::QueryPrx query = IceGrid::QueryPrx::checkedCast(proxy);
Ice::ObjectPrx obj;
string type = Ripper::MP3EncoderFactory::ice_staticId();
obj = query->findObjectByTypeOnLeastLoadedNode(type,
    IceGrid::LoadSample1);
if (!obj) {
```

```
    // no match
}
Ripper::MP3EncoderFactoryPrx factory =
    Ripper::MP3EncoderFactoryPrx::checkedCast(obj);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

Review

Incorporating intelligent load distribution is a worthwhile enhancement and is a capability that would be time consuming to implement ourselves. However, our current design uses only well-known objects in order to make queries possible. We do not really need the encoder factory object on each compute server to be individually addressable as a well-known object, a fact that seems clear when we examine the identities we assigned to them: `EncoderFactory1`, `EncoderFactory2`, and so on. IceGrid's replication features, discussed in Section 38.9, give us the tools we need to improve our design.

38.7 Templates

IceGrid templates simplify the task of creating the descriptors for an application. A template is a parameterized descriptor that you can instantiate as often as necessary. Templates are descriptors in their own right. They are components of an IceGrid application and therefore they are stored in the registry's database. As such, their use is not restricted to XML files; templates can also be created and instantiated interactively using the graphical administration tool (see Section 38.24.2).

You can define templates for server and service descriptors. The focus of this section is server templates; Section 38.8 explains service descriptors and templates.

38.7.1 Server Templates

You may recall from prior sections that the XML description of our sample application defined two nearly identical servers:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer1"
                exe="/opt/ripper/bin/server"
```

```
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    endpoints="tcp"/>
            </server>
        </node>
        <node name="Node2">
            <server id="EncoderServer2"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    endpoints="tcp"/>
            </server>
        </node>
    </application>
</icegrid>
```

This example is an excellent candidate for a server template. Equivalent definitions that incorporate a template are shown below:

```
<icegrid>
    <application name="Ripper">
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <server id="EncoderServer${index}"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    endpoints="tcp"/>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
                index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
                index="2"/>
        </node>
    </application>
</icegrid>
```

We have defined a server template named `EncoderServerTemplate`. Nested within the `server-template` element is a `server` element that defines an encoder server. The only difference between this `server` element and our previous example is that it is now parameterized: the template parameter `index` is used to form unique identifiers for the server and its adapter. The symbol

${index} is replaced with the value of the index parameter wherever it occurs.

The template is instantiated by a server-instance element, which may be used anywhere that a server element is used. The server-instance element identifies the template to be instantiated, and supplies a value for the index parameter.

Although we have not significantly reduced the length of our XML file, we have made it more readable. And more importantly, deploying this server on additional nodes has become much easier.

38.7.2 Template Parameters

Parameters enable you to customize each instance of a template as necessary. The example in Section 38.7.1 defined the index parameter with a different value for each instance to ensure that identifiers are unique. A parameter may also declare a default value that is used in the template if no value is specified for it. In our sample application the index parameter is considered mandatory and therefore should not have a default value, but we can illustrate this feature in another way. For example, suppose that the pathname of the server's executable may change on each node. We can supply a default value for this attribute and override it when necessary:

```
<icegrid>
    <application name="Ripper">
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <parameter name="exepath"
                default="/opt/ripper/bin/server"/>
            <server id="EncoderServer${index}"
                exe="${exepath}"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    endpoints="tcp"/>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
                index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
```

```
                index="2" exepath="/opt/ripper-test/bin/server"/>
        </node>
    </application>
</icegrid>
```

As you can see, the instance on `Node1` uses the default value for the new parameter `exepath`, but the instance on `Node2` defines a different location for the server's executable.

For complete details about substitution rules and other semantics, see Section 38.18.

38.7.3 Property Sets

As we saw in the preceding section, template parameters allow you to customize each instance of a server template. Template parameters with default values allow you to define commonly used configuration options. However, you might want to have additional configuration properties for a given instance without having to add a parameter. For example, to debug a server instance on a specific node, you might want to start the server with the `Ice.Trace.Network` property set; it would be inconvenient to have to add a parameter to the template just to set that property.

To cater for such scenarios, it is possible to specify additional properties for a server instance without modifying the template. You can define such properties in the `server-instance` element, for example:

```
<icegrid>
    <application>
        ...
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
                             index="2">
                <properties>
                    <property name="Ice.Trace.Network" value="2"/>
                </properties>
            </server-instance>
        </node>
    </application>
</icegrid>
```

This sets the `Ice.Trace.Network` property for a specific server.

38.7.4 Default Templates

The IceGrid registry can be configured to supply any number of default template descriptors for use in your applications. The configuration property `IceGrid.Registry.DefaultTemplates` specifies the pathname of an XML file containing template definitions. One such template file is provided in the Ice distribution as `config/templates.xml`, which contains helpful templates for deploying Ice services such as IcePatch2 and Glacier2.

The template file must use the structure shown below:

```
<icegrid>
    <application name="DefaultTemplates">
        <server-template id="EncoderServerTemplate">
            ...
        </server-template>
    </application>
</icegrid>
```

The name you give to the application is not important, and you may only define server and service (see Section 38.8.2) templates within it. After configuring the registry to use this file, your default templates become available to every application that imports them.

The descriptor for each application indicates whether the default templates should be imported. (By default they are not imported.) If the templates are imported, they are essentially copied into the application descriptor and treated no differently than templates defined by the application itself. As a result, changes to the file containing default templates have no effect on existing application descriptors. In XML, the attribute `import-default-templates` determines whether the default templates are imported, as shown in the following example:

```
<icegrid>
    <application name="Ripper"
        import-default-templates="true">
        ...
    </application>
</icegrid>
```

38.7.5 Using Templates with `icegridadmin`

The IceGrid administration tools allow you to inspect templates and instantiate new servers dynamically. First, let us ask **`icegridadmin`** to describe the server template from Section 38.7.1:

```
$ icegridadmin --Ice.Config=/opt/ripper/config
>>> server template describe Ripper \
EncoderServerTemplate
```

This command generates the following output:

```
server template `EncoderServerTemplate'
{
    parameters = `index exepath'
    server `EncoderServer${index}'
    {
        exe = `${exepath}'
        activation = `on-demand'
        properties
        {
            EncoderAdapter.Endpoints = `tcp'
        }
        adapter `EncoderAdapter'
        {
            id = `EncoderAdapter${index}'
            replica group id = `'
            endpoints = `tcp'
            register process = `false'
            server lifetime = `true'
        }
    }
}
```

Notice that the server id is a parameterized value; it cannot be evaluated until the template is instantiated with values for its parameters.

Next, we can use **icegridadmin** to create an instance of the encoder server template on a new node:

```
>>> server template instantiate Ripper Node3 \
EncoderServerTemplate index=3
```

The command requires that we identify the application, node and template, as well as supply any parameters needed by the template. The new server instance is permanently added to the registry's database, but if we intend to keep this configuration it is a good idea to update the XML description of our application to reflect these changes and avoid potential synchronization issues.

38.8 IceBox Integration

IceGrid makes it easy to configure an IceBox server (see Chapter 43) with one or more services.

38.8.1 Descriptors

An IceBox server shares many of the same characteristics as other servers, but its special requirements necessitate a new descriptor. Unlike other servers, an IceBox server generally hosts multiple independent services, each requiring its own communicator instance and configuration file.

As an example, the following application deploys an IceBox server containing one service:

```
<icegrid>
    <application name="IceBoxDemo">
        <node name="Node">
            <icebox id="IceBoxServer"
                exe="/opt/Ice/bin/icebox"
                activation="on-demand">
                <service name="ServiceA" entry="servicea:create">
                    <adapter name="${service}" endpoints="tcp"/>
                </service>
            </icebox>
        </node>
    </application>
</icegrid>
```

It looks very similar to a server descriptor. The most significant difference is the service descriptor, which is constructed much like a server in that you can declare its attributes such as object adapters and configuration properties. The order in which services are defined determines the order in which they are loaded by the IceBox server.

The value of the adapter's `name` attribute needs additional explanation. The symbol `service` is one of the names reserved by IceGrid. In the context of a service descriptor, `${service}` is replaced with the service's name, and so the object adapter is also named `ServiceA`. See Section 38.18.2 for more information on reserved names.

38.8.2 Service Templates

If you are familiar with templates in general (see Section 38.7), an IceBox service template is readily understandable:

```
<icegrid>
    <application name="IceBoxApp">
        <service-template id="ServiceTemplate">
            <parameter name="name"/>
            <service name="${name}" entry="DemoService:create">
                <adapter name="${service}"
                    endpoints="default"/>
                <property name="${service}.Identity"
                    value="${server}-${service}"/>
            </service>
        </service-template>
        <node name="Node1">
            <icebox id="IceBoxServer" endpoints="default"
                exe="/opt/Ice/bin/icebox" activation="on-demand">
                <service-instance template="ServiceTemplate"
                    name="Service1"/>
            </icebox>
        </node>
    </application>
</icegrid>
```

In this application, an IceBox server is deployed on a node and has one service instantiated from the service template. Of particular interest is the `property` descriptor, which uses another reserved name `server` to form the property value. When the template is instantiated, the symbol `${server}` is replaced with the name of the enclosing server, so the property definition expands as follows:

```
Service1.Identity=IceBoxServer-Service1
```

See Section 38.18.2 for more information on reserved names.

As with server instances, you can specify additional properties for the service instance without modifying the template. These properties can be defined in the `service-instance` element, as shown below:

```
<icegrid>
    <application name="IceBoxApp">
        ...
        <node name="Node1">
            <icebox id="IceBoxServer"endpoints="default"
                exe="/opt/Ice/bin/icebox" activation="on-demand">
                <service-instance template="ServiceTemplate"
```

```
                    name="Service1">
                    <properties>
                        <property name="Ice.Trace.Network"
                            value="1"/>
                    </properties>
                </service-instance>
            </icebox>
        </node>
    </application>
</icegrid>
```

38.8.3 Advanced Templates

A more sophisticated use of templates involves instantiating a service template in a server template:

```
<icegrid>
    <application name="IceBoxApp">
        <service-template id="ServiceTemplate">
            <parameter name="name"/>
            <service name="${name}" entry="DemoService:create">
                <adapter name="${service}"
                    endpoints="default"/>
                <property name="${name}.Identity"
                    value="${server}-${name}"/>
            </service>
        </service-template>
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <icebox id="${id}" endpoints="default"
                exe="/opt/Ice/bin/icebox" activation="on-demand">
                <service-instance template="ServiceTemplate"
                    name="Service1"/>
            </icebox>
        </server-template>
        <node name="Node1">
            <server-instance template="ServerTemplate"
                id="IceBoxServer"/>
        </node>
    </application>
</icegrid>
```

This application is equivalent to the definition from Section 38.8.2. Now, however, the process of deploying an identical server on several nodes has become much simpler.

If you need the ability to customize the configuration of a particular service instance, your server instance can define a property set that applies only to the desired service:

```
<icegrid>
    <application name="IceBoxApp">
        <node name="Node1">
            <server-instance template="ServerTemplate"
                id="IceBoxServer">
                <properties service="Service1">
                    <property name="Ice.Trace.Network"
                        value="1"/>
                </properties>
            </server-instance>
        </node>
    </application>
</icegrid>
```

As this example demonstrates, the `service` attribute of the property set denotes the name of the target service.

38.9 Object Adapter Replication

As an implementation of an Ice location service, IceGrid supports object adapter replication as described on page 12. An application defines its replica groups and their participating object adapters using descriptors, and IceGrid generates the server configurations automatically.

38.9.1 Replica Group Descriptor

The descriptor that defines a replica group can optionally declare well-known objects as well as configure the group to determine its behavior during locate requests. Consider this example:

```
<icegrid>
    <application name="ReplicaApp">
        <replica-group id="ReplicatedAdapter">
            <object identity="TheObject"
                type="::Demo::ObjectType"/>
        </replica-group>
        <node name="Node">
            <server id="ReplicaServer" activation="on-demand"
```

```
                exe="/opt/replica/bin/server">
                <adapter name="TheAdapter" endpoints="default"
                    replica-group="ReplicatedAdapter"/>
            </server>
        </node>
    </application>
</icegrid>
```

The adapter's descriptor declares itself to be a member of the replica group `ReplicatedAdapter`, which must have been previously created by a replica group descriptor.

The replica group `ReplicatedAdapter` declares a well-known object so that an indirect proxy of the form `TheObject` is equivalent to the indirect proxy `TheObject@ReplicatedAdapter`. Since this trivial example defines only one adapter in the replica group, the proxy `TheObject` is also equivalent to `TheObject@TheAdapter`.

38.9.2 Replica Group Membership

An object adapter participates in a replica group by specifying the group's id in the adapter's `ReplicaGroupId` configuration property. Identifying the replica group in the IceGrid descriptor for an object adapter causes the node to include the equivalent `ReplicaGroupId` property in the configuration file it generates for the server.

By default, the IceGrid registry requires the membership of a replica group to be statically defined. When you create a descriptor for an object adapter that identifies a replica group, the registry adds that adapter to the group's list of valid members. During an adapter's activation, when it describes its endpoints to the registry, an adapter that also claims membership in a replica group is validated against the registry's internal list.

In a properly configured IceGrid application, this activity occurs without incident, but there are situations in which validation can fail. For example, adapter activation fails if an adapter's id is changed without notifying the registry, such as by manually modifying the server configuration file that was generated by a node.

It is also possible for activation to fail when the IceGrid registry is being used solely as a location service, in which case descriptors have not been created and therefore the registry has no advance knowledge of the replica groups or their members. In this situation, adapter activation causes the server to receive `NotRegisteredException` unless the registry is configured to allow dynamic registration, which you can do by defining the following property:

```
IceGrid.Registry.DynamicRegistration=1
```

With this configuration, a replica group is created implicitly as soon as an adapter declares membership in it, and any adapter is allowed to participate.

The use of dynamic registration often leads to the accumulation of obsolete replica groups and adapters in the registry. The IceGrid administration tools (see Section 38.24) allow you to inspect and clean up the registry's state.

38.9.3 Application Changes

Replication is a perfect fit for the ripper application. The collection of encoder factory objects should be treated as a single logical object, and replication makes that possible.

Descriptors

Adding a replica group descriptor to our application is very straightforward:

```
<icegrid>
    <application name="Ripper">
        <replica-group id="EncoderAdapters">
            <object identity="EncoderFactory"
                type="::Ripper::MP3EncoderFactory"/>
        </replica-group>
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <parameter name="exepath"
                default="/opt/ripper/bin/server"/>
            <server id="EncoderServer${index}"
                exe="${exepath}"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    replica-group="EncoderAdapters"
                    endpoints="tcp"/>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
                index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
```

```
                index="2"/>
        </node>
    </application>
</icegrid>
```

The new descriptor adds the replica group called `EncoderAdapters` and registers a well-known object with the identity `EncoderFactory`. The adapter descriptor in the server template has been changed to declare its membership in the replica group.

Client

In comparison to the examples from Section 38.6.6 that used queries, the new version of our client has become much simpler:

```
Ice::ObjectPrx obj =
    communicator->stringToProxy("EncoderFactory");
Ripper::MP3EncoderFactoryPrx factory =
    Ripper::MP3EncoderFactoryPrx::checkedCast(obj);
Ripper::MP3EncoderPrx encoder = factory->createEncoder();
```

The client no longer needs to use the `IceGrid::Query` interface, but simply creates a proxy for a well-known object and lets the Ice run time transparently interact with the location service. In response to a locate request for `EncoderFactory`, the registry returns a proxy containing the endpoints of both object adapters. The Ice run time in the client selects one of the endpoints at random, meaning we have now lost some functionality compared to the prior example in which system load was considered when selecting an endpoint. We will learn how to rectify this situation in Section 38.10.

38.10 Load Balancing

Replication is an important IceGrid feature but, when combined with load balancing, replication becomes even more useful.

IceGrid nodes regularly report the system load of their hosts to the registry. The replica group's configuration determines whether the registry actually considers system load information while processing a locate request. Its configuration also specifies how many replicas to include in the registry's response.

IceGrid's load balancing capability assists the client in obtaining an initial set of endpoints for the purpose of establishing a connection. Once a client has established a connection, all subsequent requests on the proxy that initiated the connec-

tion are normally sent to the same server without further consultation with the registry. As a result, the registry's response to a locate request can only be viewed as a snapshot of the replicas at a particular moment. If system loads are important to the client, it must take steps to periodically contact the registry and update its endpoints. Section 32.17.2 provides more information on this subject.

38.10.1 Configuring a Replica Group

A replica group descriptor optionally contains a load balancing descriptor that determines how system loads are used in locate requests. The load balancing descriptor specifies the following information:

- Type

  Several types of load balancing are supported. See Section 38.10.2 for details.

- Sampling interval

  One of the load balancing types considers system load statistics, which are reported by each node at regular intervals. The replica group can specify a sampling interval of one, five, or fifteen minutes. Choosing a sampling interval requires balancing the need for up-to-date load information against the desire to minimize transient spikes.

  On Unix platforms, the node reports the system's load average for the selected interval, while on Windows the node reports the CPU utilization averaged over the interval.

- Number of replicas

  The replica group can instruct the registry to return the endpoints of one (the default) or more object adapters. If the specified number *N* is larger than one, the proxy returned in response to a locate request contains the endpoints of at most *N* object adapters. If *N* is 0, the proxy contains the endpoints of all the object adapters. The Ice run time in the client selects one of these endpoints at random (see Section 32.11.4).

For example, the descriptor shown below uses adaptive load balancing to return the endpoints of the two least-loaded object adapters sampled with five-minute intervals:

```
<replica-group id="ReplicatedAdapter">
    <load-balancing type="adaptive" load-sample="5"
        n-replicas="2"/>
</replica-group>
```

The type must be specified, but the remaining attributes are optional.

38.10.2 Load Balancing Types

A replica group can select one of the following load balancing types:

- Random

  Random load balancing selects the requested number of object adapters at random. The registry does not consider system load for a replica group with this type.

- Adaptive

  Adaptive load balancing uses system load information to choose the least-loaded object adapters over the requested sampling interval. This is the only load balancing type that uses sampling intervals.

- Round Robin

  Round robin load balancing returns the least recently used object adapters. The registry does not consider system load for a replica group with this type. Note that the round-robin information is not shared between registry replicas; each replica maintains its own notion of the "least recently used" object adapters.

- Ordered

  Ordered load balancing selects the requested number of object adapters by priority. A priority can be set for each object adapter member of the replica group.

Choosing the proper type of load balancing is highly dependent on the needs of client applications. Achieving the desired load balancing and fail-over behavior may also require the cooperation of your clients. To that end, it is very important that you understand how and when the Ice run time uses a locator to resolve indirect proxies; Section 32.17.2 discusses this subject in detail.

38.10.3 Application Changes

The only change we need to make to the ripper application is the addition of a load balancing descriptor:

```
<icegrid>
    <application name="Ripper">
        <replica-group id="EncoderAdapters">
            <load-balancing type="adaptive"/>
```

```
            <object identity="EncoderFactory"
                type="::Ripper::MP3EncoderFactory"/>
        </replica-group>
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <parameter name="exepath"
                default="/opt/ripper/bin/server"/>
            <server id="EncoderServer${index}"
                exe="${exepath}"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    replica-group="EncoderAdapters"
                    endpoints="tcp"/>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
                index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
                index="2"/>
        </node>
    </application>
</icegrid>
```

Using adaptive load balancing, we have regained the functionality we forfeited in Section 38.9.3. Namely, we now select the object adapter on the least-loaded node, and no changes were necessary in the client.

38.10.4 Interacting with Replicas

In some applications you may have a need for interacting directly with the replicas of an object. For example, the application may want to implement a custom load-balancing strategy. In this situation you might be tempted to call `ice_getEndpoints` on the proxy of a replicated object in an effort to obtain the endpoints of all replicas, but that is not the correct solution because the proxy is indirect and therefore contains no endpoints. The proper approach is to use the `findAllReplicas` operation provided by the `IceGrid::Query` interface. See Section 38.6.5 for more information.

38.11 Sessions

IceGrid provides a resource allocation facility that coordinates access to the objects and servers of an IceGrid application. To allocate a resource for exclusive use, a client must first establish a session by authenticating itself with the IceGrid registry or a Glacier2 router, after which the client may reserve objects and servers that the application indicates are allocatable. The client should release the resource when it is no longer needed, otherwise IceGrid reclaims it when the client's session terminates or expires due to inactivity.

An allocatable server offers at least one allocatable object. The server is considered to be allocated when its first allocatable object is claimed, and is not released until all of its allocated objects are released. While the server is allocated by a client, no other clients can allocate its objects.

38.11.1 Creating a Session

A client must create an IceGrid session before it can allocate objects. If you have configured a Glacier2 router to use IceGrid's session managers (see Section 38.15), the client's router session satisfies this requirement. For more information on creating a Glacier2 session, see Section 42.3.6.

In the absence of Glacier2, an IceGrid client invokes `createSession` or `createSessionFromSecureConnection` on IceGrid's `Registry` interface to create a session:

```
module IceGrid {
    exception PermissionDeniedException {
        string reason;
    };

    interface Registry {
        Session* createSession(string userId, string password)
            throws PermissionDeniedException;

        Session* createSessionFromSecureConnection()
            throws PermissionDeniedException;

        idempotent int getSessionTimeout();
    };
};
```

The `createSession` operation expects a username and password and returns a session proxy if the client is allowed to create a session. By default, IceGrid does not allow the creation of sessions. You must define the registry property `IceGrid.Registry.PermissionsVerifier` with the proxy of a permissions verifier object to enable session creation with `createSession` (see Section 38.11.2).

The `createSessionFromSecureConnection` operation does not require a username and password because it uses the credentials supplied by an SSL connection to authenticate the client (see Chapter 41). As with `createSession`, you must configure the proxy of a permissions verifier object before clients can use `createSessionFromSecureConnection` to create a session. In this case, the property is `IceGrid.Registry.SSLPermissionsVerifier` (see Section 38.11.2).

To create a session, the client obtains the registry proxy by converting the well-known proxy string `"IceGrid/Registry"` to a proxy object with the communicator, downcasts the proxy to the `IceGrid::Registry` interface, and invokes on one of the operations. The sample code below demonstrates how to do it in C++; the code will look very similar in other language mappings.

```
Ice::ObjectPrx base =
    communicator->stringToProxy("IceGrid/Registry");
IceGrid::RegistryPrx registry =
    IceGrid::RegistryPrx::checkedCast(base);
string username = ...;
string password = ...;
IceGrid::SessionPrx session;
try {
    session = registry->createSession(username, password);
} catch (const IceGrid::PermissionDeniedException & ex) {
    cout << "permission denied:\n" << ex.reason << endl;
}
```

Note that you have to substitute the correct instance name for the object identity category (see page 1902) when you call `stringToProxy`.

After creating the session, the client must keep it alive by periodically invoking its `keepAlive` operation. The session expires if the client does not invoke `keepAlive` within the configured timeout period, which can be obtained by calling the `getSessionTimeout` operation on the `Registry` interface.

If a session times out, or if the client explicitly terminates the session by invoking its `destroy` operation, IceGrid automatically releases all objects allocated using that session.

38.11.2 Access Control

As described in Section 38.11.1 above, you must configure the IceGrid registry with the proxy of at least one permissions verifier object to enable session creation:

- `IceGrid.Registry.PermissionsVerifier`

  This property supplies the proxy of an object that implements the interface `Glacier2::PermissionsVerifier`. Defining this property allows clients to create sessions using `createSession`.

- `IceGrid.Registry.SSLPermissionsVerifier`

  This property supplies the proxy of an object that implements the interface `Glacier2::SSLPermissionsVerifier`. Defining this property allows clients to create sessions using `createSessionFromSecureConnection`.

IceGrid supplies built-in permissions verifier objects:

- A null permissions verifier for TCP/IP. This object accepts any username and password and should only be used in a secure environment where no access control is necessary. You select this verifier object by defining the following configuration property:

  ```
  IceGrid.Registry.PermissionsVerifier=\
      <instance-name>/NullPermissionsVerifier
  ```

  Note that you have to substitute the correct instance name for the object identity category (see page 1902).

- A null permissions verifier for SSL, analogous to the one for TCP/IP. You select this verifier object by defining the following configuration property:

  ```
  IceGrid.Registry.SSLPermissionsVerifier=\
      <instance-name>/NullSSLPermissionsVerifier
  ```

- A file-based permissions verifier. This object uses an access control list in a file that contains username-password pairs. The format of the password file is the same as the format of Glacier2 password files described in Section 42.3.2. You enable this verifier implementation by defining the configuration property `IceGrid.Registry.CryptPasswords` with the pathname of the password file. Note that this property is ignored if you specify the proxy of a permissions verifier object using `IceGrid.Registry.PermissionsVerifier`.

If you decide to implement your own permissions verifier object, Section 42.6.1 describes the Glacier2 interfaces in detail.

38.11.3 Allocating Objects

A client allocates objects using the session proxy returned from `createSession` or `createSessionFromSecureConnection`. The proxy supports the `Session` interface shown below:

```
module IceGrid {
    exception ObjectNotRegisteredException {
        Ice::Identity id;
    };

    exception AllocationException {
        string reason;
    };

    exception AllocationTimeoutException
        extends AllocationException {
    };

    interface Session extends Glacier2::Session {

        idempotent void keepAlive();

        Object* allocateObjectById(Ice::Identity id)
            throws ObjectNotRegisteredException,
                   AllocationException;

        Object* allocateObjectByType(string type)
            throws AllocationException;

        void releaseObject(Ice::Identity id)
            throws ObjectNotRegisteredException,
                   AllocationException;

        idempotent void setAllocationTimeout(int timeout);
    };
};
```

The client is responsible for keeping the session alive by periodically invoking `keepAlive`, as discussed in Section 38.11.1.

The `allocateObjectById` operation allocates and returns the proxy for the allocatable object with the given identity. If no allocatable object with the given identity is registered, the client receives `ObjectNotRegisteredException`. If the object cannot be allocated, the client receives `AllocationException`. An allocation attempt can fail for the following reasons:

- the object is already allocated by the session
- the object is allocated by another session and did not become available during the configured allocation timeout period
- the session was destroyed.

The `allocateObjectByType` operation allocates and returns a proxy for an allocatable object registered with the given type. If more than one allocatable object is registered with the given type, the registry selects one at random. The client receives `AllocationException` if no objects with the given type could be allocated. An allocation attempt can fail for the following reasons:

- no objects are registered with the given type
- all objects with the given type are already allocated (either by this session or other sessions) and none became available during the configured allocation timeout period
- the session was destroyed.

The `releaseObject` operation releases an object allocated by the session. The client receives `ObjectNotRegisteredException` if no allocatable object is registered with the given identity and `AllocationException` if the object is not allocated by the session. Upon session destruction, IceGrid automatically releases all allocated objects.

The `setAllocationTimeout` operation configures the timeout used by the allocation operations. If no allocatable objects are available when the client invokes `allocateObjectById` or `allocateObjectByType`, IceGrid waits for the specified timeout period for an allocatable object to become available. If the timeout expires, the client receives `AllocationTimeoutException`.

38.11.4 Allocating Servers

A client does not need to explicitly allocate a server. If a server is allocatable, IceGrid implicitly allocates it to the first client that claims one of the server's allocatable objects. Likewise, IceGrid releases the server when all of its allocatable objects are released.

Server allocation is useful in two situations:

- Only allocatable servers can use the `session` activation mode, in which the server is activated on demand when allocated by a client and deactivated upon release.

- An allocatable server can be secured with IceSSL or Glacier2 so that its objects can only be invoked by the client that allocated it.

38.11.5 Security

IceGrid's resource allocation facility allows clients to coordinate access to objects and servers but does not place any restrictions on client invocations to allocated objects; any client that has a proxy for an allocated object could conceivably invoke an operation on it. IceGrid assumes that clients are cooperating with each other and respecting allocation semantics.

To prevent unauthorized clients from invoking operations on an allocated object or server, you can use IceSSL or Glacier2:

- Using IceSSL, you can secure access to a server or a particular object adapter with the properties `IceSSL.TrustOnly.Server` or `IceSSL.TrustOnly.Server.AdapterName`.

  For example, if you configure a server with the session activation mode, you can set one of the `IceSSL.TrustOnly` properties to the `${session.id}` variable, which is substituted with the session id when the server is activated for the session. If the IceGrid session was created from a secure connection, the session id will be the distinguished name associated with the secure connection, which effectively restricts access to the server or one of its adapters to the client that established the session with IceGrid.
- With Glacier2, you can secure access to an allocated object or the object adapters of an allocated server with the Glacier2 filtering mechanism (see Section 42.6.2). By default, IceGrid sessions created with a Glacier2 router are automatically given access to allocated objects, allocatable objects, certain well-known objects, and the object adapters of allocated servers. See Section 38.15 for more information.

38.11.6 Descriptors

Allocatable objects are registered using a descriptor that is similar to well-known object descriptors (see Section Section 38.17.14). Allocatable objects cannot be replicated and therefore can only be specified within an object adapter descriptor.

Servers can be specified as allocatable by setting the server descriptor's `allocatable` attribute.

As an example, the following application defines an allocatable server and an allocatable object:

```
<icegrid>
    <application name="Ripper">
        <node name="Node1">
            <server id="EncoderServer"
                exe="/opt/ripper/bin/server"
                activation="on-demand"
                allocatable="true">
                <adapter name="EncoderAdapter"
                    id="EncoderAdapter"
                    endpoints="tcp">
                    <allocatable identity="EncoderFactory"
                        type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </node>
    </application>
</icegrid>
```

38.11.7 Application Changes

We can use the allocation facility in our MP3 encoder factory to coordinate access to the MP3 encoder factories. First we need to modify the descriptors to define an allocatable object:

```
<icegrid>
    <application name="Ripper">
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <server id="EncoderServer${index}"
                exe="/opt/ripper/bin/server"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    endpoints="tcp">
                    <allocatable identity="EncoderFactory${index}"
                        type="::Ripper::MP3EncoderFactory"/>
                </adapter>
            </server>
        </server-template>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
                index="1"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
```

```
                index="2"/>
        </node>
    </application>
</icegrid>
```

Next, the client needs to create a session and allocate a factory:

```
Ice::ObjectPrx obj = session->allocateObjectByType(
    Ripper::MP3EncoderFactory::ice_staticId());
try {
    Ripper::MP3EncoderPrx encoder = factory->createEncoder();
    // Use the encoder to encode a file ...
}
catch (const Ice::LocalException & ex) {
    // There was a problem with the encoding, we catch the
    // exception to make sure we release the factory.
}
session->releaseObject(obj->ice_getIdentity());
```

It is important to release an allocated object when it is no longer needed so that other clients may use it. If you forget to release an object, it remains allocated until the session is destroyed.

38.12 Registry Replication

The failure of an IceGrid registry or registry host can have serious consequences. A client can continue to use an existing connection to a server without interruption, but any activity that requires interaction with the registry is vulnerable to a single point of failure. As a result, the IceGrid registry supports replication using a master-slave configuration to provide high availability for applications that require it.

38.12.1 Overview

In IceGrid's registry replication architecture, there is one master replica and any number of slave replicas. The master synchronizes its deployment information with the slaves so that any replica is capable of responding to locate requests, managing nodes, and starting servers on demand. Should the master registry or its host fail, properly configured clients transparently fail over to one of the slaves.

Each replica has a unique name. The name `Master` is reserved for the master replica, while replicas can use any name that can legally appear in an object identity.

Figure 38.6 illustrates the underlying concepts of registry replication:

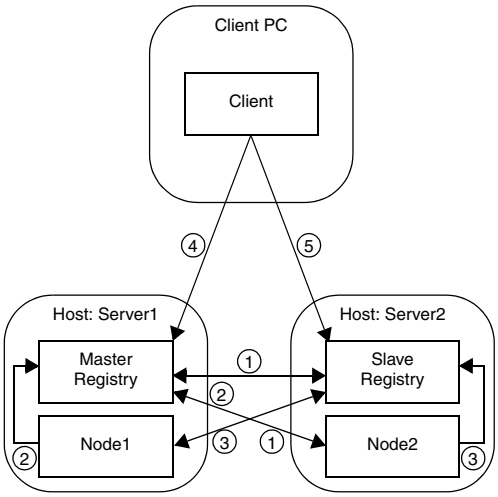


Figure 38.6. Overview of registry replication.

1. The slave replica contacts the master replica at startup and synchronizes its databases. Any subsequent modifications to the deployed applications are made via the master replica, which distributes them to all active slaves.
2. The nodes contact the master replica at startup to notify it about their availability.
3. The master replica provides a list of slave replicas to the nodes so that the nodes can also notify the slaves.
4. The client's configuration determines which replica it contacts initially. In this example, it contacts the master replica.
5. In the case of a failure, the client automatically fails over to the slave. If the master registry's host has failed, then `Node1` and any servers that were active on this host also become unavailable. The use of object adapter replication (see Section 38.9) allows the client to transparently reestablish communication with a server on `Node2`.

38.12.2 Replica Capabilities

A master registry replica has a number of responsibilities, only some of which are supported by slaves. The master replica knows all of its slaves, but the slaves are not in contact with one another. If the master replica fails, the slaves can perform several vital functions that should keep most applications running without interruption. Eventually, however, a new master replica must be started to restore full registry functionality. For a slave replica to become the master, the slave must be restarted.

Locate Requests

One of the most important functions of a registry replica is responding to locate requests from clients, and every replica has the capability to service these requests. Slaves synchronize their databases with the master so that they have all of the information necessary to transform object identities, object adapter identifiers, and replica group identifiers into an appropriate set of endpoints.

Server Activation

Nodes establish sessions with each active registry replica so that any of the replicas are capable of activating a server on behalf of a client.

Queries

Replicating the registry also replicates the object that supports the `IceGrid::Query` interface (see Section 38.6.5). A client that resolves the `IceGrid/Query` object identity receives the endpoints of all active replicas, any of which can execute the client's requests.

Allocation

A client that needs to allocate a resource must establish a session with the master replica.

Administration

The state of an IceGrid registry is accessible via the `IceGrid::Admin` interface or (more commonly) using an administrative tool that encapsulates this interface. Modifications to the registry's state, such as deploying or updating an application, can only be done using the master replica. Administrative access to slave replicas is allowed but restricted to read-only operations. The administrative utilities provide mechanisms for you to select a particular replica to contact.

For programmatic access to a replica's administrative interface, the `IceGrid/Registry` identity corresponds to the master replica and the identity `IceGrid/Registry-`*`name`* corresponds to the slave with the given name.

Glacier2 Support

The registry implements the session manager interfaces required for integration with a Glacier2 router (see Section 38.15). The master replica supports the object identities `IceGrid/SessionManager` and `IceGrid/AdminSession-Manager`. The slave replicas offer support for read-only administrative sessions using the object identity `IceGrid/AdminSessionManager-`*`name`*.

38.12.3 Configuration

Incorporating registry replication into an application is primarily accomplished by modifying your IceGrid configuration settings.

Replicas

Each replica must specify a unique name in the configuration property `IceGrid.Registry.ReplicaName`. The default value of this property is `Master`, therefore the master replica can omit this property if desired.

At startup, a slave replica attempts to register itself with its master in order to synchronize its databases and obtain the list of active nodes. The slave uses the proxy supplied by the `Ice.Default.Locator` property to connect to the master, therefore this proxy must be defined and contain at least the endpoint of the master replica.

For better reliability if a failure occurs, we recommend that you also include the endpoints of all slave replicas in the `Ice.Default.Locator` property. There is no harm in adding the slave's own endpoints to the proxy in `Ice.Default.Locator`; in fact, it makes configuration simpler because all of the slaves can share the same property definition. Although slaves do not communicate with each other, it is possible for one of the slaves to be promoted to the master, therefore supplying the endpoints of all slaves minimizes the chance of a communication failure.

Shown below is an example of the configuration properties for a master replica:

```
IceGrid.InstanceName=DemoIceGrid
IceGrid.Registry.Client.Endpoints=default -p 12000
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.Data=db/master
...
```

You can configure a slave replica to use this master with the following settings:

```
Ice.Default.Locator=DemoIceGrid/Locator:default -p 12000
IceGrid.Registry.Client.Endpoints=default -p 12001
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.Data=db/replica1
IceGrid.Registry.ReplicaName=Replica1
...
```

Clients

The endpoints contained in the `Ice.Default.Locator` property determine which registry replicas the client can use when issuing locate requests. If high availability is important, this property should include the endpoints of at least two (and preferably all) replicas. Not only does this increase the reliability of the client, it also distributes the work load of responding to locate requests among all of the replicas.

Continuing the example from the previous section, you can configure a client with the `Ice.Default.Locator` property as shown below:

```
Ice.Default.Locator=\
    DemoIceGrid/Locator:default -p 12000:default -p 12001
```

Nodes

As with slave replicas and clients, an IceGrid node should be configured with an `Ice.Default.Locator` property that contains the endpoints of all replicas. Doing so allows a node to notify each of them about its presence, thereby enabling the replicas to activate its servers and obtain the endpoints of its object adapters.

The following properties demonstrate how to configure a node with a replicated registry:

```
Ice.Default.Locator=\
    DemoIceGrid/Locator:default -p 12000:default -p 12001
IceGrid.Node.Name=node1
IceGrid.Node.Endpoints=default
IceGrid.Node.Data=db/node1
```

Diagnostics

You can use several configuration properties to enable trace messages that may help in diagnosing registry replication issues:

- `IceGrid.Registry.Trace.Replica`

  Displays information about the sessions established between master and slave replicas.

- `IceGrid.Registry.Trace.Node`
  `IceGrid.Node.Trace.Replica`

  Displays information about the sessions established between replicas and nodes.

38.13 Application Distribution

In the chapter so far, "deployment" has meant the creation of descriptors in the registry. A broader definition involves a number of other tasks:

- Writing IceGrid configuration files and preparing data directories on each computer
- Installing the IceGrid binaries and dependent libraries on each computer
- Starting the registry and/or node on each computer, and possibly configuring the systems to launch them automatically
- Distributing your server executables, dependent libraries and supporting files to the appropriate nodes.

The first three tasks are the responsibility of the system administrator, but IceGrid can help with the fourth. Using an IcePatch2 server (see Chapter 45), you can

configure the nodes to download servers automatically and patch them at any time. Figure 38.7 shows the interactions of the components.

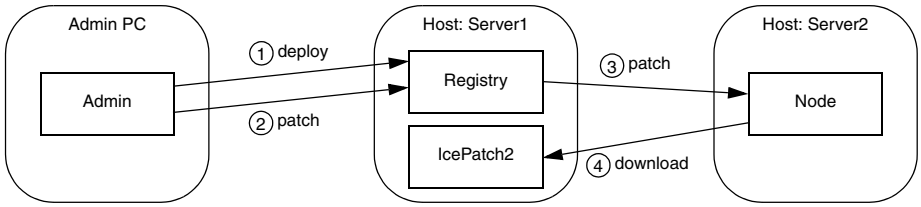


Figure 38.7. Overview of application distribution.

As you can see, deploying an IceGrid application has greater significance when IcePatch2 is also involved. After deployment, the administrative tool initiates a patch, causing the registry to notify all active nodes that are configured for application distribution to begin the patching process. Since each IceGrid node is an IcePatch2 client, the node performs the patch just like any IcePatch2 client: it downloads everything if no local copy of the distribution exists, otherwise it does an incremental patch in which it downloads only new files and those whose signatures have changed.

The benefits of this feature are clear:

- The distribution files are maintained in a central location
- Updating a distribution on all of the nodes is a simple matter of preparing the master distribution and letting IceGrid do the rest
- Manually transferring executables and supporting files to each computer is avoided, along with the mistakes that manual intervention sometimes introduces.

38.13.1 Deploying an IcePatch2 Server

If you plan to use IceGrid's distribution capabilities, we generally recommend deploying an IcePatch2 server along with your application. Doing so gives you the same benefits as any other IceGrid server, including on-demand activation and remote administration. We'll only use one server in our sample application, but you might consider replicating a number of IcePatch2 servers in order to balance the patching load for large distributions.

Patching Considerations

Deploying an IcePatch2 server with your application presents a chicken-and-egg dilemma: how do the nodes download their distributions if the IcePatch2 server is included in the deployment? To answer this question, we need to learn more about IceGrid's behavior.

Deploying and patching are treated as two separate steps: first you deploy the application, then you initiate the patching process. The **`icegridadmin`** utility combines these steps into one command (**`application add`**), but also provides an option to disable the patching step if so desired.

Let's consider the state of the application after deployment but before patching: we have described the servers that run on each node, including file system-dependent attributes such as the pathnames of their executables and default working directories. If these pathnames refer to directories in the distribution, and the distribution has not yet been downloaded to that node, then clearly we cannot attempt to use those servers until patching has completed. Similarly, we cannot deploy an IcePatch2 server whose executable resides in the distribution to be downloaded[2].

For these reasons, we assume that the IcePatch2 server and supporting libraries are distributed by the system administrator along with the IceGrid registry and nodes to the appropriate computers. The server should be configured for on-demand activation so that its node starts it automatically when patching begins. If the server is configured for manual activation, you must start it prior to patching.

Server Template

The Ice distribution includes an IcePatch2 server template that simplifies the inclusion of IcePatch2 in your application. The relevant portion from the file `config/templates.xml` is shown below:

```
<server-template id="IcePatch2">
    <parameter name="instance-name"
        default="${application}.IcePatch2"/>
    <parameter name="endpoints" default="default"/>
    <parameter name="directory"/>

    <server id="${instance-name}" exe="icepatch2server"
```

2. We are ignoring the case where a temporary IcePatch2 server is used to bootstrap other IcePatch2 servers.

```
        application-distrib="false" activation="on-demand">
        <adapter name="IcePatch2" endpoints="${endpoints}">
            <object identity="${instance-name}/server"
                type="::IcePatch2::FileServer"/>
        </adapter>
        <adapter name="IcePatch2.Admin" id=""
            endpoints="tcp -h 127.0.0.1"/>
        <property name="IcePatch2.InstanceName"
            value="${instance-name}"/>
        <property name="IcePatch2.Directory"
            value="${directory}"/>
    </server>
</server-template>
```

Notice that the server's pathname is `icepatch2server`, meaning the program must be present in the node's executable search path. The only mandatory parameter is `directory`, which specifies the server's data directory and becomes the value of the `IcePatch2.Directory` property. The value of the `instance-name` parameter is used as the server's identifier when the template is instantiated; its default value includes the name of the application in which the template is used. This identifier also affects the identities of the two well-known objects declared by the server (see Section 45.6).

Consider the following sample application:

```
<icegrid>
    <application name="PatchDemo">
        <node name="Node">
            <server-instance template="IcePatch2"
                directory="/opt/icepatch2/data"/>
            ...
        </node>
    </application>
</icegrid>
```

Instantiating the `IcePatch2` template creates a server identified as `PatchDemo.IcePatch2` (as determined by the default value for the `instance-name` parameter). The well-known objects use this value as the category in their identities, such as `PatchDemo.IcePatch2/server`.

In order to refer to the `IcePatch2` template in your application, you must have already configured the registry to use the `config/templates.xml` file as your default templates (see Section 38.7.4), or copied the template into the XML file describing your application.

38.13.2 Distribution Descriptor

A distribution descriptor provides the details that a node requires in order to download the necessary files. Specifically, the descriptor supplies the proxy of the IcePatch2 server and the names of the subdirectories comprising the distribution, all of which are optional. If the descriptor does not define the proxy, the following default value is used instead:

```
${application}.IcePatch2/server
```

You may recall that this value matches the default identity configured by the IcePatch2 server template described in Section 38.13.1. Also notice that this is an indirect proxy, implying that the IcePatch2 server was deployed with the application and can be started on-demand if necessary.

If the descriptor does not select any subdirectories, the node downloads the entire contents of the IcePatch2 data directory.

In XML, a descriptor having the default behavior as described above can be written as shown below:

```
<distrib/>
```

To specify a proxy, use the `icepatch` attribute:

```
<distrib icepatch="PatchDemo.IcePatch2/server"/>
```

Finally, select subdirectories using a nested element:

```
<distrib>
    <directory>dir1</directory>
    <directory>dir2/subdir</directory>
</distrib>
```

By including only certain subdirectories in a distribution, you are minimizing the time and effort required to download and patch each node. For example, each node in a heterogeneous network might download a platform-specific subdirectory and another subdirectory containing files common to all platforms.

38.13.3 Application and Server Distributions

A distribution descriptor can be used in two contexts: within an application, and within a server. When the descriptor appears at the application level, it means every node in the application downloads that distribution. This is useful for distributing files required by all of the nodes on which servers are deployed, especially in a grid of homogeneous computers where it would be tedious to repeat the

same distribution information in each server descriptor. Here is a simple XML example:

```
<icegrid>
    <application name="PatchDemo">
        <distrib>
            <directory>Common</directory>
        </distrib>
        ...
    </application>
</icegrid>
```

At the server level, a distribution descriptor downloads the specified directories for the private use of the server:

```
<icegrid>
    <application name="PatchDemo">
        <distrib>
            <directory>Common</directory>
        </distrib>
        <node name="Node">
            <server id="SimpleServer" ...>
                <distrib>
                    <directory>ServerFiles</directory>
                </distrib>
            </server>
        </node>
    </application>
</icegrid>
```

When a distribution descriptor is defined at both the application and server levels, as shown in the previous example, IceGrid assumes that a dependency relationship exists between the two unless the server is configured otherwise. IceGrid checks this dependency before patching a server; if the server is dependent on the application's distribution, IceGrid patches the application's distribution first, and then proceeds to patch the server's. You can disable this dependency by modifying the server's descriptor:

```
<icegrid>
    <application name="PatchDemo">
        <distrib>
            <directory>Common</directory>
        </distrib>
        <node name="Node">
            <server id="SimpleServer" application-distrib="false"
                ...>
```

```
                <distrib>
                    <directory>ServerFiles</directory>
                </distrib>
            </server>
        </node>
    </application>
</icegrid>
```

Setting the `application-distrib` attribute to `false` informs IceGrid to consider the two distributions independent of one another.

38.13.4 Server Integrity

Before an IceGrid node begins patching a distribution, it ensures that all relevant servers are shut down and prevents them from reactivating until patching completes. For example, the node disables all of the servers whose descriptors declare a dependency on the application distribution (see Section 38.13.3).

38.13.5 Using Distributions

The node stores application and server distributions in its data directory. The pathnames of the distributions are represented by reserved variables that you can use in your descriptors:

- `application.distrib`

  This variable can be used within server descriptors to refer to the top-level directory of the application distribution.

- `server.distrib`

  The value of this variable is the top-level directory of a server distribution. It can be used only within a server descriptor that has a distribution.

The XML example shown below illustrates the use of these variables:

```
<icegrid>
    <application name="PatchDemo">
        <distrib>
            <directory>Common</directory>
        </distrib>
        <node name="Node">
            <server id="Server1"
                exe="${application.distrib}/Common/Bin/Server1"
                ...>
            </server>
```

```
            <server id="Server2"
                exe="${server.distrib}/Server2Files/Bin/Server2"
                ...>
                <option>-d</option>
                <option>${server.distrib}/Server2Files</option>
                <distrib>
                    <directory>Server2Files</directory>
                </distrib>
            </server>
        </node>
    </application>
</icegrid>
```

Notice that the descriptor for `Server2` supplies the server's distribution directory as command-line options.

For more information on variables, see Section 38.18.

38.13.6 Application Changes

Adding an application distribution to our ripper example requires two minor changes to our descriptors from Section 38.9.3:

```
<icegrid>
    <application name="Ripper">
        <replica-group id="EncoderAdapters">
            <load-balancing type="adaptive"/>
            <object identity="EncoderFactory"
                type="::Ripper::MP3EncoderFactory"/>
        </replica-group>
        <server-template id="EncoderServerTemplate">
            <parameter name="index"/>
            <parameter name="exepath"
                default="/opt/ripper/bin/server"/>
            <server id="EncoderServer${index}"
                exe="${exepath}"
                activation="on-demand">
                <adapter name="EncoderAdapter"
                    replica-group="EncoderAdapters"
                    endpoints="tcp"/>
            </server>
        </server-template>
        <distrib/>
        <node name="Node1">
            <server-instance template="EncoderServerTemplate"
                index="1"/>
```

```
            <server-instance template="IcePatch2"
                directory="/opt/ripper/icepatch"/>
        </node>
        <node name="Node2">
            <server-instance template="EncoderServerTemplate"
                index="2"/>
        </node>
    </application>
</icegrid>
```

An application distribution is sufficient for this example because we are deploying the same server on each node. We have also deployed an IcePatch2 server on `Node1` using the template described in Section 38.13.1.

38.14 Administrative Sessions

To access IceGrid's administrative facilities from a program, you must first establish an administrative session. Once done, a wide range of services are at your disposal, including the manipulation of IceGrid registries, nodes, and servers; deployment of new components such as well-known objects; and dynamic monitoring of IceGrid events.

Note that an administrative session can be established with either the master or a slave registry replica, but a session with a slave replica is restricted to read-only operations; see Section 38.12 for more information.

38.14.1 Creating an Administrative Session

The `Registry` interface provides two operations for creating an administrative session:

```
module IceGrid {
    exception PermissionDeniedException {
        string reason;
    };

    interface Registry {
        AdminSession* createAdminSession(string userId,
                                         string password)
            throws PermissionDeniedException;

        AdminSession* createAdminSessionFromSecureConnection()
```

```
            throws PermissionDeniedException;

        idempotent int getSessionTimeout();

        // ...
    };
};
```

The `createAdminSession` operation expects a username and password and returns a session proxy if the client is allowed to create a session. By default, IceGrid does not allow the creation of administrative sessions. You must define the property `IceGrid.Registry.AdminPermissionsVerifier` with the proxy of a permissions verifier object to enable session creation with `createAdminSession`. The verifier object must implement the interface `Glacier2::PermissionsVerifier` (see Section 42.6.1).

The `createAdminSessionFromSecureConnection` operation does not require a username and password because it uses the credentials supplied by an SSL connection to authenticate the client (see Chapter 41). As with `createAdminSession`, you must configure the proxy of a permissions verifier object before clients can use `createAdminSessionFromSecureConnection` to create a session. In this case, the `IceGrid.Registry.AdminSSLPermissionsVerifier` property specifies the proxy of a verifier object that implements the interface `Glacier2::SSLPermissionsVerifier` (see Section 42.6.1).

As an example, the following code demonstrates how to obtain a proxy for the registry and invoke `createAdminSession`:

```
Ice::ObjectPrx base =
    communicator->stringToProxy("IceGrid/Registry");
IceGrid::RegistryPrx registry =
    IceGrid::RegistryPrx::checkedCast(base);
string username = ...;
string password = ...;
IceGrid::AdminSessionPrx session;
try {
    session = registry->createAdminSession(username, password);
} catch (const IceGrid::PermissionDeniedException & ex) {
    cout << "permission denied:\n" << ex.reason << endl;
}
```

The `AdminSession` interface provides operations for accessing log files (see Section 38.14.2) and establishing observers (see Section 38.14.3). Furthermore, two additional operations are worthy of your attention:

```
module IceGrid {
    interface AdminSession extends Glacier2::Session {
        idempotent void keepAlive();
        idempotent Admin* getAdmin();
        // ...
    };
};
```

If your program uses an administrative session indefinitely, you must prevent the session from expiring by invoking `keepAlive` periodically. You can determine the timeout period by calling `getSessionTimeout` on the `Registry` interface (see Section 38.11.1). Typically a program uses a dedicated thread for this purpose.

The `getAdmin` operation returns a proxy for the `IceGrid::Admin` interface, which provides complete access to the registry's settings. For this reason, you must use extreme caution when enabling administrative sessions.

38.14.2 Log Files

IceGrid's `AdminSession` interface provides operations for remotely accessing the log files of a registry, node, or server:

```
module IceGrid {
interface AdminSession extends Glacier2::Session {
    // ...

    FileIterator* openServerLog(string id, string path, int count)
        throws FileNotAvailableException, ServerNotExistException,
               NodeUnreachableException, DeploymentException;
    FileIterator* openServerStdErr(string id, int count)
        throws FileNotAvailableException, ServerNotExistException,
               NodeUnreachableException, DeploymentException;
    FileIterator* openServerStdOut(string id, int count)
        throws FileNotAvailableException, ServerNotExistException,
               NodeUnreachableException, DeploymentException;

    FileIterator* openNodeStdErr(string name, int count)
        throws FileNotAvailableException, NodeNotExistException,
               NodeUnreachableException;
    FileIterator* openNodeStdOut(string name, int count)
        throws FileNotAvailableException, NodeNotExistException,
               NodeUnreachableException;

    FileIterator* openRegistryStdErr(string name, int count)
        throws FileNotAvailableException,
```

```
                RegistryNotExistException,
                RegistryUnreachableException;
    FileIterator * openRegistryStdOut(string name, int count)
        throws FileNotAvailableException,
                RegistryNotExistException,
                RegistryUnreachableException;
};
};
```

In order to access the text of a program's standard output or standard error log, you must configure it using the `Ice.StdOut` and `Ice.StdErr` properties, respectively (see Section D.11). For registries and nodes, you must define these properties explicitly but, for servers, the node defines these properties automatically if the property `IceGrid.Node.Output` is defined (see Section D.15), causing the server's output to be logged in individual files.

If `IceGrid.Node.Output` is *not* defined, the following rules apply:

- If the node is started from a console or shell, servers share the node's `stdout` and `stderr`. If `Ice.StdOut` and/or `Ice.StdErr` properties are defined for the node, the servers' output is redirected to the specified files as well.
- If the node is started as a Unix daemon and **`--noclose`** is not used, the servers' output is lost, except if `Ice.StdOut` and/or `Ice.StdErr` properties are set for node, in which case the servers' output is redirected to the specified files.
- If the node is started as a Windows service, the servers' output is lost even if `Ice.StdOut` and/or `Ice.StdErr` are set.

Log messages from the node itself are sent to `stderr` unless you set `Ice.Syslog` (for Unix). If the node is started as a Windows service, its log messages always are sent to the Windows event log.

In the case of `openServerLog`, the value of the `path` argument must resolve to the same file as one of the server's `log` descriptors (see Section 38.17.12). This security measure prevents a client from opening an arbitrary file on the server's host.

All of the operations accept a `count` argument and return a proxy to a `FileIterator` object. The `count` argument determines where to start reading the log file: if the value is negative, the iterator is positioned at the beginning of the file, otherwise the iterator is positioned to return the last `count` lines of text.

The `FileIterator` interface is quite simple:

```
module IceGrid {
interface FileIterator {
    bool read(int size, out Ice::StringSeq lines)
        throws FileNotAvailableException;
    void destroy();
};
};
```

A client may invoke the `read` operation as many times as necessary. The `size` argument specifies the maximum number of bytes that `read` can return; the client must not use a size that would cause the reply to exceed the client's configured maximum message size (see the property `Ice.MessageSizeMax` in Section D.11).

If this is the client's first call to `read`, the `lines` argument holds whatever text was available from the iterator's initial position, and the iterator is repositioned in preparation for the next call to `read`. The operation returns false to indicate that more text is available and true if all available text has been read.

Line termination characters are removed from the contents of `lines`. When displaying the text, you must be aware that the first and last elements of the sequence can be partial lines. For example, the last line of the sequence might be incomplete if the limit specified by `size` is reached. The next call to `read` returns the remainder of that line as the first element in the sequence.

As an example, the C++ code below displays the contents of a log file and waits for new text to become available:

```
IceGrid::FileIteratorPrx iter = ...;
while(true) {
    Ice::StringSeq lines;
    bool end = iter->read(10000, lines);
    if (!lines.empty()) {
        // The first line might be a continuation from
        // the previous call to read.
        cout << lines[0];
        for (Ice::StringSeq::const_iterator p =
             ++lines.begin(); p != lines.end(); ++p)
            cout << endl << *p << flush;
    }
    if (end)
        sleep(1);
}
```

Notice that the loop includes a delay in case `read` returns true, which prevents the client from entering a busy loop when no data is currently available.

The client should call `destroy` when the iterator object is no longer required. At the time the client's session terminates, IceGrid reclaims any iterators that were not explicitly destroyed.

If the client waits indefinitely for new data, it must invoke `keepAlive` on the administrative session to prevent it from expiring (see Section 38.14.1).

38.14.3 Dynamic Monitoring

IceGrid allows an application to monitor relevant state changes by registering callback objects. (The IceGrid GUI tool uses these callback interfaces for its implementation.) The callback interfaces are useful to, for example, automatically generate an email notification when a node goes down or some other state change of interest occurs.

The Observer Interfaces

IceGrid offers a callback interface for each major component of the IceGrid architecture:

```
module IceGrid {
interface NodeObserver {
    void nodeInit(NodeDynamicInfoSeq nodes);
    void nodeUp(NodeDynamicInfo node);
    void nodeDown(string name);
    void updateServer(string node, ServerDynamicInfo updatedInfo);
    void updateAdapter(string node,
                       AdapterDynamicInfo updatedInfo);
};

interface ApplicationObserver {
    void applicationInit(int serial,
                         ApplicationInfoSeq applications);
    void applicationAdded(int serial, ApplicationInfo desc);
    void applicationRemoved(int serial, string name);
    void applicationUpdated(int serial,
                            ApplicationUpdateInfo desc);
};

interface AdapterObserver {
    void adapterInit(AdapterInfoSeq adpts);
    void adapterAdded(AdapterInfo info);
    void adapterUpdated(AdapterInfo info);
    void adapterRemoved(string id);
};
```

```
interface ObjectObserver {
    void objectInit(ObjectInfoSeq objects);
    void objectAdded(ObjectInfo info);
    void objectUpdated(ObjectInfo info);
    void objectRemoved(Ice::Identity id);
};

interface RegistryObserver {
    void registryInit(RegistryInfoSeq registries);
    void registryUp(RegistryInfo node);
    void registryDown(string name);
};
};
```

The next section describes how to install an observer.

Registering Observers

The `AdminSession` interface provides two operations for registering your observers:

```
module IceGrid {
    interface AdminSession extends Glacier2::Session {
        idempotent void keepAlive();

        idempotent void setObservers(
                RegistryObserver* registryObs,
                NodeObserver* nodeObs,
                ApplicationObserver* appObs,
                AdapterObserver* adptObs,
                ObjectObserver* objObs)
            throws ObserverAlreadyRegisteredException;

        idempotent void setObserversByIdentity(
                Ice::Identity registryObs,
                Ice::Identity nodeObs,
                Ice::Identity appObs,
                Ice::Identity adptObs,
                Ice::Identity objObs)
            throws ObserverAlreadyRegisteredException;

        // ...
    };
};
```

You should invoke `setObservers` and supply proxies when it is possible for the registry to establish a separate connection to the client to deliver its callbacks. If network restrictions such as firewalls prevent such a connection, you should use the `setObserversByIdentity` operation, which creates a bidirectional connection instead (see Section 36.7).

You can pass a null proxy for any parameter to `setObservers`, or an empty identity for any parameter to `setObserversByIdentity`, if you want to use only some of the observers. In addition, passing a null proxy or an empty identity for an observer cancels a previous registration of that observer. The operations raise `ObserverAlreadyRegisteredException` if you pass a proxy or identity that was registered in a previous call.

Once the observers are registered, operations corresponding to state changes will be invoked on the observers. (See the online Slice API Reference for details on the data passed to the observers. You can also look at the source code for the IceGrid GUI implementation in the Ice for Java distribution to see how observers are used by the GUI.)

Finally, remember to invoke `keepAlive` periodically to prevent the session from expiring (see Section 38.14.1).

38.15 Glacier2 Integration

This section provides information on integrating a Glacier2 router (see Chapter 42) into your IceGrid environment.

38.15.1 Configuration Requirements

A typical IceGrid client must be configured with a locator proxy (see Section 38.4.3), but the configuration requirements change when the client

accesses the location service indirectly via a Glacier2 router as shown in Figure 38.8.

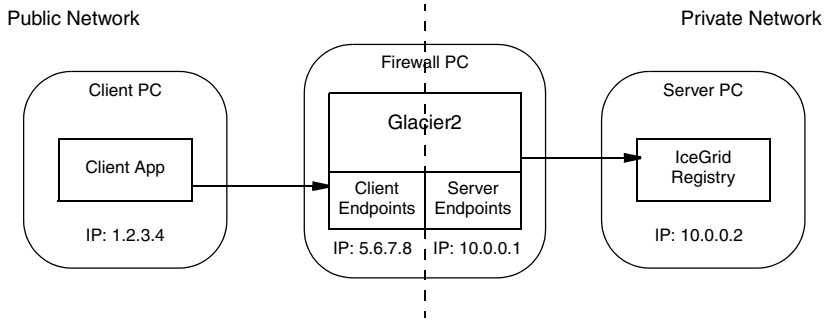


Figure 38.8. Using IceGrid via a Glacier2 router.

In this situation, it is the router that must be configured with a locator proxy.

Assuming the registry's client endpoint in Figure 38.8 uses port `8000`, the router requires the following configuration property:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h 10.0.0.2 -p 8000
```

Fortunately, the node supplies this property when it starts the router, so there is no need to configure it explicitly. Note that all of the router's clients use the same locator.

38.15.2 Remote Administration

If you intend to administer IceGrid remotely via a Glacier2 router, you must define one of the following properties (or both), depending on whether you use user name and password authentication or a secure connection:

```
Glacier2.SessionManager=IceGrid/AdminSessionManager
Glacier2.SSLSessionManager=IceGrid/AdminSSLSessionManager
```

These session managers are accessible via the registry's administrative session manager endpoints (see Section 38.21.1), so the Glacier2 router must be authorized to establish a connection to these endpoints. Note that you must secure these endpoints, otherwise arbitrary clients can manipulate the session managers. An administrative session is allowed to access any object by default. To restrict access to the `IceGrid::AdminSession` object and the `IceGrid::Admin` object that is

returned by the session's `getAdmin` operation, you must set the property `IceGrid.Registry.AdminSessionFilters` to one (see page 1911).

38.15.3 Allocating Servers and Objects

To allocate servers and objects, a program can establish a client session via Glacier2. Depending on the authentication method, one or both of the following properties must be set in the Glacier2 configuration:

```
Glacier2.SessionManager=IceGrid/SessionManager
Glacier2.SSLSessionManager=IceGrid/SSLSessionManager
```

These session managers are accessible via the registry's session manager endpoints, so the Glacier2 router must be authorized to establish a connection to these endpoints.

A client session is allowed to access any object by default. To restrict access to the `IceGrid::Session` and `IceGrid::Query` objects, you must set the property `IceGrid.Registry.SessionFilters` to one (see page 1916). However, you can use the allocation mechanism to access additional objects and adapters. IceGrid adds an identity filter when a client allocates an object and removes that filter again when the object is released. When a client allocates a server, IceGrid adds an adapter identity filter for the server's indirect adapters and removes that filter again when the server is released.

38.15.4 Session Management

Providing access to administrative sessions (Section 38.15.2) and client sessions (Section 38.15.3) both require that you define at least one of the properties `Glacier2.SessionManager` and `Glacier2.SSLSessionManager`, which presents a potential problem if you intend to access both types of sessions via the same Glacier2 router.

The simplest solution is to dedicate a router instance to each type of session. However, if you need to access both types of sessions from a single router, you can accomplish it only if you use a different authentication mechanism for each type of session. For example, you can configure the router as follows:

```
Glacier2.SessionManager=IceGrid/SessionManager
Glacier2.SSLSessionManager=IceGrid/AdminSSLSessionManager
```

This configuration uses user name and password authentication for client sessions, and SSL authentication for administrative sessions. If this restriction is too limiting, you must use two router instances.

38.15.5 Deploying a Router

The Ice distribution includes default server templates for Ice services such as IcePatch2 (see Section 38.13.1) and Glacier2 that simplify the task of deploying these servers in an IceGrid domain.

The relevant portion from the file `config/template.xml` is shown below:

```
<server-template id="Glacier2">
    <parameter name="instance-name"
               default="${application}.Glacier2"/>
    <parameter name="client-endpoints"/>
    <parameter name="server-endpoints"/>
    <parameter name="session-timeout" default="0"/>

    <server id="${instance-name}" exe="glacier2router">
    <properties>
        <property name="Glacier2.Client.Endpoints"
                  value="${client-endpoints}"/>
        <property name="Glacier2.Server.Endpoints"
                  value="${server-endpoints}"/>
        <property name="Glacier2.Admin.Endpoints"
                  value="tcp -h 127.0.0.1"/>
        <property name="Glacier2.Admin.RegisterProcess"
                  value="1"/>
        <property name="Glacier2.InstanceName"
                  value="${instance-name}"/>
        <property name="Glacier2.SessionTimeout"
                  value="${session-timeout}"/>
    </properties>
</server-template>
```

Notice that the server's pathname is `glacier2router`, meaning the program must be present in the node's executable search path. Another important point is the server's activation mode: it uses manual activation (the default), meaning the router must be started manually. This requirement becomes clear when you consider that the router is the point of contact for remote clients; if the router is not running, there is no way for a client to contact the locator and cause the router to be started on-demand.

The template defines only a few properties; if you want to set additional properties, you can define them in the server instance property set.

Of interest is the `instance-name` parameter, which allows you to configure the `Glacier2.InstanceName` property. The parameter's default value includes the name of the application in which the template is used. This parameter also affects the identities of the objects implemented by the router (see Section 42.3.5).

Consider the following sample application:

```
<icegrid>
    <application name="Glacier2Demo">
        <node name="Node">
            <server-instance template="Glacier2"
                client-endpoints="tcp -h 5.6.7.8 -p 8000"
                session-timeout="300"
                server-endpoints="tcp -h 10.0.0.1"/>
            ...
        </node>
    </application>
</icegrid>
```

Instantiating the `Glacier2` template creates a server identified as `Glacier2Demo.Glacier2` (as determined by the default value for the `instance-name` parameter). The router's objects use this value as the category in their identities, such as `Glacier2Demo.Glacier2/router`. The router proxy used by clients must contain a matching identity.

In order to refer to the `Glacier2` template in your application, you must have already configured the registry to use the `config/templates.xml` file as your default templates (see Section 38.7.4), or copied the template into the XML file describing your application.

Note that IceGrid cannot start a Glacier2 router if the router's security configuration requires that a passphrase be entered. In this situation, you have no choice but to start the router yourself so that you can provide the passphrase when prompted.

38.16 Using an SQL Database

By default, IceGrid uses a Freeze database (see Chapter 39) to store its persistent state. You can configure IceGrid to use an SQL database by setting a number of properties. (Note that SQL support requires recompilation of IceGrid with appro-

priate settings. See the release notes of your Ice distribution for details for how to do this, as well as the supported SQL databases.)

If you use IceGrid with an SQL database, its persistent state is stored in four tables:

- `<instance-name>_<replica-name>_Applications`
- `<instance-name>_<replica-name>_Adapters`
- `<instance-name>_<replica-name>_Objects`
- `<instance-name>_<replica-name>_InternalObjects`

A number of properties control how IceGrid accesses an SQL database:

- `Ice.Plugin.DB`

  To use IceGrid with an SQL database, you must set this property to the value `IceGridSqlDB:createSqlDB`.
- `IceGrid.SQL.DatabaseType`
- `IceGrid.SQL.DatabaseName`
- `IceGrid.SQL.HostName`
- `IceGrid.SQL.Port`
- `IceGrid.SQL.UserName`
- `IceGrid.SQl.Password`

These properties are described in detail in Appendix D.

38.17 XML Reference

This section provides a reference for the XML elements that define IceGrid descriptors, in alphabetical order.

IceGrid XML files must use UTF-8 encoding.

38.17.1 Adapter

An `adapter` element defines an indirect object adapter. Refer to Section 32.4 for more information on object adapters.

Context

This element may only appear as a child of a `server` or `service` element.

Attributes

This element supports the attributes in Table 38.2.

Table 38.2. Attributes of the `adapter` element.

Attribute	Description	Required
`endpoints`	Specifies one or more endpoints for this object adapter. These endpoints typically do not specify a port. This attribute is translated into a definition of the adapter's `Endpoints` configuration property (see Appendix D).	No
`id`	Specifies an object adapter identifier. The identifier must be unique among all adapters and replica groups in the registry. This attribute is translated into a definition of the adapter's `AdapterId` configuration property (see Appendix D). If not defined, a default value is constructed from the adapter name and server id (and service name for an IceBox service).	Yes
`name`	The name of the object adapter as used in the server that creates it.	Yes
`priority`	Specifies the priority of the object adapter as an integer value. The object adapter priority is used by the `Ordered` replica group load balancing policy to determine the order of the endpoints returned by a locate request. Endpoints are ordered from the smallest priority value to the highest. If not defined, the value is 0. See Section 38.10.2 for more information.	No

Table 38.2. Attributes of the `adapter` element.

Attribute	Description	Required
`register-process`	This attribute is only valid if the enclosing server uses an Ice version prior to 3.3. In Ice 3.3 or later, this functionality is replaced by the administrative facility (see Section 38.22). If the value is `true`, this object adapter registers an object in the IceGrid registry that facilitates graceful shutdown of the server. Only one object adapter in a server should set this attribute to `true`. If not defined, the default value is `false`.	No
`replica-group`	Specifies a replica group identifier. A non-empty value signals that this object adapter is a member of the indicated replica group. This attribute is translated into a definition of the adapter's `RepliaGroupId` configuration property (see Appendix D). See Section 38.9 for more information on replication. If not defined, the default value is an empty string.	No
`server-lifetime`	A value of `true` indicates that the lifetime of this object adapter is the same as the lifetime of its server. This information is used by the IceGrid node to determine the state of the server. Specifically, the server is considered activated (see Section 38.25) when all the object adapters with the server-lifetime attribute set to `true` are registered with the registry (the object adapter registers itself during activation). Furthermore, server deactivation is considered to begin as soon as one object adapter with the server-lifetime attribute set to `true` is unregistered with the registry (the object adapter unregisters itself during deactivation). If not defined, the default value is `true`.	No

An optional nested `description` element provides free-form descriptive text.

Example

```
<adapter name="MyAdapter"
    endpoints="default"
    id="MyAdapterId"
    replica-group="MyReplicaGroup">
    <description>A description of the adapter.</description>
    ...
</adapter>
```

38.17.2 Allocatable

An allocatable element defines an allocatable object in the IceGrid registry. Clients can allocate this object using only its identity, or they can allocate objects of a specific type. Refer to Section 38.11.3 for more information on allocatable objects.

Context

This element may only appear as a child of an `adapter` element (Section 38.17.1).

Attributes

This element supports the attributes in Table 38.3.

Table 38.3. Attributes of the `allocatable` element.

Attribute	Description	Required
`identity`	Specifies the identity by which this allocatable object is known.	Yes
`property`	Specifies the name of a property to generate that contains the stringified identity of this allocatable.	No
`type`	An arbitrary string used to group allocatable objects. By convention, the string represents the most-derived Slice type of the object, but an application is free to use another convention.	No

38.17.3 Application

An `application` element defines an IceGrid application. Refer to Section 38.3 for more information. An application typically contains at least one `node` element, but it may also be used for other purposes such as defining server and service templates, default templates, replica groups and property sets.

Context

This element must be a child of an `icegrid` element. Only one `application` element is permitted per file.

Attributes

This element supports the attributes in Table 38.4.

Table 38.4. Attributes of the `application` element.

Attribute	Description	Required
`import-default-templates`	If `true`, the default templates configured for the IceGrid registry are imported and available for use within this application. See Section 38.7.4 for more information on default templates. If not specified, the default value is `false`.	No
`name`	The name of the application. This name must be unique among all applications in the registry. Within the application, child elements can refer to its name using the reserved variable `${application}`.	Yes

An optional nested `description` element provides free-form descriptive text.

Example

```
<icegrid>
    <application name="MyApplication"
        import-default-templates="true">
        <description>A description of the
```

```
            application.</description>
        ...
    </application>
</icegrid>
```

38.17.4 DbEnv

A `dbenv` element causes an IceGrid node to generate Freeze configuration properties for the server or service in which it is defined, and may cause the node to create a database environment directory if necessary. This element may contain `dbproperty` elements (Section 38.17.5). See Chapter 39 for more information on Freeze.

Context

This element may only appear as a child of a `server` element (Section 38.17.19) or a `service` element (Section 38.17.22).

Attributes

This element supports the attributes in Table 38.5.

Table 38.5. Attributes of the `dbenv` element.

Attribute	Description	Required
`home`	Specifies the directory to use as the database environment. If not defined, a subdirectory within the node's data directory is used.	No
`name`	The name of the database environment.	Yes

The values of the name and home attributes are used to define the configuration property shown below:

`Freeze.DbEnv.`*`name`*`.DbHome=`*`home`*

An optional nested `description` element provides free-form descriptive text.

Example

```
<dbenv name="MyEnv" home="/opt/data/db">
    <description>A description of the
        database environment.</description>
    ...
</dbenv>
```

38.17.5 DbProperty

A `dbproperty` element defines a BerkeleyDB configuration property. See Chapter 39 for more information on Freeze.

Context

This element may only appear as a child of a `dbenv` element (Section 38.17.4).

Attributes

This element supports the attributes in Table 38.6.

Table 38.6. Attributes of the `dbproperty` element.

Attribute	Description	Required
`name`	The name of the configuration property.	Yes
`value`	The value of the configuration property. If not defined, the value is an empty string.	No

Example

```
<dbenv name="MyEnv" home="/opt/data/db">
    <description>A description of the
        database environment.</description>
    <dbproperty name="set_cachesize" value="0 52428800 1"/>
</dbenv>
```

38.17.6 Description

A `description` element specifies a description of its parent element.

Context

This element may only appear as a child of the `application`, `replica-group`, `node`, `server`, `service`, `icebox`, `adapter`, and `dbenv` elements.

Attributes

This element supports no attributes.

Example

```
<node name="localnode">
    <description>Free form descriptive text for
    localnode</description>
</node>
```

38.17.7 Directory

A `directory` element specifies a directory in a distribution. Refer to Section 38.13 for more information on distribution descriptors.

Context

This element may only appear as a child of the `distrib` element (Section 38.17.8).

Attributes

This element supports no attributes.

38.17.8 Distrib

A `distrib` element specifies the files to download from an IcePatch2 server as well as the server's proxy. Refer to Section 38.13 for more information on distribution descriptors.

Context

This element may only appear as a child of an `application` element (Section 38.17.3) or a `server` element (Section 38.17.19).

Attributes

This element supports the attributes in Table 38.7.

Table 38.7. Attributes of the `distrib` element.

Attribute	Description	Required
`icepatch`	Specifies the proxy of the IcePatch2 server. If not defined, the default value is `${application}.IcePatch2/server`.	No

Example

```
<distrib icepatch="DemoIcePatch2/server:tcp -p 12345">
    <directory>dir1</directory>
    <directory>dir2/subdir</directory>
</distrib>
```

38.17.9 IceBox

An `icebox` element defines an IceBox server to be deployed on a node. It typically contains at least one `service` element (Section 38.17.22), and may supply additional information such as command-line options (Section 38.17.26), environment variables (Section 38.17.27), configuration properties (Section 38.17.17) and a server distribution (Section 38.13).

The element may optionally contain a single `adapter` element (Section 38.17.1) that configures the IceBox service manager's object adapter and must have the name `IceBox.ServiceManager`. If no such element is defined, the service manager's object adapter is disabled. Note however that the service manager's functionality remains available as an administrative facet even when its object adapter is disabled. See Chapter 43 for more information on IceBox.

Context

This element may only appear as a child of a `node` element (Section 38.17.13) or a `server-template` element (Section 38.17.21).

Attributes

This element supports the same attributes as the `server` element (see Table 38.16).

The IceGrid node on which this server is deployed generates the following configuration property for the server:

```
IceBox.InstanceName=${server}
```

An optional nested `description` element provides free-form descriptive text.

Example

```
<icebox id="MyIceBox"
    activation="on-demand"
    activation-timeout="60"
    application-distrib="false"
    deactivation-timeout="60"
    exe="/opt/Ice/bin/icebox"
    pwd="/">
    <option>--Ice.Trace.Network=1</option>
    <env>PATH=/opt/Ice/bin:$PATH</env>
    <property name="IceBox.UseSharedCommunicator.Service1"
        value="1"/>
    <service name="Service1".../>
    <service-instance template="ServiceTemplate" name="Service2"/>
</icebox>
```

38.17.10 IceGrid

The `icegrid` element is the top-level element for IceGrid descriptors in XML files. This elements supports no attributes.

38.17.11 Load Balancing

A `load-balancing` element determines the load balancing policy used by a replica group. Refer to Section 38.10 for details on load balancing.

Context

This element may only appear as a child of a `replica-group` element (Section 38.17.18).

Attributes

This element supports the attributes in Table 38.8.

Table 38.8. Attributes of the `load-balancing` element.

Attribute	Description	Required
`load-sample`	Specifies the load sample to use for the adaptive load balancing policy. It can only be defined if `type` is set to `adaptive`. Legal values are `1`, `5` and `15`. If not specified, the load sample default value is `1`.	No
`n-replicas`	Specifies the maximum number of replicas used to compute the endpoints of the replica group. If not specified, the default value is `1`.	No
`type`	Specifies the type of load balancing. Legal values are `adaptive`, `ordered`, `round-robin` and `random`.	Yes

Example

```
<application name="MyApp">
    <replica-group id="ReplicatedAdapter">
        <load-balancing type="adaptive" load-sample="15"
            n-replicas="3"/>
        <description>A description of this
            replica group.</description>
        <object identity="WellKnownObject" .../>
    </replica-group>
    ...
</application>
```

38.17.12 Log

A `log` element specifies the name of a log file for a server or service. A `log` element must be defined for each log file that can be accessed remotely by an administrative tool. Note that it is not necessary to define a log element for the values of the `Ice.StdErr` and `Ice.StdOut` properties.

See Section 38.14.2 for more information on log files.

Context

This element may only appear as a child of a `server` element (Section 38.17.19) or a `service` element (Section 38.17.22).

Attributes

This element supports the attributes in Table 38.9.

Table 38.9. Attributes of the `log` element.

Attribute	Description	Required
`path`	The path name of the log file. If a relative path is specified, it is relative to the current working directory of the node. The node must have sufficient access privileges to read the file.	Yes
`property`	Specifies the name of a property in which to store the path name of the log file as given in the `path` attribute.	No

Example

```
<server id="MyServer" ...>
    <log path="${server}.log" property="LogFile"/>
</server>
```

38.17.13 Node

A `node` element defines an IceGrid node. The servers that the node is responsible for managing are described in child elements.

Context

This element may only appear as a child of an `application` element (Section 38.17.3). Multiple `node` elements having the same name may appear in an application. Their contents are merged and the last definition of `load-factor` has precedence.

Attributes

This element supports the attributes in Table 38.10.

Table 38.10. Attributes of the `node` element.

Attribute	Description	Required
`load-factor`	A floating point value defining the factor that is multiplied with the node's load average. The load average is used by the adaptive load balancing policy to figure out which node is the least loaded (see Section 38.10.2). The default is `1.0` on Unix platforms and `1/NCPUS` on Windows (where `NCPUS` is the number of CPUs in the node's computer). Note that, if Unix and Windows machines are part of a replica group, the Unix and Windows figures are not directly comparable, but the registry still makes an attempt to pick the least-loaded node.	No
`name`	Specifies the name of this node. The name must be unique among all nodes in the registry. Within the node, child elements can refer to its name using the reserved variable `${node}`. An **`icegridnode`** process representing this node must be started on the desired computer and its configuration property `IceGrid.Node.Name` must match this attribute (see Section 38.21.2).	Yes

Example

```
<node name="Node1" load-factor="2.0">
    <description>A description of this node.</description>
    <server id="Server1" ...>
        <property name="NodeName" value="${node}"/>
        ...
    </server>
</node>
```

38.17.14 Object

An `object` element defines a well-known object in the IceGrid registry. Clients can refer to this object using only its identity, or they can search for well-known objects of a specific type. Refer to Section 38.6 for more information on well-known objects.

Context

This element may only appear as a child of an `adapter` element (Section 38.17.1) or a `replica-group` element (Section 38.17.18).

Attributes

This element supports the attributes in Table 38.11.

Table 38.11. Attributes of the `object` element.

Attribute	Description	Required
`identity`	Specifies the identity by which this object is known.	Yes
`property`	Specifies the name of a property to generate that contains the stringified identity of the object. This attribute is only allowed if this `object` element is a child of an `adapter` element.	No
`type`	An arbitrary string used to group objects. By convention, the string represents the most-derived Slice type of the object, but an application is free to use another convention.	No

Example

```
<adapter name="MyAdapter" id="WellKnownAdapter" ...>
    <object identity="WellKnownObject"
        type="::Module::WellKnownInterface"/>
</adapter>
```

In the configuration above, the object can be located via the equivalent proxies `WellKnownObject` and `WellKnownObject@WellKnownAdapter`.

38.17.15 Parameter

A `parameter` element defines a template parameter. Template parameters must be declared with this element to be used in template instantiation.

Context

This element may only appear as a child of a `server-template` element (Section 38.17.21) or a `service-template` element (Section 38.17.24).

Attributes

This element supports the attributes in Table 38.12.

Table 38.12. Attributes of the `parameter` element.

Attribute	Description	Required
`name`	The name of the parameter. For example, if `index` is used as the name of a parameter, it can be referenced using `${index}` in the server or service template.	Yes
`default`	An optional default value for the parameter. This value is used if the parameter is not defined when a server or service is instantiated.	No

Example

```
<server-template id="MyServerTemplate">
    <parameter name="index"/>
    <parameter name="exepath" default="/opt/myapp/bin/server"/>
    ...
</server-template>
```

38.17.16 Properties

The `properties` element is used in three situations:

- as a named property set if the `id` attribute is specified
- as a reference to a named property set if the `refid` attribute is specified
- as an unnamed property set if the `id` or `refid` attributes are not specified.

A property set is useful to define a set of properties (a named property set) in application or node descriptors. Named property sets can be included in named or unnamed property sets with property set references.

Context

A named property set element may only be a child of an `application` element (Section 38.17.3) or a `node` (Section 38.17.13) element. An unnamed property set element may only be a child of a `server`, `icebox`, `service`, `server-instance` or `service-instance` element. An unnamed property set element with the `service` attribute defined may only be a child of a `server-instance` element. A reference to a named property set can only be a child of a named or unnamed property set element.

Attributes

This element supports the attributes in Table 38.13.

Table 38.13. Attributes of the `properties` element.

Attribute	Description	Required
`id`	Defines a new named property set with the given identifier. The identifier must be unique among all named property sets defined in the same scope. If not specified, the properties element refers to an unnamed property set or a property set reference.	No
`refid`	Defines a reference to the named property set with the given identifier. If not specified, the element refers to an unnamed or named property set.	No
`service`	Specifies the name of a service that is defined in the enclosing `server-instance` descriptor. The server instance must be an IceBox server that includes a service with the given name. An unnamed property set with this attribute defined extends the properties of the service. If not specified, the unnamed property set extends the properties of the server instance.	No

Example

```
<application name="Simple">
   <properties id="Debug">
        <property name="Ice.Trace.Network" value="1"/>
   </properties>

    <server id="MyServer" exe="./server">
        <properties>
            <properties refid="Debug"/>
            <property name="AppProperty" value="1"/>
        </properties>
    </server>
</application>
```

38.17.17 Property

An IceGrid node generates a configuration file for each of its servers and services. This file generally should not be edited manually because any changes are lost the next time the node generates the file. The `property` element is the correct way to define additional properties in a configuration file.

Note that IceGrid administrative utilities can retrieve the configuration properties of a server or service, as described in Section 38.22.4.

Context

This element may only appear as a child of a `server` element (Section 38.17.19), a `service` element (Section 38.17.22), an `icebox` element (Section 38.17.9) or a `properties` element (Section 38.17.16).

Attributes

This element supports the attributes in Table 38.14.

Table 38.14. Attributes of the `property` element.

Attribute	Description	Required
`name`	Specifies the property name.	Yes
`value`	Specifies the property value. If not defined, the value is an empty string.	No

Example

```
<server id="MyServer" ...>
    <property name="Ice.ThreadPool.Server.SizeMax" value="10"/>
    ...
</server>
```

This `property` element adds the following definition to the server's configuration file:

```
Ice.ThreadPool.Server.SizeMax=10
```

38.17.18 Replica Group

A `replica-group` element creates a virtual object adapter in order to provide replication and load balancing for a collection of object adapters. An `adapter` element (Section 38.17.1) declares its membership in a group by identifying the desired replica group. The element may declare well-known objects (Section 38.6) that are available in all of the participating object adapters. A `replica-group` element may contain a `load-balancing` child element that specifies the load-balancing algorithm the registry should use when resolving locate requests. If not specified, the registry uses a random load balancing policy with the number of replicas set to 0.

Refer to Section 38.9 for more information on replication and Section 38.10 for details on load balancing.

Context

This element may only appear as a child of an `application` element (Section 38.17.3).

Attributes

This element supports the attributes in Table 38.15.

Table 38.15. Attributes of the `replica-group` element.

Attribute	Description	Required
`id`	Specifies the identifier of the replica group, which must be unique among all adapters and replica groups in the registry. This identifier can be used in indirect proxies in place of an adapter identifier.	Yes

An optional nested `description` element provides free-form descriptive text.

Example

```
<application name="MyApp">
    <replica-group id="ReplicatedAdapter">
        <load-balancing type="adaptive" load-sample="15"
            n-replicas="3"/>
        <description>A description of this
            replica group.</description>
        <object identity="WellKnownObject" .../>
    </replica-group>
    ...
</application>
```

In this example, the proxy `WellKnownObject` is equivalent to the proxy `WellKnownObject@ReplicatedAdapter`.

38.17.19 Server

A `server` element defines a server to be deployed on a node. It typically contains at least one `adapter` element (Section 38.17.1), and may supply additional information such as command-line options (Section 38.17.26), environment variables (Section 38.17.27), configuration properties (Section 38.17.17), and a server distribution (Section 38.13).

Context

This element may only appear as a child of a `node` element (Section 38.17.13) or a `server-template` element (Section 38.17.21).

Attributes

This element supports the attributes in Table 38.16.

Table 38.16. Attributes of the `server` element.

Attribute	Description	Required
`activation`	Specifies whether the server's activation mode. Legal values are `manual`, `on-demand`, `always` and `session`. If not specified, manual activation is used by default. See Section 38.25 for more information.	No
`activation-timeout`	Defines the number of seconds a node will wait for the server to activate. If the timeout expires, a client waiting to receive the endpoints of an object adapter in this server will receive an empty set of endpoints. If not defined, the default timeout is the value of the `IceGrid.Node.WaitTime` property configured for the server's node.	No
`allocatable`	Specifies whether the server can be allocated. A server is allocated implicitly when one of its allocatable objects is allocated. The value of this attribute is ignored if the server activation mode is `session`; a server with this activation mode is always allocatable. Otherwise, if not specified and the server activation mode is not `session`, the server is not allocatable.	No
`application-distrib`	Specifies whether this server's distribution is dependent on the application's distribution. If the value is `true`, the server cannot be patched until the application has been patched. If not defined, the default value is `true`. See Section 38.13.3 for more information on distribution dependencies.	No

Table 38.16. Attributes of the `server` element.

Attribute	Description	Required
`deactivation-timeout`	Defines the number of seconds a node will wait for the server to deactivate. If the timeout expires, the node terminates the server process. If not defined, the default timeout is the value of the node's configuration property `IceGrid.Node.WaitTime`.	No
`exe`	The pathname of the server executable.	Yes
`ice-version`	Specifies the Ice version in use by this server. If not defined, IceGrid assumes the server uses the same version that IceGrid itself uses. A server that uses an Ice version prior to 3.3 must define this attribute if its adapters use the `register-process` attribute (see Section 38.17.1). For example, a server using Ice 3.2.x should use `3.2` as the value of this attribute.	No
`id`	Specifies the identifier for this server. The identifier must be unique among all servers in the registry. Within the server, child elements can refer to its identifier using the reserved variable `${server}`.	Yes
`pwd`	The default working directory for the server. If not defined, the server is started in the node's current working directory.	No

Table 38.16. Attributes of the `server` element.

Attribute	Description	Required
`user`	Specifies the name of the user account under which the server is activated and run. If a user account mapper is configured for the node, the value of this attribute is used to find the corresponding account in the map. On Unix, the node must be running as root to be able to run servers under a different user account. On Windows, or if the node is not running as root on Unix, the only permissible value for this attribute is an empty string or the name of the user account under which the node is running. On Unix, if the node is running as root and this attribute is not specified, the server is run under the user `${session.id}` if the server activation mode is `session` or under the user `nobody` if the activation mode is `manual`, `on-demand` or `always`.	No

An optional nested `description` element provides free-form descriptive text.

Example

```
<server id="MyServer"
    activation="on-demand"
    activation-timeout="60"
    application-distrib="false"
    deactivation-timeout="60"
    exe="/opt/app/bin/myserver"
    pwd="/">
    <option>--Ice.Trace.Network=1</option>
    <env>PATH=/opt/Ice/bin:$PATH</env>
    <property name="ServerId" value="${server}"/>
    <adapter name="Adapter1" .../>
</server>
```

38.17.20 Server Instance

A `server-instance` element deploys an instance of a `server-template` element (Section 38.17.21) on a node. Refer to Section 38.7 for more information on templates.

Context

This element may only appear as a child of a `node` element (Section 38.17.13).

Attributes

This element supports the attributes in Table 38.17.

Table 38.17. Attributes of the `server-instance` element.

Attribute	Description	Required
`template`	Identifies the server template.	Yes

All other attributes of the element must correspond to parameters declared by the template. The `server-instance` element must provide a value for each parameter that does not have a default value supplied by the template.

Example

```
<icegrid>
    <application name="SampleApp">
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <server id="${id}" activation="manual" .../>
        </server-template>
        <node name="Node1">
            <server-instance template="ServerTemplate"
                id="TheServer"/>
        </node>
    </application>
</icegrid>
```

38.17.21 Server Template

A `server-template` element defines a template for a `server` element (Section 38.17.19), simplifying the task of deploying multiple instances of the same server definition. The template should contain a parameterized `server` element that is instantiated using a `server-instance` element (Section 38.17.20). Refer to Section 38.7 for more information on templates.

Context

This element may only appear as a child of an `application` element (Section 38.17.3).

Attributes

This element supports the attributes in Table 38.18.

Table 38.18. Attributes of the `server-template` element.

Attribute	Description	Required
`id`	Specifies the identifier for the server template. This identifier must be unique among all server templates in the application.	Yes

Parameters

A template may declare parameters that are used to instantiate the `server` element. You can define a default value for each parameter. Parameters without a default value are considered mandatory and values for them must be supplied by the `server-instance` element. See Section 38.18 for more information on parameter semantics.

Example

```
<icegrid>
    <application name="SampleApp">
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <server id="${id}" activation="manual" .../>
        </server-template>
        <node name="Node1">
```

```
            <server-instance template="ServerTemplate"
                id="TheServer"/>
        </node>
    </application>
</icegrid>
```

38.17.22 Service

A `service` element defines an IceBox service. It typically contains at least one `adapter` element (Section 38.17.1), and may supply additional information such as configuration properties (Section 38.17.17).

Refer to Section 38.8 for more information on using IceBox services in IceGrid.

Context

This element may only appear as a child of an `icebox` element (Section 38.17.9) or a `service-template` element (Section 38.17.24).

Attributes

This element supports the attributes in Table 38.19.

Table 38.19. Attributes of the `service` element.

Attribute	Description	Required
`entry`	Specifies the entry point of this service.	Yes
`name`	Specifies the name of this service. Within the service, child elements can refer to its name using the reserved variable `${service}`.	Yes

An optional nested `description` element provides free-form descriptive text.

Example

```
<icebox id="MyIceBox" ...>
    <service name="Service1" entry="service1:Create">
        <description>A description of this service.</description>
        <property name="ServiceName" value="${service}"/>
```

```
        <adapter name="MyAdapter" id="${service}Adapter" .../>
    </service>
    <service name="Service2" entry="service2:Create"/>
</icebox>
```

38.17.23 Service Instance

A `service-instance` element creates an instance of a `service-template` element in an IceBox server (see Section 38.8). Refer to Section 38.7 for more information on templates.

Context

This element may only appear as a child of an `icebox` element (Section 38.17.9).

Attributes

This element supports the attributes in Table 38.20.

Table 38.20. Attributes of the `service-instance` element.

Attribute	Description	Required
`template`	Identifies the service template.	Yes

All other attributes of the element must correspond to parameters declared by the template. The `service-instance` element must provide a value for each parameter that does not have a default value supplied by the template.

Example

```
<icebox id="IceBoxServer" ...>
    <service-instance template="ServiceTemplate" name="Service1"/>
</icebox>
```

38.17.24 Service Template

A `service-template` element defines a template for a `service` element (Section 38.17.22), simplifying the task of deploying multiple instances of the

same service definition. The template should contain a parameterized `service` element that is instantiated using a `service-instance` element (Section 38.17.23). Refer to Section 38.8.2 for more information on service templates.

Context

This element may only appear as a child of an `application` element (Section 38.17.3).

Attributes

This element supports the attributes in Table 38.21.

Table 38.21. Attributes of the `service-template` element.

Attribute	Description	Required
`id`	Specifies the identifier for the service template. This identifier must be unique among all service templates in the application.	Yes

Parameters

A template may declare parameters that are used to instantiate the `service` element. You can define a default value for each parameter. Parameters without a default value are considered mandatory and values for them must be supplied by the `service-instance` element. See Section 38.18 for more information on parameter semantics.

Example

```
<icegrid>
    <application name="IceBoxApp">
        <service-template id="ServiceTemplate">
            <parameter name="name"/>
            <service name="${name}" entry="DemoService:create">
                <adapter name="${service}" .../>
            </service>
        </service-template>
        <node name="Node1">
            <icebox id="IceBoxServer" ...>
```

```
                <service-instance template="ServiceTemplate"
                    name="Service1"/>
            </icebox>
        </node>
    </application>
</icegrid>
```

38.17.25 Variable

A `variable` element defines a variable. See Section 38.18 for more information on variable semantics.

Context

This element may only appear as a child of an `application` element (Section 38.17.3) or `node` element (Section 38.17.13).

Attributes

This element supports the attributes in Table 38.22.

Table 38.22. Attributes of the `variable` element.

Attribute	Description	Required
`name`	Specifies the variable name. The value of this variable is substituted whenever its name is used in variable syntax, as in `${name}`.	Yes
`value`	Specifies the variable value. If not defined, the default value is an empty string.	No

Example

```
<icegrid>
    <application name="SampleApp">
        <variable name="Var1" value="foo"/>
        <variable name="Var2" value="${Var1}bar"/>
        ...
    </application>
</icegrid>
```

38.17.26 Command Line Options

Server descriptors (Section 38.17.19) and icebox descriptors (Section 38.17.9) may specify command-line options that the node will pass to the program at startup. As the node prepares to execute the server, it assembles the command by appending options to the server executable's pathname.

In XML, you define a command-line option using the `option` element:

```
<server id="Server1" ...>
    <option>--Ice.Trace.Protocol</option>
    ...
</server>
```

The node preserves the order of options, which is especially important for Java servers. For example, JVM options must appear before the class name, as shown below:

```
<server id="JavaServer" exe="java" ...>
    <option>-Xnoclassgc</option>
    <option>ServerClassName</option>
    <option>--Ice.Trace.Protocol</option>
    ...
</server>
```

The node translates these options into the following command:

```
java -Xnoclassgc ServerClassName --Ice.Trace.Protocol
```

38.17.27 Environment Variables

Server descriptors (Section 38.17.19) and icebox descriptors (Section 38.17.9) may specify environment variables that the node will define when starting a server. An environment variable definition uses the familiar *`name=value`* syntax, and you can also refer to other environment variables within the value. The exact syntax for variable references depends on the platform on which the server's descriptor is deployed.

On a Unix platform, the Bourne shell syntax is required:

```
LD_LIBRARY_PATH=/opt/Ice/lib:$LD_LIBRARY_PATH
```

On a Windows platform, the syntax uses the conventional style:

```
PATH=C:\Ice\lib;%PATH%
```

In XML, the `env` element supplies a definition for an environment variable:

```
<node name="UnixBox">
    <server id="UnixServer" exe="/opt/app/bin/server" ...>
        <env>LD_LIBRARY_PATH=/opt/Ice/lib:$LD_LIBRARY_PATH</env>
        ...
    </server>
</node>
<node name="WindowsBox">
    <server id="WindowsServer" exe="C:/app/bin/server.exe" ...>
        <env>PATH=C:\Ice\lib;%PATH%</env>
        ...
    </server>
</node>
```

If a value refers to an environment variable that is not defined, the reference is substituted with an empty string.

Environment variable definitions may also refer to descriptor variables and template parameters:

```
<node name="UnixBox">
    <server id="UnixServer" exe="/opt/app/bin/server" ...>
        <env>PATH=${server.distrib}/bin:$PATH</env>
        ...
    </server>
</node>
```

On Unix, an environment variable `VAR` can be referenced as `$VAR` or `${VAR}`. You must be careful when using the latter syntax because IceGrid assumes `${VAR}` refers to a descriptor variable or parameter and will report an error if no match is found. If you prefer to use this style to refer to environment variables, you must escape these occurrences as shown in the example below:

```
<node name="UnixBox">
    <server id="UnixServer" exe="/opt/app/bin/server" ...>
        <env>PATH=${server.distrib}/bin:$${PATH}</env>
        ...
    </server>
</node>
```

IceGrid does not attempt to perform substitution on `$${PATH}`, but rather removes the leading `$` character and then performs environment variable substitution on `${PATH}`. See Section 38.18.1 for more information on escaping variables.

38.18 Variable and Parameter Semantics

Variable descriptors (see Section 38.17.25) allow you to define commonly-used information once and refer to them symbolically throughout your application descriptors.

38.18.1 Syntax

Substitution for a variable or parameter *VP* is attempted whenever the symbol ${*VP*} is encountered, subject to the limitations and rules described below. Substitution is case-sensitive, and a fatal error occurs if *VP* is not defined.

Limitations

Substitution is only performed in string values, and excludes the following cases:

- in the identifier of a template descriptor definition:

  ```
  <server-template id="${invalid}" ...>
  ```

- in the name of a variable definition

  ```
  <variable name="${invalid}" ...>
  ```

- in the name of a template parameter definition

  ```
  <parameter name="${invalid}" ...>
  ```

- in the name of a template parameter assignment

  ```
  <server-instance template="T" ${invalid}="val" ...>
  ```

- in the name of a node definition

  ```
  <node name="${invalid}" ...>
  ```

- in the name of an application definition

  ```
  <application name="${invalid}" ...>
  ```

Substitution is not supported for values of other types. The example below demonstrates an invalid use of substitution:

```
<variable name="register" value="true"/>
<node name="Node">
    <server id="Server1" ...>
        <adapter name="Adapter1" register-process=${register} .../>
```

In this case, a variable cannot supply the value of `register-process` because that attribute expects a boolean value, not a string.

Most values are strings, however, so this limitation is rarely a problem.

Escaping a Variable

You can prevent substitution by escaping a variable reference with an additional leading `$` character. For example, in order to assign the literal string `${abc}` to a variable, you must escape it as shown below:

```
<variable name="x" value="$${abc}"/>
```

The extra `$` symbol is only meaningful when immediately preceding a variable reference, therefore text such as `US$$55` is not modified. Each occurrence of the characters `$$` preceding a variable reference is replaced with a single `$` character, and that character does not initiate a variable reference. Consider these examples:

```
<variable name="a" value="hi"/>
<variable name="b" value="$${a}"/>
<variable name="c" value="$$${a}"/>
<variable name="d" value="$$$${a}"/>
```

After substitution, `b` has the value `$${a}`, `c` has the value `$hi`, and `d` has the value `$${a}`.

38.18.2 Reserved Names

IceGrid defines a set of read-only variables to hold information that may be of use to descriptors. The names of these variables are reserved and cannot be used as variable or parameter names. Table 38.23 describes the purpose of each variable and defines the context in which it is valid.

Table 38.23. Reserved names.

Name	Description
`application`	The name of the enclosing application.
`application.distrib`	The pathname of the enclosing application's distribution directory, and an alias for `${node.data-dir}/distrib/${application}`.

Table 38.23. Reserved names.

Name	Description
`node`	The name of the enclosing node.
`node.os`	The name of the enclosing node's operating system. On Unix, this is value is provided by `uname`. On Windows, the value is `Windows`.
`node.hostname`	The host name of the enclosing node.
`node.release`	The operating system release of the enclosing node. On Unix, this value is provided by `uname`. On Windows, the value is obtained from the `OSVERSIONINFO` data structure.
`node.version`	The operating system version of the enclosing node. On Unix, this value is provided by `uname`. On Windows, the value represents the current service pack level.
`node.machine`	The machine hardware name of the enclosing node. On Unix, this value is provided by `uname`. On Windows, the value can be x86, x64, or IA64, depending on the machine architecture.
`node.datadir`	The absolute pathname of the enclosing node's data directory.
`server`	The id of the enclosing server.
`server.distrib`	The pathname of the enclosing server's distribution directory, and an alias for `${node.datadir}/servers/${server}/distrib`.
`service`	The name of the enclosing service.
`session.id`	The client session identifier. For sessions created with a user name and password, the value is the user ID; for sessions created from a secure connection, the value is the distinguished name associated with the connection.

The availability of a variable is easily determined in some cases, but may not be readily apparent in others. For example, the following example represents a valid use of the ${node} variable:

```
<icegrid>
    <application name="App">
        <server-template id="T" ...>
            <parameter name="id"/>
            <server id="${id}" ...>
                <property name="NodeName" value="${node}"/>
                ...
            </server>
        </server-template>
        <node name="TheNode">
            <server-instance template="T" id="TheServer"/>
        </node>
    </application>
</icegrid>
```

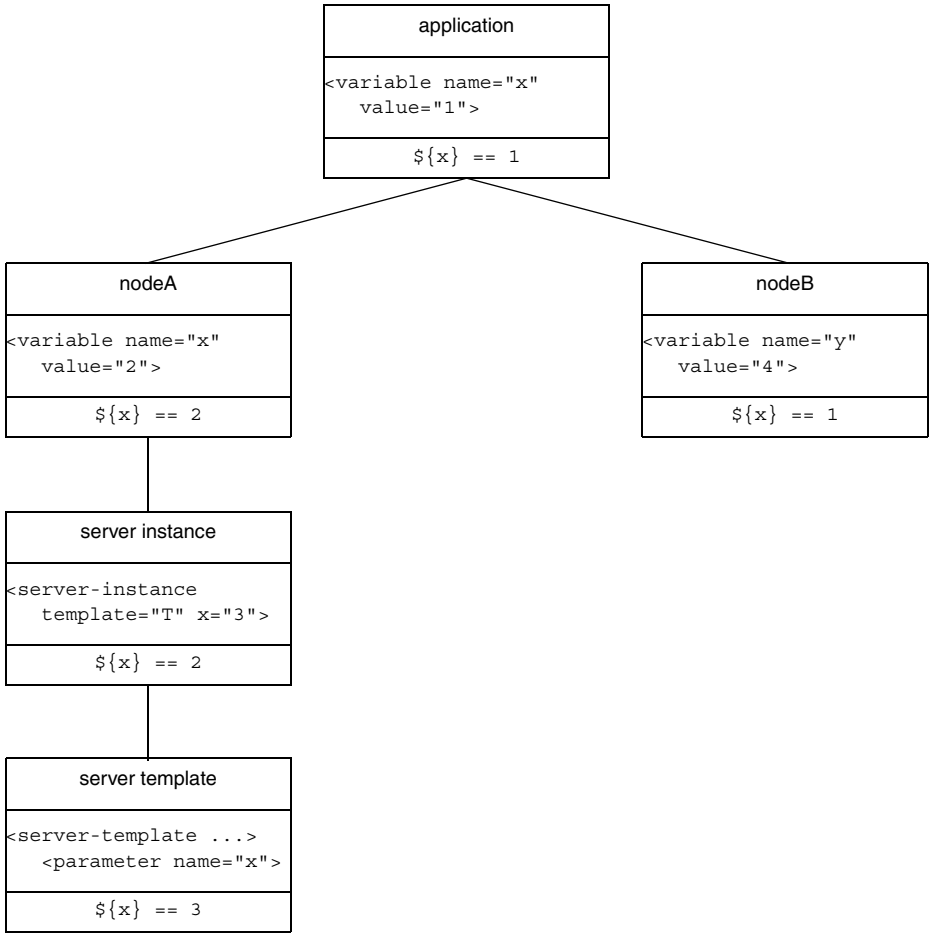
Although the server template descriptor is defined as a child of an application descriptor, its variables are not evaluated until it is instantiated. Since a template *instance* is always enclosed within a node, it is able to use the ${node} variable.

38.18.3 Scoping Rules

Descriptors may only define variables at the application and node levels. Each node introduces a new scope, such that defining a variable at the node level overrides (but does not modify) the value of an application variable with the same name. Similarly, a template parameter overrides the value of a variable with the same name in an enclosing scope. A descriptor may refer to a variable defined in

any enclosing scope, but its value is determined by the nearest scope. Figure 38.9 illustrates these concepts.

Figure 38.9. Variable scoping semantics.



In this diagram, the variable `x` is defined at the application level with the value `1`. In `nodeA`, `x` is overridden with the value `2`, whereas `x` remains unchanged in `nodeB`. Within the context of `nodeA`, `x` continues to have the value `2` in a server instance definition. However, when `x` is used as the name of a template parameter,

the node's definition of `x` is overridden and `x` has the value `3` in the template's scope.

Resolving a Reference

To resolve a variable reference `${var}`, IceGrid searches for a definition of `var` using the following order of precedence:

1. Pre-defined variables (see Section 38.18.2)
2. Template parameters, if applicable
3. Node variables, if applicable
4. Application variables

After the initial substitution, any remaining references are resolved recursively using the following order of precedence:

1. Pre-defined variables (see Section 38.18.2)
2. Node variables, if applicable
3. Application variables

Template Parameters

Template parameters are not visible in nested template instances. This situation can only occur when an IceBox server template instantiates a service template, as shown in the following example:

```
<icegrid>
    <application name="IceBoxApp">
        <service-template id="ServiceTemplate">
            <parameter name="name"/>
            <service name="${name}" entry="DemoService:create">
                ...
                <property name="${name}.Identity"
                    value="${id}-${name}"/> <!-- WRONG! -->
            </service>
        </service-template>
        <server-template id="ServerTemplate">
            <parameter name="id"/>
            <icebox id="${id}" endpoints="default" ...>
                <service-instance template="ServiceTemplate"
                    name="Service1"/>
            </icebox>
        </server-template>
        <node name="Node1">
            <server-instance template="ServerTemplate"
```

```
                id="IceBoxServer"/>
        </node>
    </application>
</icegrid>
```

The service template incorrectly refers to `id`, which is a parameter of the server template.

Template parameters can be referenced only in the body of a template; they cannot be used define other parameters. For example, the following is illegal:

```
<server-template id="ServerTemplate">
    <parameter name="par1"/>
    <parameter name="par2" default="${par1}"/>
    ...
</server-template>
```

Modifying a Variable

A variable definition can be overridden in an inner scope, but the inner definition does not modify the outer variable. If a variable is defined multiple times in the same scope (which is only relevant in XML definitions), the most recent definition is used for all references to that variable. Consider the following example:

```
<application name="MyApp">
    <variable name="x" value="1"/>
    <variable name="y" value="${x}"/>
    <variable name="x" value="2"/>
    ...
</application>
```

When descriptors such as these are created, IceGrid validates their variable references but does not perform substitution until the descriptor is acted upon (such as when a node is generating a configuration file for a server). As a result, the value of `y` in the above example is `2` because that is the most recent definition of `x`.

38.19 Property Set Semantics

Ice servers and clients are configured with properties. For servers deployed with IceGrid, these properties are automatically generated into a configuration file from the information contained in the application descriptor. The settings in that configuration file are passed to server via the **`--Ice.Config`** command-line option.

Descriptors allow you to define property sets to efficiently manage and specify properties. Here are some of the benefits of using property sets:

- You can define sets of properties at the application or node element level and reference these properties in other property sets.
- You can specify properties for a specific server or service instance.

There are two kinds of property sets:

- Named property sets

  Named property sets are defined at the application or node level. They are useful only as the target of references from other property sets. Specifically, a named property set has no effect unless you reference it from a `server` descriptor.

- Unnamed property sets

  Unnamed property sets are defined at the `server`, `service`, `icebox`, `server-instance` or `service-instance` element level and define the properties for a server or service. Unnamed property sets can reference named property sets.

Named and unnamed property sets are defined with the same `properties` element. The context and the attributes of a `properties` element distinguish named property sets from unnamed property sets. Here is an example that defines a named and an unnamed property set:

```
<application name="App">
    <properties id="Debug">
        <property name="UseDebug" value="1"/>
    </properties>

    <node name="TheNode">
        <server id="TheServer" exe="./server">
            <properties>
                <property name="Identity" value="hello"/>
            </properties>
        </server>
    </node>
</application>
```

In this example, we define the named property set `Debug` and the unnamed property set of the server `TheServer`. The server configuration will contain only the `Identity` property because the server property set does not reference the `Debug` named property set.

The `properties` element is used to reference a named property set: if a `properties` element appears inside another `properties` element, it is a reference to another property set and it must specify the `refid` attribute. With the previous example, to reference the `Debug` property set, we would write the following:

```
<application name="App">
    <properties id="Debug">
        <property name="UseDebug" value="1"/>
    </properties>

    <node name="TheNode">
        <server id="TheServer" exe="./server">
            <properties>
                <properties refid="Debug"/>
                <property name="Identity" value="hello"/>
        </properties>
        </server>
    </node>
</application>
```

Property sets, whether named or unnamed, are evaluated as follows:

1. Within a `properties` element, IceGrid locates all references to named property sets and evaluates all property settings in the referenced property sets.

2. Explicit property definitions following any named references are then evaluated and added to the property set formed in the preceding step. This means that explicit property settings override corresponding settings in any referenced property sets.

It is illegal to define a reference to a property set after setting a property value, so references to property sets must precede property definitions. For example, the following is illegal:

```
<properties>
    <property name="Prop1" value="Value1"/>
    <properties refid="Ref1"/>
</properties>
```

Just as the order of the property definitions is important, the order of property set references is also important. For example, the following two property sets are not equivalent:

```
<properties>
    <properties refid="Ref1"/>
    <properties refid="Ref2"/>
</properties>

<properties>
    <properties refid="Ref2"/>
    <properties refid="Ref1"/>
</properties>
```

Named property sets are evaluated at the point of definition. If you reference other property sets or use variables in a named property set definition, you must make sure that the referenced property sets or variables are defined in the same scope. For example, the following is correct:

```
<application name="App">

    <variable name="level" value="1"/>

    <properties id="DebugApp">
        <property name="DebugLevel value="${level}">
    </properties>

</application>
```

However, the following example is wrong because the `${level}` variable is not defined at the `application` scope:

```
<application name="App">

    <properties id="DebugApp">
        <property name="DebugLevel value="${level}">
    </properties>

    <node name="TheNode">
        <variable name="level" value="1"/>
    </node>

</application>
```

If both the `application` and the `node` define the `${level}` variable, the value of the `${level}` variable in the `DebugApp` property set will be the value of the variable defined in the application descriptor.

So far, we have seen the definition of an unnamed property set only in a server descriptor. However, it is also possible to define an unnamed property set for

server or service instances. This is a good way to specify or override properties specific to a server or service instance. For example:

```
<application name="TheApp">
  <server-template id="Template">

    <parameter name="instance-name"/>

    <server id="${instance-name}" exe="./server">
      <properties>
        <property name="Timeout" value="30"/>
      </properties>
    </server>
  </server-template>

  <node name="TheNode">
    <server-instance template="Template" instance-name="MyInst">
      <properties>
        <property name="Debug" value="1"/>
        <property name="Timeout" value="-1"/>
      </properties>
    </server-instance>
  </node>
</application>
```

Here, the server instance overrides the `Timeout` property and defines an additional `Debug` property.

The server or service instance properties are evaluated as follows:

1. The unnamed property set from the template server or service descriptor is evaluated.
2. The unnamed property set from the server or service instance descriptor is evaluated and the resulting properties are added to the property set formed in the preceding step. This means that property settings in a server or service instance descriptor override corresponding settings in a template server or service descriptor.

The server or service instance unnamed property set and its parameters provide two different ways to customize the properties of a server or service template instance. It might not always be obvious which method to use: is it better to use a parameter to parameterize a given property or is it better to just specify it in the server or service instance property set?

For example, in the previous descriptor, we could have used a parameter with a default value for the `Timeout` property:

```
<application name="TheApp">
  <server-template id="Template">

    <parameter name="instance-name"/>
    <parameter name="timeout" default="30"/>

    <server id="${instance-name}" exe="./server">
      <properties>
        <property name="Timeout" value="${timeout}"/>
      </properties>
    </server>
  </server-template>

  <node name="TheNode">
    <server-instance template="Template"
                     instance-name="MyInst" timeout="-1">
      <properties>
        <property name="Debug" value="1"/>
      </properties>
    </server-instance>
  </node>
</application>
```

Here are some guidelines to help you decide whether to use a parameter or a property:

- Use a parameter for a property that should always be set.
- Use a parameter if you want to make the property obvious to the reader and user of the template.
- Do not use a parameter for optional properties if you want to rely on a default value for the server.
- Do not use parameters for properties that are rarely used.

38.20 XML Features

IceGrid provides some convenient features to simplify the task of defining descriptors in XML.

38.20.1 Targets

An IceGrid XML file may contain optional definitions that are deployed only when specifically requested. These definitions are called targets and must be defined within a `target` element. The elements that may legally appear within a `target` element are determined by its enclosing element. For example, a `node` element is legal inside a `target` element of an `application` element, but not inside a `target` element of a `server` element. Each `target` element must define a value for the `name` attribute, but names are not required to be unique. Rather, targets should be considered as optional components or features of an application that are deployed in certain circumstances.

The example below defines targets named `debug` that, if requested during deployment, configure their servers with an additional property:

```
<icegrid>
    <application name="MyApp">
        <node name="Node">
            <server id="Server1" ...>
                <target name="debug">
                    <property name="Ice.Trace.Network" value="2"/>
                </target>
                ...
            </server>
            <server id="Server2" ...>
                <target name="debug">
                    <property name="Ice.Trace.Network" value="2"/>
                </target>
                ...
            </server>
        </node>
    </application>
</icegrid>
```

Target names specified in an **`icegridadmin`** command (see Section 38.24.1) can be unqualified names like `debug`, in which case every target with that name is deployed, regardless of the target's nesting level. If you want to deploy targets more selectively, you can specify a fully-qualified name instead. A fully-qualified target name consists of its unqualified name prefaced by the names or identifiers of each enclosing element. For instance, a fully-qualified target name from the example above is `MyApp.Node.Server1.debug`.

38.20.2 Including Files

You can include the contents of another XML file into the current file using the `include` element, which is replaced with the contents of the included file. The elements in the included file must be enclosed in an `icegrid` element, as shown in the following example:

```
<!-- File: A.xml -->
<icegrid>
    <server-template id="ServerTemplate">
        <parameter name="id"/>
        ...
    </server-template>
</icegrid>

<!-- File: B.xml -->
<icegrid>
    <application name="MyApp">
        <include file="A.xml"/>
        <node name="Node">
            <server-instance template="ServerTemplate" .../>
        </node>
    </application>
</icegrid>
```

In `B.xml`, the `include` element identifies the name of the file to include using the `file` attribute. The top-level `icegrid` element is discarded from `A.xml` and its contents are inserted at the position of the `include` element in `B.xml`.

Note that the file name of an included file is relative to the application descriptor, not relative to the working directory.

You can include specific targets (see Section 38.20.1) from a file by specifying their names in the optional `targets` attribute. If multiple targets are included, their names must be separated by whitespace. The example below illustrates the use of a target:

```
<!-- File: A.xml -->
<icegrid>
    <server-template id="ServerTemplate">
        <parameter name="id"/>
        ...
    </server-template>
    <target name="targetA">
        <server-template id="AnotherTemplate">
            ...
        </server-template>
```

```
    </target>
</icegrid>

<!-- File: B.xml -->
<icegrid>
    <application name="MyApp">
        <include file="A.xml" targets="targetA"/>
        <node name="Node">
            <server-instance template="ServerTemplate" .../>
            <server-instance template="AnotherTemplate" .../>
        </node>
    </application>
</icegrid>
```

38.21 Server Reference

38.21.1 icegridregistry

The IceGrid registry is a centralized repository of information, including deployed applications and well-known objects. A registry can optionally be collocated with an IceGrid node, which conserves resources and can be convenient during development and testing. The registry server is implemented by the **`icegridregistry`** executable.

Usage

The registry supports the following command-line options:

```
$ icegridregistry -h
Usage: icegridregistry [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
--nowarn             Don't print any security warnings.
--readonly           Start the master registry in read-only mode.
```

The **`--readonly`** option prevents any updates to the registry's database; it also prevents slaves from synchronizing their databases with this master. This option is useful when you need to verify that the master registry's database is correct after promoting a slave to become the new master (see Section 38.21.5).

Additional command line options are supported, including those that allow the registry to run as a Windows service or Unix daemon. See Appendix H for more information.

Configuring Endpoints

The IceGrid registry creates up to five sets of endpoints, configured with the following properties:

- `IceGrid.Registry.Client.Endpoints`

  Client-side endpoints supporting the following interfaces:

  - `Ice::Locator`
  - `IceGrid::Query`
  - `IceGrid::Registry`
  - `IceGrid::Session`
  - `IceGrid::AdminSession`
  - `IceGrid::Admin`

  There are security implications in allowing access to administrative sessions, as explained in the next section.

- `IceGrid.Registry.Server.Endpoints`

  Server-side endpoints for object adapter registration.

- `IceGrid.Registry.SessionManager.Endpoints`

  Optional endpoints for supporting integration with a Glacier2 router. See Section 38.15 for more information.

- `IceGrid.Registry.AdminSessionManager.Endpoints`

  Optional endpoints for supporting integration with a Glacier2 router. See Section 38.15 for more information.

- `IceGrid.Registry.Internal.Endpoints`

  Internal endpoints used by IceGrid nodes and registry replicas. This property must be defined even if no nodes or replicas are being used.

See Appendix D for more information on these properties.

Security Considerations

A client that successfully establishes an administrative session with the registry has the ability to compromise the security of the registry host. As a result, it is imperative that you configure the registry carefully if you intend to allow the use of administrative sessions.

Administrative sessions are disabled unless you explicitly configure the registry to use an authentication mechanism. To allow authentication with a user name and password, you can specify a password file using the property `IceGrid.Registry.AdminCryptPasswords` or use your own permissions verifier object by supplying its proxy in the property `IceGrid.Registry.AdminPermissionsVerifier`. Your verifier object must implement the `Glacier2::PermissionsVerifier` interface.

To authenticate administrative clients using their SSL connections, define `IceGrid.Registry.AdminSSLPermissionsVerifier` with the proxy of a verifier object that implements the `Glacier2::SSLPermissionsVerifier` interface.

Section 42.6.1 provides more information on implementing permissions verifier objects.

Configuring a Data Directory

You must provide an empty directory in which the registry can initialize its databases. The pathname of this directory is supplied by the configuration property `IceGrid.Registry.Data`.

The files in this directory must not be edited manually, but rather indirectly using one of the administrative tools described in Section 38.24. To clear a registry's databases, first ensure the server is not currently running, then remove all of the files in its data directory and restart the server.

Minimal Configuration

The registry requires values for the three mandatory endpoint properties, as well as the data directory property, as shown in the following example:

```
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.Data=/opt/ripper/registry
```

In addition, we also recommend defining `IceGrid.InstanceName`, which is discussed in Section 38.21.3.

The remaining configuration properties are discussed in Appendix D.

38.21.2 icegridnode

An IceGrid node is a process that activates, monitors, and deactivates registered server processes. You can run any number of nodes in a domain, but typically

there is one node per host. A node must be running on each host on which servers are activated automatically, and nodes cannot run without an IceGrid registry.

The IceGrid node server is implemented by the **icegridnode** executable. If you wish to run a registry and node in one process, **icegridnode** is the executable you must use.

Usage

The node supports the following command-line options:

```
Usage: icegridnode [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
--nowarn             Don't print any security warnings.
--readonly           Start the collocated master registry in
                     read-only mode.

--deploy DESCRIPTOR [TARGET1 [TARGET2 ...]]
                     Add or update descriptor in file DESCRIPTOR,
                     with optional targets.
```

If you are running the node with a collocated registry, the **--readonly** option prevents any updates to the registry's database; it also prevents slaves from synchronizing their databases with this master. This option is useful when you need to verify that the master registry's database is correct after promoting a slave to become the new master (see Section 38.21.5).

The **--deploy** option allows an application to be deployed automatically as the node process starts, which can be especially useful during testing. The command expects the name of the XML deployment file, and optionally allows the names of the individual targets within the file to be specified.

Additional command line options are supported, including those that allow the node to run as a Windows service or Unix daemon. See Appendix H for more information.

Security Considerations

It is important that you give careful consideration to the permissions of the account under which the node runs. If the servers that the node will activate have no special access requirements, and all of the servers can use the same account, it is recommended that you do not run the node under an account with system privileges, such as the root account on Unix or the Administrator account on Windows. See Section 38.25.4 for more information on this subject.

Configuring Endpoints

The IceGrid node's endpoints are defined by the `IceGrid.Node.Endpoints` property and must be accessible to the registry. It is not necessary to use a fixed port because each node contacts the registry at startup to provide its current endpoint information.

Configuring a Data Directory

The node requires an empty directory that it can use to store server files. The pathname of this directory is supplied by the configuration property `IceGrid.Node.Data`. To clear a node's state, first ensure the server is not currently running, then remove all of the files in its data directory and restart the server.

When running a collocated node and registry server, we recommend using separate directories for the registry and node data directories.

Minimal Configuration

A minimal node configuration is shown in the following example:

```
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=Node1
IceGrid.Node.Data=/opt/ripper/node

Ice.Default.Locator=IceGrid/Locator:tcp -p 4061
```

The value of the `IceGrid.Node.Name` property must match that of a deployed node known by the registry.

The `Ice.Default.Locator` property is used by the node to contact the registry. The value is a proxy that contains the registry's client endpoints (see Section 38.4.3).

If you wish to run a collocated registry and node server, add the property `IceGrid.Node.CollocateRegistry=1` and include the registry's configuration properties as described in Section 38.21.1.

The remaining configuration properties are discussed in Appendix D.

38.21.3 Object Identities

The IceGrid registry hosts several well-known objects. Table 38.24 shows the default identities of these objects and their corresponding Slice interfaces.

Table 38.24. IceGrid's well-known objects.

Default Identity	Interface
`IceGrid/AdminSessionManager`	`Glacier2::SessionManager`
`IceGrid/AdminSessionManager-`*`replica`*	`Glacier2::SessionManager`
`IceGrid/AdminSSLSessionManager`	`Glacier2::SSLSessionManager`
`IceGrid/AdminSSLSessionManager-`*`replica`*	`Glacier2::SSLSessionManager`
`IceGrid/Locator`	`Ice::Locator`
`IceGrid/Query`	`IceGrid::Query`
`IceGrid/Registry`	`IceGrid::Registry`
`IceGrid/Registry-`*`replica`*	`IceGrid::Registry`
`IceGrid/RegistryUserAccountMapper`	`IceGrid::UserAccountMapper`
`IceGrid/RegistryUserAccountMapper-`*`replica`*	`IceGrid::UserAccountMapper`
`IceGrid/SessionManager`	`Glacier2::SessionManager`
`IceGrid/SSLSessionManager`	`Glacier2::SSLSessionManager`

It is a good idea to assign unique identities to these objects by configuring them with different values for the `IceGrid.InstanceName` property, as shown in the following example:

```
IceGrid.InstanceName=MP3Grid
```

This property changes the identities of the well-known objects to use `MP3Grid` instead of `IceGrid` as the identity category. For example, the identity of the locator becomes `MP3Grid/Locator`.

The client's configuration must also be changed to reflect the new identity:

```
Ice.Default.Locator=MP3Grid/Locator:tcp -h registryhost -p 4061
```

Furthermore, any uses of these identities in application code must be updated as well.

38.21.4 Persistent Data

The IceGrid registry and node both store information in the data directories specified by the `IceGrid.Registry.Data` and `IceGrid.Node.Data` properties, respectively. This section describes what the registry and node are storing and discusses backup and recovery techniques.

Registry

The contents of the registry's data directory depends on the registry's database configuration. By default, the data directory contains a Freeze database environment. It might also contain an SQLite database, or nothing at all if the registry is using a remote SQL server. Regardless of the storage mechanism, the registry stores the following information:

- Applications deployed using the `addApplication` operation on the `IceGrid::Admin` interface (which includes the IceGrid GUI and command-line administrative clients). Applications specify servers, well-known objects, object adapters, replica groups, and allocatable objects. Applications can be removed with the `removeApplication` operation.
- Well-known objects registered using the `addObject` and `addObjectWithType` operations on the `IceGrid::Admin` interface. Well-known objects added by these operations can be removed using the `removeObject` operation.
- Adapter endpoints registered dynamically by servers using the `Ice::LocatorRegistry` interface. The property `IceGrid.Registry.DynamicRegistration` must be set to a value larger than zero to allow the dynamic registration of object adapters. These adapters can be removed using the `removeAdapter` operation.
- Some internal proxies used by the registry to contact nodes and other registry replicas during startup. The proxies enable the registry to notify these entities about the registry's availability.

Client and administrative sessions established with the IceGrid registry are not persistent. If the registry is restarted, these sessions must be recreated. For client sessions in particular, this implies that objects allocated using the allocation mechanism will no longer be allocated once the IceGrid registry restarts.

If the registry's database is corrupted or lost, you must recover the deployed applications, the well-known objects, and the adapter endpoints. You do not need to worry about the internal proxies stored by the registry, as the nodes and registry replicas will eventually contact the registry again.

Depending on your deployed applications and your use of the registry, you should consider backing up the registry's database, especially if you cannot easily recover the persistent information.

For example, if you rely on dynamically-registered adapters, or on well-known objects registered programmatically via the `IceGrid::Admin` interface, you should back up the registry database because recovering this information may be difficult. On the other hand, if you only deploy a few applications from XML files, you can easily recover the applications by redeploying their XML files, and therefore backing up the database may be unnecessary.

Be aware that restarting the registry with an empty database may cause the server information stored by the nodes to be deleted. This can be an issue if the deployed servers have databases stored in the node data directory. The next section provides more information on this subject.

Node

The IceGrid node stores information for servers deployed by IceGrid applications. This information is stored in the `servers` subdirectory of the node's data directory. There is one subdirectory per server; the name of the subdirectory is the server's ID. Each server directory contains configuration files, database environments (see Section 38.17.4) and the distribution data of the server (see Section 38.13). The node's data directory also contains a `distrib` directory to store per-application distribution data. This directory contains a subdirectory for each application that specifies a distribution and has a server deployed on the node.

If a server directory is deleted, the node recreates it at startup. The node will also recreate the server configuration files and the database environment directories. However, the node cannot restore the prior contents of a server's database environment. It is your responsibility to back up these database environments and restore them when necessary. If the server or application distribution data is

deleted from the node's data directory, you can easily recover the deleted information by patching these distributions again using the IceGrid administrative tools.

If you store your server database environments outside the node's data directory (such as in a directory that is regularly backed up), or if you do not have any database environments inside the node's data directory, you do not need to back up the contents of the node's data directory.

38.21.5 Slave Promotion

You may need to promote a slave to be the new master if the current master becomes unavailable. For example, this situation can occur when the original master cannot be restarted immediately due to a hardware problem, or when your application requires a feature that is only accessible via the master, such as the resource allocation mechanism or the ability to modify the deployment data.

To promote a slave to become the new master, you must shut down the slave and change its `IceGrid.Registry.ReplicaName` property to `Master` (or remove the property altogether). On restart, the new master notifies the nodes and registries that were active before it was shut down. An inactive registry or node will eventually connect to the new master if its default locator proxy contains the endpoint of the new master registry or the endpoint of a slave that is connected to the new master. If you cannot afford any down-time of the registry and want to minimize the down-time of the master, you should run at least two slaves. That way, if the master becomes unavailable, there will always be one registry available while you promote one of the slaves.

A slave synchronizes its database upon connecting to the new master, therefore it is imperative that you promote a slave whose database is valid and up-to-date. To verify that the promoted master database is up-to-date, you can start the new master with the **`--readonly`** command-line option. While this option is in force, the new master does not update its database, and slaves do not synchronize their databases. You can review the master database with the IceGrid administrative tools and, if the deployment looks correct, you can restart the master without the **`--readonly`** option to permit updates and slave synchronization.

Note that there is nothing to prevent you from running two masters. If you start two masters and they contain different versions of the deployment information, some slaves and nodes might get updated with out-of-date deployment information (causing some of your servers to be deactivated). You can correct the problem by shutting down the faulty master, but it is important to keep this issue in mind when you restart a master since it might disrupt your applications.

38.21.6 Using a Different Database

By default, the IceGrid registry uses a Freeze database (see Chapter 39) to store its persistent state. You can configure the registry to use an SQL database by setting a number of properties.

If you use the registry with an SQL database, its persistent state is stored in four tables:

- `<instance-name>_<replica-name>_Applications`
- `<instance-name>_<replica-name>_Adapters`
- `<instance-name>_<replica-name>_Objects`
- `<instance-name>_<replica-name>_InternalObjects`

In these tables, *`instance-name`* is the value of `IceGrid.InstanceName` and *`replica-name`* is the value of `IceGrid.Registry.ReplicaName`.

A number of properties control how IceStorm accesses an SQL database:

- `Ice.Plugin.DB`

  To use the IceGrid registry with an SQL database, you must set this property to the value `IceGridSqlDB:createSqlDB`.
- `IceGrid.SQL.DatabaseType`
- `IceGrid.SQL.DatabaseName`
- `IceGrid.SQL.HostName`
- `IceGrid.SQL.Port`
- `IceGrid.SQL.UserName`
- `IceGrid.SQl.Password`

  These properties are described in detail in Appendix D.

38.22 Administrative Facility Integration

The Ice administrative facility described in Section 32.18 provides a general purpose solution for administering individual Ice programs. IceGrid extends this functionality in several convenient ways:

- IceGrid automatically enables the facility in deployed servers.
- IceGrid uses the `Process` facet to terminate an active server, giving it an opportunity to perform an orderly shutdown.

- IceGrid provides a secure mechanism for invoking administrative operations on deployed servers.
- IceGrid administrative tools use the facility to display the properties of servers and services, and manipulate and monitor IceBox services.

We discuss each of these items in separate sections below.

38.22.1 Enabling the Facility

As we saw in Section 38.5.4, the configuration properties for a deployed server include definitions for the following properties:

- `Ice.Admin.Endpoints`
- `Ice.Admin.ServerId`

In conjunction with the `Ice.Default.Locator` property, these definitions satisfy the requirements explained in Section 32.18.2 for enabling the administrative facility. See Appendix D for more information on these properties.

Endpoints

If a server's descriptor does not supply a value for `Ice.Admin.Endpoints`, IceGrid supplies the default value shown below:

```
Ice.Admin.Endpoints=tcp -h 127.0.0.1
```

For the security reasons explained in Section 32.18.8, IceGrid specifies the local host interface (`127.0.0.1`) so that administrative access is limited to clients running on the same host. This configuration permits the IceGrid node to invoke operations on the server's `admin` object, but prevents remote access unless the client establishes an IceGrid administrative session (see Section 38.14).

Specifying a static port is unnecessary because the server registers its endpoints with IceGrid upon each new activation.

38.22.2 Server Deactivation

An IceGrid node uses the `Ice::Process` interface (see Section 32.18.4) to gracefully deactivate a server. In programs using Ice 3.3 or later, this interface is implemented by the administrative facet named `Process`. In earlier versions of Ice, an object adapter implemented this interface in a special servant if the adapter's `RegisterProcess` property was enabled.

Regardless of version, the Ice run time registers an `Ice::Process` proxy with the IceGrid registry when properly configured. Registration normally occurs during communicator initialization, but it can be delayed when a server needs to install its own administrative facets (see Section 32.18.7).

When the node is ready to deactivate a server, it invokes the `shutdown` operation on the server's `Ice::Process` proxy. If the server does not terminate in a timely manner, the node asks the operating system to terminate the process. Each server can be configured with its own deactivation timeout (see Section 38.17.19). If no timeout is configured, the node uses the value of the property `IceGrid.Node.WaitTime`, which defaults to `60` seconds.

If a server does not register an `Ice::Process` proxy, the IceGrid node cannot request a graceful termination and must resort instead to a more drastic, and potentially harmful, alternative by asking the operating system to terminate the server's process. On Unix, the node sends the `SIGTERM` signal to the process and, if the server does not terminate within the deactivation timeout period, sends the `SIGKILL` signal.

On Windows, the node first sends a `Ctrl+Break` event to the server and, if the server does not stop within the deactivation timeout period, terminates the process immediately.

Servers that disable the `Process` facet can install a signal handler in order to intercept the node's notification about pending deactivation. For example, portable C++ programs could use the `IceUtil::CtrlCHandler` class (see Section 31.10) for this purpose. However, we recommend that servers be allowed to use the `Process` facet when possible.

38.22.3 Administrative Requests

Section 38.22.1 explained the reasoning behind IceGrid's use of the local host interface when defining the endpoints of a deployed server's administrative object adapter. Briefly, this configuration allows local clients such as the IceGrid node to access the server's `admin` object while preventing direct invocations from remote clients. A server's `admin` object may still be accessed remotely, but only by clients that establish an IceGrid administrative session. To facilitate these requests, IceGrid uses an intermediary object that relays requests to the server via

its node. For example, Figure 38.10 illustrates the path of a `getProperty` invocation:

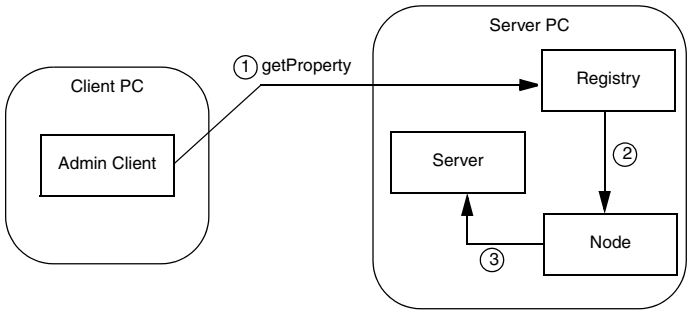


Figure 38.10. Routing for administrative requests on a server.

Obtaining a Proxy

During an administrative session, a client has two ways of obtaining the intermediary proxy for a server's `admin` object:

```
module IceGrid {
    interface Admin {
        idempotent string getServerAdminCategory();
        idempotent Object* getServerAdmin(string id)
            throws ServerNotExistException,
                   NodeUnreachableException,
                   DeploymentException;
        // ...
    };
};
```

If the client wishes to construct the proxy itself and already knows the server's id, the client need only modify the proxy of the `IceGrid::Admin` object with a new identity. The identity's category must be the return value of `getServerAdminCategory`, while its name is the id of the desired server. The example below demonstrates how to create the proxy and access the `Properties` facet of a server:

```
// C++
IceGrid::AdminSessionPrx session = ...;
IceGrid::AdminPrx admin = session->getAdmin();
Ice::Identity serverAdminId;
serverAdminId.category = admin->getServerAdminCategory();
```

```
serverAdminId.name = "MyServerId";
Ice::PropertiesAdminPrx props =
    Ice::PropertiesAdminPrx::checkedCast(
        admin->ice_identity(serverAdminId), "Properties");
```

Alternatively, the `getServerAdmin` operation returns a proxy that refers to the `admin` object of the given server. This operation performs additional validation and therefore may raise one of the exceptions shown in its signature above.

Callbacks without Glacier2

IceGrid also supports the relaying of callback requests from a back-end server to an administrative client over the client's existing connection to the registry, which is especially important for a client using a network port that is forwarded by a firewall or protected by a secure tunnel.

For this mechanism to work properly, a client that established its administrative session directly with IceGrid and not via a Glacier2 router must take additional steps to ensure that the proxies for its callback objects contain the proper identities and endpoints. The `IceGrid::AdminSession` interface provides an operation to help with the client's preparations:

```
module IceGrid {
interface AdminSession ... {
    idempotent Object* getAdminCallbackTemplate();
    // ...
};
};
```

As its name implies, the `getAdminCallbackTemplate` operation returns a *template proxy* that supplies the identity and endpoints a client needs to configure its callback objects. The information contained in the template proxy is valid for the lifetime of the administrative session. This operation returns a null proxy if the client's administrative session was established via a Glacier2 router, in which case the client should use the callback strategy described in the next section instead.

The endpoints contained in the template proxy are those of an object adapter in the IceGrid registry. The client must transfer these endpoints to the proxies for its callback objects so that callback requests from a server are sent first to IceGrid

and then relayed over a bidirectional connection (see Section 36.7) to the client, as shown in Figure 38.11.

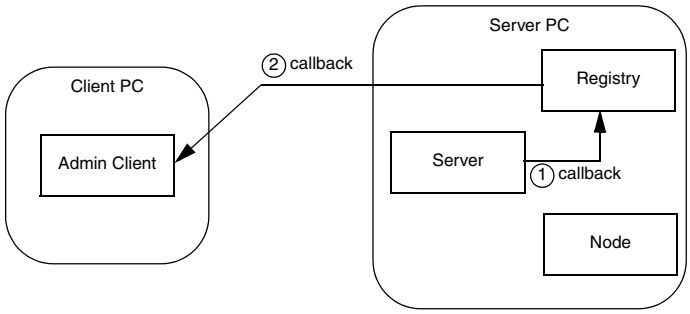


Figure 38.11. Routing for callback requests from a server.

The complete list of steps is shown below:

1. Invoke `getAdminCallbackTemplate` to obtain the template proxy.
2. Extract the category from the template proxy's identity and use it in all callback objects.
3. Extract the endpoints from the template proxy and use them to establish the published endpoints of the callback object adapter.
4. Create the callback object adapter and associate it with the administrative session's connection, thereby establishing a bidirectional connection with IceGrid.
5. Add servants to the callback object adapter.

As an example, let us assume that we have deployed an IceBox server with the server id `icebox1` and our objective is to register a `ServiceObserver` callback that monitors the state of the IceBox services (see Section 43.5.1). The first step is to obtain a proxy for the administrative facet named `IceBox.ServiceManager`:

```
// C++
IceGrid::AdminSessionPrx session = ...;
IceGrid::AdminPrx admin = session->getAdmin();
Ice::ObjectPrx obj = admin->getServerAdmin("icebox1");
IceBox::ServiceManagerPrx svcmgr =
    IceBox::ServiceManagerPrx::checkedCast(
        obj, "IceBox.ServiceManager");
```

Next, we retrieve the template proxy and compose the published endpoints for our callback object adapter:

```
Ice::ObjectPrx tmpl = admin->getAdminCallbackTemplate();
Ice::EndpointSeq endpts = tmpl->ice_getEndpoints();
string publishedEndpoints;
for (Ice::EndpointSeq::const_iterator p = endpts.begin();
     p != endpts.end(); ++p) {
    if (p == endpts.begin())
        publishedEndpoints = (*p)->toString();
    else
        publishedEndpoints += ":" + (*p)->toString();
}
communicator->getProperties()->setProperty(
    "CallbackAdapter.PublishedEndpoints", publishedEndpoints);
```

The final steps involve creating the callback object adapter, adding a servant, establishing the bidirectional connection and registering our callback with the service manager:

```
Ice::ObjectAdapterPtr callbackAdapter =
    communicator->createObjectAdapter("CallbackAdapter");
Ice::Identity cbid;
cbid.category = tmpl->ice_getIdentity().category;
cbid.name = "observer";
IceBox::ServiceObserverPtr obs = new ObserverI;
Ice::ObjectPrx cbobj = callbackAdapter->add(obs, cbid);
IceBox::ServiceObserverPrx cb =
    IceBox::ServiceObserverPrx::uncheckedCast(cbobj);
callbackAdapter->activate();
session->ice_getConnection()->setAdapter(callbackAdapter);
svcmgr->addObserver(cb);
```

At this point the client is ready to receive callbacks from the IceBox server whenever one of its services changes state.

Callbacks with Glacier2

A client that creates an administrative session via a Glacier2 router (see Section 38.15) already has a bidirectional connection over which callbacks from administrative facets are relayed. The flow of requests is shown in Figure 38.12,

which presents a simplified view with the router and IceGrid services all running on the same host.

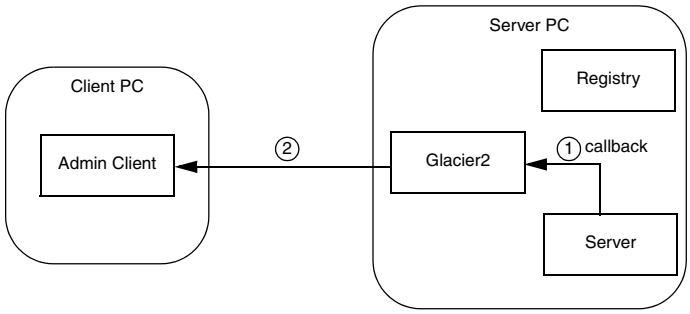


Figure 38.12. Routing for callback requests from a server.

To prepare for receiving callbacks, the client must perform the same steps as for any router client (see Section 42.4):

1. Obtain a proxy for the router.
2. Retrieve the category to be used in callback objects.
3. Create the callback object adapter and associate it with the router, thereby establishing a bidirectional connection.
4. Add servants to the callback object adapter.

Repeating the example from the previous section, we assume that we have deployed an IceBox server with the server id `icebox1` and our objective is to register a `ServiceObserver` callback that monitors the state of the IceBox services (see Section 43.5.1). The first step is to obtain a proxy for the administrative facet named `IceBox.ServiceManager`:

```
// C++
IceGrid::AdminSessionPrx session = ...;
IceGrid::AdminPrx admin = session->getAdmin();
Ice::ObjectPrx obj = admin->getServerAdmin("icebox1");
IceBox::ServiceManagerPrx svcmgr =
    IceBox::ServiceManagerPrx::checkedCast(
        obj, "IceBox.ServiceManager");
```

Now we are ready to create the object adapter and register the observer:

```
Ice::RouterPrx router = communicator->getDefaultRouter();
Ice::ObjectAdapterPtr callbackAdapter =
    communicator->createObjectAdapterWithRouter(
        "CallbackAdapter", router);
Ice::Identity cbid;
cbid.category = router->getCategoryForClient();
cbid.name = "observer";
IceBox::ServiceObserverPtr obs = new ObserverI;
Ice::ObjectPrx cbobj = callbackAdapter->add(obs, cbid);
IceBox::ServiceObserverPrx cb =
    IceBox::ServiceObserverPrx::uncheckedCast(cbobj);
callbackAdapter->activate();
svcmgr->addObserver(cb);
```

At this point the client is ready to receive callbacks from the IceBox server whenever one of its services changes state.

38.22.4 Using the Facility in IceGrid Utilities

This section discusses the ways in which the IceGrid utilities make use of the administrative facility. See Section 38.24 for more information on the administrative utilities.

Properties

The command line and graphical utilities allow you to explore the configuration properties of a server or service.

One property in particular, `BuildId`, is given special consideration by the graphical utility. Although it is not used by the Ice run time, the `BuildId` property gives you the ability to describe the build configuration of your application. The property's value is shown by the graphical utility in its own field in the attributes of a server or service, as well as in the list of properties. You can also retrieve the value of this property using the command-line utility with the following statement:

```
> server property MyServerId BuildId
```

Or, for an IceBox service, with this command:

```
> service property MyServerId MyService BuildId
```

Section 32.18.5 describes the Slice interface that the utilities use to access these properties, and Section 38.22.3 explains how to obtain an appropriate proxy.

Administering IceBox Services

IceBox provides an administrative facet that implements the `IceBox::ServiceManager` interface, which supports operations for stopping an active service, and for starting a service that is currently inactive. These operations are available in both the command line and graphical utilities.

IceBox also defines a `ServiceObserver` interface for receiving callbacks when services are stopped or started. The graphical utility implements this interface so that it can present an updated view of the state of an IceBox server. Section 38.22.3 includes examples that demonstrate how to register an observer with the IceBox administrative facet.

See Chapter 43 for more information on IceBox.

38.23 Securing IceGrid

IceGrid's registry and node services expose multiple network endpoints that a malicious client could use to gain access to IceGrid functionality and interfere with deployed applications. This presents a significant security risk in network environments that are exposed to untrusted clients. For example, a malicious client could connect to a node and use IceGrid's internal interfaces to deploy and run its own server executable.

Using a firewall is one way to prevent unauthorized use of IceGrid's facilities. Another solution is to use IceSSL: you can generate SSL certificates for each component and configure them to trust and accept connections only from other authorized components. The remainder of this section discusses the IceSSL solution but also provides useful information for those interested in securing IceGrid with a firewall.

To restrict access using IceSSL, we need to establish trust relationships between IceGrid registry replicas, nodes, and deployed servers. IceSSL allows us to do this using configuration properties, as described in Section 41.4.6. The trust relationships are based on the information contained in SSL certificates.

There are several possible strategies for generating certificates. At a minimum you will need the following:

- one certificate for all of the registries
- one certificate for all of the nodes
- one certificate for all of the servers managed by IceGrid

The certificates that you generate for registries and nodes should be protected with a password to ensure that only privileged users can start these services. However, we do not recommend using a password to protect the certificate for deployed servers because it would need to be specified in clear text in each server's configuration (servers that are activated by IceGrid must not prompt for a password). Furthermore, this password might appear in multiple places, such as an XML descriptor file, the IceGrid registry database, and property files generated by IceGrid nodes. The complexity involved in protecting access to every file that contains a clear text password could be overwhelming. Instead, we recommend that you protect access to the server certificate using file system permissions.

Depending on your organization and the roles of each person that uses IceGrid, you may decide to create additional certificates. For example, you might create a unique certificate for each IceGrid node instance if you deploy nodes on end-user machines and wish to configure the IceGrid registry to authorize connections only from the nodes of trusted users.

You can use the **`iceca`** script to establish a certificate authority and generate certificates (see Section 41.7). The sections that follow describe the interactions between the registry, node, and servers, and show how to configure IceSSL to restrict access to trusted peers. For the purposes of this discussion, we assume that the SSL certificates use the common names shown below:

- `IceGrid Registry`
- `IceGrid Node`
- `Server`

The Ice distribution includes a C++ example that demonstrates how to configure a secure IceGrid deployment (see `demo/IceGrid/secure`). The example includes a script to generate certificates for a registry, a node, a Glacier2 router, and a server. For more information, see the `README` file provided with the example.

38.23.1 Registry Endpoints

The IceGrid registry has three mandatory endpoints representing the client, server, and internal endpoints. The registry also has two optional endpoints (the session manager and administrative session manager endpoints) that are only useful when IceGrid is accessed via Glacier2 (see Section 38.15).

Client Endpoint

The registry client endpoint is used by Ice applications that create client sessions in order to use the object allocation facility. It is also used by administrative clients that create sessions for managing the registry. Finally, the client endpoint is used by Ice applications that use the `IceGrid::Query` interface or resolve indirect proxies via the IceGrid locator.

Two distinct permission verifiers authorize the creation of client sessions (see Section 38.11.2) and administrative sessions (see Section 38.14.1). The remaining functionality available via the client endpoint, such as resolving objects and object adapters using the `IceGrid::Query` interface or the Ice locator mechanism, is accessible to any client that is able to connect to the client endpoint.

It is safe to use an insecure transport for the client endpoint if it is only being used for locator queries. However, you should use a secure transport if you have enabled client and administrative sessions (by configuring the appropriate permission verifiers). Creating a session over an insecure transport poses a security risk because the user name and password are sent in clear text over the network.

If you include secure and insecure transports in the registry's client endpoints, you should ensure that applications that need to authenticate with IceGrid permission verifiers use a secure transport (see Section 41.4.7 for more information on configuring proxies to use a secure connection).

It is not necessary to restrict SSL access to the client endpoints (using the property `IceSSL.TrustOnly.Server.IceGrid.Registry.Client`) as long as you use client and administrative permission verifiers for authentication. This property is only useful for restricting access to client and administrative sessions when using null permission verifiers. Note however that if both client and administrative sessions are enabled, you will only be able to restrict access to one set of clients since you cannot distinguish clients that create client sessions from clients that create administrative sessions.

Server Endpoint

Ice servers use the registry's server endpoint to register their object adapter endpoints and send information to administrative clients connected via the registry (see Section 38.22).

Securing this endpoint with IceSSL is necessary to prevent a malicious program from potentially hijacking a server by registering its endpoints first. The property definition shown below demonstrates how to limit access to this endpoint to trusted Ice servers:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.Server=CN="Server"
```

Internal Endpoint

IceGrid nodes and registry replicas use the internal endpoint to communicate with the registry. For example, nodes connect to the internal endpoint of each active registry, and registry slaves establish a session with their master via this endpoint.

The internal endpoint must be secured with IceSSL to prevent malicious Ice applications from gaining access to sensitive functionality that is intended to be used only by nodes and registry replicas. You can restrict access to this endpoint with the following property:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.Internal=CN="IceGrid
Node";CN="IceGrid Registry"
```

Session Manager Endpoint

The session manager endpoint is used by Glacier2 to create IceGrid client sessions (see Section 38.15.3). The functionality exposed by this endpoint is unrestricted so you must either secure it or disable it (this endpoint is disabled by default). The property shown below demonstrates how to configure IceSSL so that only Glacier2 routers are accepted by this endpoint:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.SessionManager=CN="Glacie
r2 Router Client"
```

In this example, `Glacier2 Router Client` is the common name of the Glacier2 router used by clients to create IceGrid client sessions.

Administrative Session Manager Endpoint

Glacier2 routers use the registry's administrative session manager endpoint to create IceGrid administrative sessions (see Section 38.15.2). The functionality exposed by this endpoint is unrestricted, so you must either secure it or disable it (this endpoint is disabled by default). The property shown below demonstrates how to configure IceSSL so that only Glacier2 routers are accepted by this endpoint:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.AdminSessionManager=CN="G
lacier2 Router Admin"
```

In this example, `Glacier2 Router Admin` is the common name of the Glacier2 router used by clients to create IceGrid administrative sessions. Note that if you use a single Glacier2 router instance for both client and administrative sessions (see Section 38.15.4), you will need to use the same common name to restrict access to both session manager endpoints:

```
IceSSL.TrustOnly.Server.IceGrid.Registry.SessionManager=CN="Glacie
r2 Router Client"
IceSSL.TrustOnly.Server.IceGrid.Registry.AdminSessionManager=CN="G
lacier2 Router Client"
```

Outgoing Connections

The registry establishes outgoing connections to other registries and nodes. You should configure the `IceSSL.TrustOnly.Client` property to restrict connections to these trusted peers:

```
IceSSL.TrustOnly.Client=CN="IceGrid Registry";CN="IceGrid Node"
```

The registry can also connect to Glacier2 routers and permission verifier objects. To allow connections to these services, you must include in this property the common names of Glacier2 routers that create client or administrative sessions, as well as the common names of servers that host the permission verifier objects.

38.23.2 Node Endpoints

An IceGrid node has only one endpoint, which is used for internal communications with the registry. As a result, it should be configured to accept connections only from IceGrid registries:

```
IceSSL.TrustOnly.Server=CN="IceGrid Registry"
```

A node also establishes outgoing connections to the registry's internal endpoint, as well as the `Ice.Admin` endpoint of deployed servers (see Section 38.23.3). You should configure the `IceSSL.TrustOnly.Client` property as shown below to verify the identity of these peers:

```
IceSSL.TrustOnly.Client=CN="Server";CN="IceGrid Registry"
```

38.23.3 Server's `Ice.Admin` Endpoints

By default, IceGrid sets the endpoints of a deployed server's `Ice.Admin` adapter to `tcp -h 127.0.0.1`. This setting is already quite secure because it only accepts connections from processes running on the same host. However, since you already need to configure IceSSL so that a server can authenticate with the IceGrid registry (servers connect to the registry to register their endpoints), you might as well use a secure endpoint for the `Ice.Admin` adapter and configure it accept connections only from IceGrid nodes:

```
IceSSL.TrustOnly.Server.Ice.Admin=CN="IceGrid Node"
```

This is only necessary if the `Ice.Admin` endpoint is enabled (which it is by default).

You can also set the `IceSSL.TrustOnly.Client` property so that the server is only permitted to connect to the IceGrid registry:

```
IceSSL.TrustOnly.Client=CN="IceGrid Registry"
```

If your server invokes on other servers, you will need to modify this setting to allow secure connections to them.

38.24 Administrative Utilities

IceGrid provides two administrative clients: a command-line tool and a graphical application.

38.24.1 Command Line Client

The **`icegridadmin`** utility is a command-line tool for administering an IceGrid domain. Deploying an application with this utility requires an XML file that defines the descriptors.

Usage

The IceGrid administration tool supports the following command-line options:

```
Usage: icegridadmin [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
-e COMMANDS          Execute COMMANDS.
-d, --debug          Print debug messages.
-s, --server         Start icegridadmin as a server (to parse XML
                     files).
-u, --username       Login with the given username.
-p, --password       Login with the given password.
-S, --ssl            Authenticate through SSL.
-r, --replica NAME   Connect to the replica NAME.
```

The **`-e`** option causes the tool to execute the given commands and then exit without entering an interactive mode. The **`-s`** option starts **`icegridadmin`** in a server mode that supports the `IceGrid::FileParser` interface; a proxy for the

object is printed to standard output. If neither **-e** nor **-s** is specified, the tool enters an interactive mode in which you issue commands at a prompt.

To communicate with the IceGrid registry, **icegridadmin** establishes an administrative session as described in Section 38.14.1. The tool uses SSL authentication if you specify the **-S** option or define its equivalent property `IceGridAdmin.AuthenticateUsingSSL`. Otherwise, **icegridadmin** uses password authentication and prompts you for the username and password if you do not specify them via command-line options or properties. If you want **icegridadmin** to establish its session using a Glacier2 router, define `Ice.Default.Router` appropriately. See Section D.16 for more information on the tool's configuration properties.

Once the session is successfully established, **icegridadmin** displays its command prompt. The **help** command displays the following usage information:

help

Print this message.

exit, **quit**

Exit this program.

CATEGORY help

Print the help section of the given **CATEGORY**.

COMMAND help

Print the help of the given **COMMAND**.

The tool's commands are organized by category. The supported command categories are shown below:

- **application**
- **node**
- **registry**
- **server**
- **service**
- **adapter**
- **object**
- **server template**
- **service template**

You can obtain more information about each category using the **help** command:

```
>>> application help
```

Application Commands

application add [-n | --no-patch] DESC [TARGET ...] [NAME=VALUE ...]

Add applications described in the XML descriptor file **DESC**. If specified the optional targets are deployed. Variables are defined using the **NAME=VALUE** syntax. The application is automatically patched unless the **-n** or **--no-patch** option is used to disable it (see Section 38.13).

application remove NAME

Remove the application named **NAME**.

application describe NAME

Describe the application named **NAME**.

application diff DESC [TARGET ...] [NAME=VALUE ...]

Print the differences between the application in the XML descriptor file **DESC** and the current deployment. Variables are defined using the **NAME=VALUE** syntax.

application update DESC [TARGET ...] [NAME=VALUE ...]

Update the application in the XML descriptor file **DESC**. Variables are defined using the **NAME=VALUE** syntax.

application patch [-f | --force] NAME

Patch the application named **NAME**. If **-f** or **--force** is specified, IceGrid will first shut down any servers that depend on the data to be patched.

application list

List all deployed applications.

Node Commands

node list

List all registered nodes.

`node describe NAME`

Show information about node **`NAME`**.

`node ping NAME`

Ping node **`NAME`**.

`node load NAME`

Print the load of the node **`NAME`**.

`node processors [NAME]`

Print the number of processor sockets for node **`NAME`**. If **`NAME`** is omitted, print the number of processor sockets for each node. (The `IceGrid.Node.ProcessorSocketCount` property allows you to explicitly set this value for systems where the number of sockets cannot be obtained programatically.)

`node show [OPTIONS] NAME [stderr | stdout]`

Print the text from the node's standard error or standard output. The supported options are shown below:

`-f, --follow`

Wait for new text to be available.

`-t, --tail N`

Print the last **`N`** lines of text.

`-h, --head N`

Print the first **`N`** lines of text.

`node shutdown NAME`

Shutdown node **`NAME`**.

Registry Commands

`registry list`

List all registered registries.

registry describe NAME

Show information about registry **NAME**.

registry ping NAME

Ping registry **NAME**.

registry show [OPTIONS] NAME [stderr | stdout]

Print the text from the registry's standard error or standard output. The supported options are shown below:

-f, --follow

Wait for new text to be available.

-t, --tail N

Print the last **N** lines of text.

-h, --head N

Print the first **N** lines of text.

registry shutdown NAME

Shutdown registry **NAME**.

Server Commands

server list

List all registered servers.

server remove ID

Remove server **ID**.

server describe ID

Describe server **ID**.

server properties ID

Get the run-time properties of server **ID**.

server property ID NAME

Get the run-time property **NAME** of server **ID**.

server state ID

Get the state of server **ID**.

server pid ID

Get the process id of server **ID**.

server start ID

Start server **ID**.

server stop ID

Stop server **ID**.

server patch ID

Patch server **ID**.

server signal ID SIGNAL

Send **SIGNAL** (such as `SIGTERM` or `15`) to server **ID**.

server stdout ID MESSAGE

Write **MESSAGE** on server **ID**'s standard output.

server stderr ID MESSAGE

Write **MESSAGE** on server **ID**'s standard error.

server show [OPTIONS] ID [stderr | stdout | LOGFILE]

Print the text from the server's standard error, standard output, or the log file **LOGFILE**. The supported options are shown below:

-f, --follow

Wait for new text to be available.

-t, --tail N

Print the last **N** lines of text.

-h, --head N

Print the first **N** lines of text.

`server enable ID`

Enable server **`ID`**.

`server disable ID`

Disable server **`ID`** (a disabled server can't be started on demand or administratively).

Service Commands

`service start ID NAME`

Starts service **`NAME`** in IceBox server **`ID`**.

`service stop ID NAME`

Stops service **`NAME`** in IceBox server **`ID`**.

`service describe ID NAME`

Describes service **`NAME`** in IceBox server **`ID`**.

`service properties ID NAME`

Get the run-time properties of service **`NAME`** from IceBox server **`ID`**.

`service property ID NAME PROPERTY`

Get the run-time property **`PROPERTY`** of service **`NAME`** from IceBox server **`ID`**.

`service list ID`

List the services in IceBox server **`ID`**.

Adapter Commands

`adapter list`

List all registered adapters.

`adapter endpoints ID`

Show the endpoints of adapter or replica group **`ID`**.

`adapter remove ID`

Remove adapter or replica group **`ID`**.

Object Commands

`object add PROXY [TYPE]`

Add a well-known object to the registry, optionally specifying its type.

`object remove IDENTITY`

Remove a well-known object from the registry.

`object find TYPE`

Find all well-known objects with the type **`TYPE`**.

`object describe EXPR`

Describe all well-known objects whose stringified identities match the expression **`EXPR`**. A trailing wildcard is supported in **`EXPR`**, for example "`object describe Ice*`".

`object list EXPR`

List all well-known objects whose stringified identities match the expression **`EXPR`**. A trailing wildcard is supported in **`EXPR`**, for example "`object list Ice*`".

Server Template

`server template instantiate APPLICATION NODE TEMPLATE [NAME=VALUE ...]`

Instantiate the requested server template defined in the given application on a node. Variables are defined using the **`NAME=VALUE`** syntax.

`server template describe APPLICATION TEMPLATE`

Describe a server template **`TEMPLATE`** from the given application.

Service Template

`service template describe APPLICATION TEMPLATE`

Describe a service template **`TEMPLATE`** from the given application.

Configuration

`icegridadmin` requires that the locator proxy be defined in the configuration property `Ice.Default.Locator`. If a configuration file already exists that

defines this property, you can start **icegridadmin** using the configuration file as shown below:

```
$ icegridadmin --Ice.Config=<file>
```

Otherwise, you can define the property on the command line:

```
$ icegridadmin --Ice.Default.Locator=<proxy>
```

Section 38.4.3 describes how to configure the `Ice.Default.Locator` property for an IceGrid client.

38.24.2 Graphical Client

The graphical administration tool, IceGrid Admin, allows you to perform anything that you can do from the command line via a GUI. Please refer to the instructions included with your Ice distribution for details on how to start the administration tool.

38.25 Server Activation

You can choose among four activation modes for servers deployed and managed by an IceGrid node:

- manual

  You must start the server explicitly via the IceGrid GUI or **icegridadmin**, or programmatically via the `IceGrid::Admin` interface.

- always

  IceGrid activates the server when its node starts. If the server stops, IceGrid automatically reactivates it.

- on demand

  IceGrid activates the server when a client invokes an operation on an object in the server.

- session

  This mode also provides on-demand activation but requires the server to be allocated by a session.

38.25.1 Activation in Detail

On-demand server activation is a valuable feature of distributed computing architectures for a number of reasons:

- It minimizes application startup times by avoiding the need to pre-start all servers.
- It allows administrators to use their computing resources more efficiently because only those servers that are actually needed are running.
- It provides more reliability in the case of some server failure scenarios, e.g., the server is reactivated after a failure and may still be capable of providing some services to clients until the failure is resolved.
- It allows remote activation and deactivation.

On-demand activation occurs when an Ice client requests the endpoints of one of the server's object adapters via a locate request (see Section 38.3.1). If the server is not active at the time the client issues the request, the node activates the server and waits for the target object adapter to register its endpoints. Once the object adapter endpoints are registered, the registry returns the endpoint information back to the client. This sequence ensures that the client receives the endpoint information *after* the server is ready to receive requests.

38.25.2 Requirements

In order to use on-demand activation for an object adapter, the adapter must have an identifier and be entered in the IceGrid registry.

When using session activation mode, IceGrid requires that the server be allocated; on-demand activation fails for servers that have not been allocated. (See Section 38.11 for more information on server allocation.)

The session activation mode recognizes an additional reserved variable in the server descriptor, `${session.id}`. The value of this variable is the user ID or, for SSL sessions, the distinguished name associated with the session.

38.25.3 Efficiency

Once a server is activated, it remains running indefinitely (unless it uses the session activation mode). A node deactivates a server only when explicitly requested to do so (see Section 32.17.6). As a result, server processes tend to accumulate on the node's host.

One of the advantages of on-demand activation is the ability to manage computing resources more efficiently. Of course there are many aspects to this, but Ice makes one technique particularly simple: servers can be configured to terminate gracefully after they have been idle for a certain amount of time.

A typical scenario involves a server that is activated on demand, used for a while by one or more clients, and then terminated automatically when no requests have been made for a configurable number of seconds. All that is necessary is setting the server's configuration property `Ice.ServerIdleTime` to the desired idle time. See Appendix D for more information on this property.

For a server activated in session activation mode, IceGrid deactivates the server when the session releases the server or when the session is destroyed.

38.25.4 Activating Servers with Specific User IDs

On Unix platforms you can activate server processes with specific effective user IDs, provided that the IceGrid node is running as root. If the IceGrid node does not run as root, servers are always activated with the effective user ID of the IceGrid node process. (The same is true for Windows—servers always run with the same user ID as the IceGrid node process.)

For the remainder of this section, we assume that the node runs as root on a Unix machine.

The `user` attribute of the `server` descriptor specifies the user ID for a server. If this attribute is not specified and the activation mode is not `session`, the default value is `nobody`. Otherwise, the default value is `${session.id}` if the activation mode is `session`.

Since individual users often have different account names and user IDs on different machines, IceGrid provides a mechanism to map the value of the `user` attribute in the `server` descriptor to a user account. To do this, you must configure the node to use a user account mapper object. This object must implement the `IceGrid::UserAccountMapper` interface:

```
exception UserAccountNotFoundException {};

interface UserAccountMapper {
    string getUserAccount(string user)
            throws UserAccountNotFoundException;
};
```

The IceGrid node invokes `getUserAccount` and passes the value of the `server` descriptor's `user` attribute. The return value is the name of the user account.

IceGrid provides a built-in file-based user account mapper that you can configure for the node and the registry. The file contains any number of user–account-ID pairs. Each pair appears on a separate line, with white space separating the user account from the identifier. For example, the file shown below contains two entries that map two distinguished names to the user account `lisa`:

```
lisa O=ZeroC\\, Inc., OU=Ice, CN=Lisa
lisa O=ZeroC\\, Inc., OU=Ice, CN=Lisa S.
```

The distinguished names must be unique. If the same distinguished name appears several times in a file, the last entry is used.

You can specify the path of the user account file with the `IceGrid.Registry.UserAccounts` property for the registry and the `IceGrid.Node.UserAccounts` property for a node.

To configure an IceGrid node to use the IceGrid registry file-based user account mapper, you need to set the `IceGrid.Node.UserAccountMapper` property to the well-known proxy `IceGrid/RegistryUserAccountMapper`. Alternatively, you can set this property to the proxy of your own user account mapper object. Note that if this property is set, the node ignores the setting of `IceGrid.Node.UserAccounts`.

38.25.5 Endpoint Registration

Section 32.17.5 discusses the configuration requirements for enabling automatic endpoint registration in servers. It should be noted however that IceGrid simplifies the configuration process in two ways:

1. The IceGrid deployment mechanism automates the creation of a configuration file for the server, including the definition of object adapter identifiers and endpoints (see Section 38.4.4).

2. A server that is activated automatically by an IceGrid node does not need to explicitly configure a proxy for the locator because the IceGrid node defines it in the server's configuration file.

38.26 Solving Problems

38.26.1 Activation Failure

Server activation failure is usually indicated by the receipt of a `NoEndpointException`. This can happen for a number of reasons, but the most likely cause is an incorrect configuration. For example, an IceGrid node may fail to activate a server because the server's executable file, shared libraries, or classes could not be found. There are several steps you can take in this case:

1. Enable activation tracing in the node by setting the configuration property `IceGrid.Node.Trace.Activator=3`.
2. Examine the tracing output and verify the server's command line and working directory are correct.
3. Relative pathnames specified in a command line may not be correct relative to the node's current working directory. Either replace relative pathnames with absolute pathnames, or restart the node in the proper working directory.
4. Verify that the server is configured with the correct `PATH` or `LD_LIBRARY_PATH` settings for its shared libraries. For a Java server, its `CLASSPATH` may also require changes.

Another cause of activation failure is a server fault during startup. After you have confirmed that the node successfully spawns the server process using the steps above, you should then check for signs of a server fault (e.g., on Unix, look for a `core` file in the node's current working directory). See Section 38.26.3 for more information on server failures.

38.26.2 Proxy Failure

A client may receive `Ice::NotRegisteredException` if binding fails for an indirect proxy (see Section 32.17.2). This exception indicates that the proxy's object identity or object adapter is not known by the IceGrid registry. The following steps may help you discover the cause of the exception:

1. Use **`icegridadmin`** (see Section 38.24.1) to verify that the object identity or object adapter identifier is actually registered, and that it matches what is used by the proxy:

```
>>> adapter list
...
>>> object find ::Hello
```

. . .

2. If the problem persists, review your configuration to ensure that the locator proxy used by the client matches the registry's client endpoints, and that those endpoints are accessible to the client (i.e., are not blocked by a firewall).
3. Finally, enable locator tracing in the client by setting the configuration property `Ice.Trace.Locator=2`, then run the client again to see if any log messages are emitted that may indicate the problem.

38.26.3 Server Failure

Diagnosing a server failure can be difficult, especially when servers are activated automatically on remote hosts. Here are a few suggestions:

1. If the server is running on a Unix host, check the current working directory of the IceGrid node process for signs of a server failure, such as a `core` file.
2. Judicious use of tracing can help to narrow the search. For example, if the failure occurs as a result of an operation invocation, enable protocol tracing in the Ice run time by setting the configuration property `Ice.Trace.Protocol=1` to discover the object identity and operation name of all requests.

   Of course, the default log output channels (standard out and standard error) will probably be lost if the server is activated automatically, so either start the server manually (see below) or redirect the log output (see Appendix D for a description of the `Ice.UseSyslog` property).

   You can also use the `Ice::Logger` interface to emit your own trace messages.
3. Run the server in a debugger; a server configured for automatic activation can also be started manually if necessary. However, since the IceGrid node did not activate the server, it cannot monitor the server process and therefore will not know when the server terminates. This will prevent subsequent activation unless you clean up the IceGrid state when you have finished debugging and terminated the server. You can do this by starting the server using **`icegridadmin`** (see Section 38.24.1):

   ```
   >>> server start TheServer
   ```

   This will cause the node to activate (and therefore monitor) the server process. If you do not want to leave the server running, you can stop it with the **`server stop`** command.

4. After the server is activated and is in a quiescent state, attach your debugger to the running server process. This avoids the issues associated with starting the server manually (as described in the previous step), but does not provide as much flexibility in customizing the server's startup environment.

Another cause for a server to fail to activate correctly is if there is a mismatch in the adapter names used by the server for its adapters, and the adapter names specified in the server's deployment descriptor. After starting a server process, the node waits for the server to activate all its object adapters and report them as ready; if the server does not do this, the node reports a failure once a timeout expires. The timeout is controlled by the setting of the property `IceGrid.Node.WaitTime`. (The default value is 60 seconds.)

You can check the status of each of a server's adapters using **`icegridadmin`** or the GUI tool. While the node waits for an adapter to be activated by the server, it reports the status of the adapter as "activating". If you experience timeouts before each adapter's status changes to "active", the most likely cause is that the deployment descriptor for the server either mentions more object adapters than are actually created by the server, or that the server uses a name for one or more adapters that does not match the corresponding name in the deployment descriptor.

38.26.4 Disabling Faulty Servers

You may find it necessary to disable a server that terminates in an error condition. For example, on a Unix platform each server failure might result in the creation of a new (and potentially quite large) core file. This problem is exacerbated when the server is used frequently, in which case repeated cycles of activation and failure can consume a great deal of disk space and threaten the viability of the application as a whole.

As a defensive measure, you can configure an IceGrid node to disable these servers automatically using the `IceGrid.Node.DisableOnFailure` property. In the disabled state, a server cannot be activated on demand. The default value of the property is zero, meaning the node does not disable a server that terminates improperly. A positive value causes the node to temporarily disable a faulty server, with the value representing the number of seconds the server should remain disabled. If the property has a negative value, the server is disabled indefinitely, or until the server is explicitly enabled or started via an administrative action.

38.27 Summary

This chapter provided a detailed discussion of IceGrid, including the modifications required to incorporate IceGrid into client and server applications, as well as the configuration and administration of IceGrid components. Once you understand the basic concepts on which IceGrid is founded, you quickly begin to appreciate the flexibility, power, and convenience of IceGrid's capabilities:

- Replication, load balancing, and automatic server activation increase reliability and make more efficient use of processing resources.
- The location service simplifies administrative requirements and minimizes coupling between clients and servers.
- Deploying an application can be done using easily-understood, reusable XML files, or interactively using a graphical client.
- Distributing and updating executables, dependent libraries and other files on all nodes can be automated and managed remotely.

In short, IceGrid provides the tools you need to develop robust, enterprise-class Ice applications.

Chapter 39
Freeze

39.1 Chapter Overview

This chapter describes how to use Freeze to add persistence to Ice applications. Section 39.3 presents the Freeze evictors, and Section 39.4 demonstrates a Freeze evictor in another file system implementation. Section 39.5 discusses Freeze maps and shows how to use them in a simple example. Section 39.6 examines an implementation of the file system using a Freeze map. Finally, Sections 39.7 and 39.7 describes Freeze catalogs and backups.

39.2 Introduction

Freeze represents a set of persistence services, as shown in Figure 39.1.

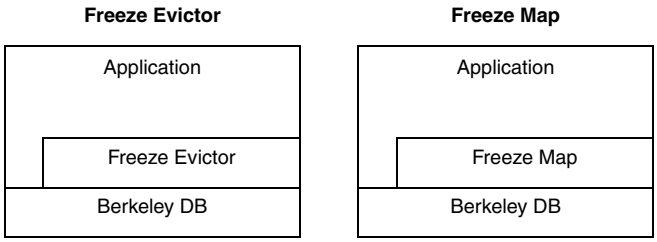


Figure 39.1. Layer diagram for Freeze persistence services.

The Freeze persistence services are described below:

- The Freeze evictor is a highly-scalable implementation of an Ice servant locator that provides automatic persistence and eviction of servants with only minimal application code.
- The Freeze map is a generic associative container. Code generators are provided that produce type-specific maps for Slice key and value types. Applications interact with a Freeze map just like any other associative container, except the keys and values of a Freeze map are persistent.

As you will see from the examples in this chapter, integrating a Freeze map or evictor into your Ice application is quite straightforward: once you define your persistent data in Slice, Freeze manages the mundane details of persistence.

Freeze is implemented using Berkeley DB, a compact and high-performance embedded database. The Freeze map and evictor APIs insulate applications from the Berkeley DB API, but do not prevent applications from interacting directly with Berkeley DB if necessary.

39.3 Freeze Evictors

Freeze evictors combine persistence and scalability features into a single facility that is easily incorporated into Ice applications.

As an implementation of the `ServantLocator` interface (see Section 32.7), a Freeze evictor takes advantage of the fundamental separation between Ice object

and servant to activate servants on demand from persistent storage, and to deactivate them again using customized eviction constraints. Although an application may have thousands of Ice objects in its database, it is not practical to have servants for all of those Ice objects resident in memory simultaneously. The application can conserve resources and gain greater scalability by setting an upper limit on the number of active servants, and letting a Freeze evictor handle the details of servant activation, persistence, and deactivation.

39.3.1 Specifying Persistent State

The persistent state of servants managed by a Freeze evictor must be described in Slice. Specifically, every servant must implement a Slice class, and a Freeze evictor automatically stores and retrieves all the (Slice-defined) data members of these Slice classes. Data members that are not specified in Slice are not persistent.

A Freeze evictor relies on the Ice object factory facility to load persistent servants from disk: the evictor creates a brand new servant using the registered factory and then restores the servant's data members. Therefore, for every persistent servant class you define, you need to register a corresponding object factory with the Ice communicator. (See the relevant language mapping chapter for more details on object factories.)

39.3.2 Servant Association

With a Freeze evictor, each <*object identity*, *facet*> pair is associated with its own dedicated persistent object (servant). Such a persistent object cannot serve several identities or facets. Each servant is loaded and saved independently of other servants; in particular, there is no special grouping for the servants that serve the facets of a given Ice object.

Like an object adapter, the Freeze evictor provides operations named `add`, `addFacet`, `remove`, and `removeFacet`. They have the same signature and semantics, except that with the Freeze evictor, the mapping and the state of the mapped servants is stored in a database.

39.3.3 Background Save and Transactional Evictors

Freeze provides two types of evictors, a background save evictor and a transactional evictor, with corresponding local interfaces: `BackgroundSaveEvictor` and `TransactionalEvictor`. These two local interfaces derive from the

`Freeze::Evictor` local interface, which defines most evictor operations, in particular `add`, `addFacet`, `remove`, and `removeFacet`.

Furthermore, the on-disk format of these two types of evictors is the same: you can switch from one type of evictor to the other without any data transformation.

Background Save Evictor

A background-save evictor keeps all its servants in a map and writes the state of newly-created, modified, and deleted servants to disk asynchronously, in a background thread. You can configure how often servants are saved; for example you could decide to save every three minutes, or whenever ten or more servants have been modified. For applications with frequent updates, this allows you to group many updates together to improve performance.

The downside of the background-save evictor is recovery from a crash. Because saves are asynchronous, there is no way to force an immediate save to preserve a critical update. Moreover, you cannot group several related updates together: for example, if you transfer funds between two accounts (servants) and a crash occurs shortly after this update, it is possible that, once your application comes back up, you will see the update on one account but not on the other. Your application needs to handle such inconsistencies when restarting after a crash.

Similarly, a background-save evictor provides no ordering guarantees for saves. If you update servant 1, servant 2, and then servant 1 again, it is possible that, after recovering from a crash, you will see the latest state for servant 1, but no updates at all for servant 2.

Transactional Evictor

A transactional evictor maintains a servant map as well, but only keeps read-only servants in this map. The state of these servants corresponds to the latest data on disk. Any servant creation, update, or deletion is performed within a database transaction. This transaction is committed (or rolled back) immediately, typically at the end of the dispatch of the current operation, and the associated servants are then discarded.

With such an evictor, you can ensure that several updates, often on different servants (possibly managed by different transactional evictors) are grouped together: either all or none of these updates occur. In addition, updates are written almost immediately, so crash recovery is a lot simpler: few (if any) updates will be lost, and you can maintain consistency between related persistent objects.

However, an application based on a transactional evictor is likely to write a lot more to disk than an application with a background-save evictor, which may have an adverse impact on performance.

39.3.4 Eviction Strategy

Both background-save and transactional evictors associate a queue with their servant map and manage this queue using a "least recently used" eviction algorithm: if the queue is full, the least recently used servant is evicted to make room for a new servant.

Here is the sequence of events for activating a servant as shown in Figure 39.2. Let us assume that we have configured the evictor with a size of five, that the queue is full, and that a request has arrived for a servant that is not currently active. (With a transactional evictor, we also assume this request does not change any persistent state.)

1. A client invokes an operation.
2. The object adapter invokes on the evictor to locate the servant.
3. The evictor first checks its servant map and fails to find the servant, so it instantiates the servant and restores its persistent state from the database.
4. The evictor adds an item for the servant (servant 1) at the head of the queue.
5. The queue's length now exceeds the configured maximum, so the evictor removes servant 6 from the queue as soon as it is eligible for eviction. With a background save evictor, this occurs once there are no outstanding requests pending on servant 6, and once the servant's state has been safely stored in the database. With a transactional save, the servant is removed from the queue immediately.

6. The object adapter dispatches the request to the new servant.

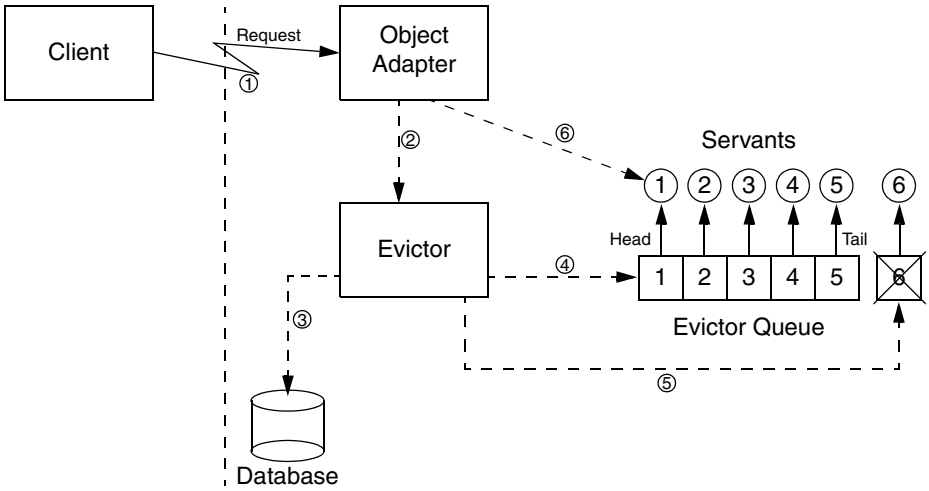


Figure 39.2. An evictor queue after restoring servant 1 and evicting servant 6.

39.3.5 Detecting Updates

A Freeze evictor considers that a servant's persistent state has been modified when a read-write operation on this servant completes. To indicate whether an operation is read-only or read-write, you add metadata directives to the Slice definitions of the objects:

- The `["freeze:write"]` directive informs the evictor that an operation modifies the persistent state of the target servant.
- The `["freeze:read"]` directive informs the evictor that an operation does not modify the persistent state of the target.

If no metadata directive is present, an operation is assumed to *not* modify its target.

Here is how you could mark the operations on an interface with these metadata directives:

```
interface Example {
    ["freeze:read"]  string readonlyOp();
    ["freeze:write"] void   writeOp();
};
```

This marks `readonlyOp` as an operation that does not modify its target, and marks `writeOp` as an operation that does modify its target. Because, without any directive, an operation is assumed to *not* modify its target, the preceding definition can also be written as follows:

```
interface Example {
    string readonlyOp(); // ["freeze:read"] implied
    ["freeze:write"] void writeOp();
};
```

The metadata directives can also be applied to an interface or a class to establish a default. This allows you to mark an interface as `["freeze:write"]` and to only add a `["freeze:read"]` directive to those operations that are read-only, for example:

```
["freeze:write"]
interface Example {
    ["freeze:read"] string readonlyOp();
                    void   writeOp1();
                    void   writeOp2();
                    void   writeOp3();
};
```

This marks `writeOp1`, `writeOp2`, and `writeOp3` as read-write operations, and `readonlyOp` as a read-only operation.

Note that it is important to correctly mark read-write operations with a `["freeze:write"]` metadata directive—without the directive, Freeze will not know when an object has been modified and may not store the updated persistent state to disk.

Also note that, if you make calls directly on servants (so the calls are not dispatched via the Freeze evictor), the evictor will have no idea when a servant's persistent state is modified; if any such direct call modifies the servant's data members, the update may be lost.

39.3.6 Evictor Iterator

A Freeze evictor iterator provides the ability to iterate over the identities of the objects stored in an evictor. The operations are similar to Java iterator methods:

`hasNext` returns true while there are more elements, and `next` returns the next identity:

```
local interface EvictorIterator {
    bool hasNext();
    Ice::Identity next();
};
```

You create an iterator by calling `getIterator` on your evictor:

```
EvictorIterator getIterator(string facet, int batchSize);
```

The new iterator is specific to a facet (specified by the `facet` parameter). Internally, this iterator will retrieve identities in batches of `batchSize` objects; we recommend to use a fairly large batch size to get good performance.

39.3.7 Indexing a Database

A Freeze evictor supports the use of indexes to quickly find persistent servants using the value of a data member as the search criteria. The types allowed for these indexes are the same as those allowed for Slice dictionary keys (see Section 4.9.4).

The **`slice2freeze`** and **`slice2freezej`** tools can generate an `Index` class when passed the **`--index`** option:

- **`--index`** ***`CLASS,TYPE,MEMBER`***
  `[,case-sensitive|case-insensitive]`

  `CLASS` is the name of the class to be generated. ***`TYPE`*** denotes the type of class to be indexed (objects of different classes are not included in this index). ***`MEMBER`*** is the name of the data member in ***`TYPE`*** to index. When ***`MEMBER`*** has type `string`, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

The generated `Index` class supplies three methods whose definitions are mapped from the following Slice operations:

- `sequence<Ice::Identity>`
  `findFirst(`*`member-type`* `index, int firstN)`

  Returns up to `firstN` objects of ***`TYPE`*** whose ***`MEMBER`*** is equal to `index`. This is useful to avoid running out of memory if the potential number of objects matching the criteria can be very large.

- `sequence<Ice::Identity> find(`*`member-type`* `index)`

  Returns all the objects of ***TYPE*** whose ***MEMBER*** is equal to `index`.

- `int count(<type> index)`

  Returns the number of objects of ***TYPE*** having ***MEMBER*** equal to `index`.

Indexes are associated with a Freeze evictor during evictor creation. See the definition of the `createBackgroundSaveEvictor` and `createTransactionalEvictor` functions for details.

Indexed searches are easy to use and very efficient. However, be aware that an index adds significant write overhead: with Berkeley DB, every update triggers a read from the database to get the old index entry and, if necessary, replace it.

If you add an index to an existing database, by default existing facets are not indexed. If you need to populate a new or empty index using the facets stored in your Freeze evictor, set the property `Freeze.Evictor.`*`env-name`*`.`*`file-name`*`.PopulateEmptyIndices` to a value other than 0, which instructs Freeze to iterate over the corresponding facets and create the missing index entries during the call to `createBackgroundSaveEvictor` or `createTransactionalEvictor`. When you use this feature, you must register the object factories for all of the facet types before you create your evictor.

39.3.8 Using a Servant Initializer

In some applications, it may be necessary to initialize a servant after the servant is instantiated by the evictor but before an operation is dispatched to the servant. The Freeze evictor allows an application to specify a servant initializer for this purpose.

To illustrate the sequence of events, let us assume that a request has arrived for a servant that is not currently active:

1. The evictor restores a servant for the target Ice object (and facet) from the database. This involves two steps:
   1. The Ice run time locates and invokes the factory for the Ice facet's type, thereby obtaining a new instance with uninitialized data members.
   2. The data members are populated from the persistent state.
2. The evictor invokes the application's servant initializer (if any) for the servant.
3. If the evictor is a background-save evictor, it adds the servant to its cache.
4. The evictor dispatches the operation.

With a background-save evictor, the servant initializer is called before the object is inserted into the evictor's internal cache, and *without* holding any internal lock, but in such a way that when the servant initializer is called, the servant is guaranteed to be inserted in the evictor cache.

There is only one restriction on what a servant initializer can do: it must not make a remote invocation on the object (facet) being initialized. Failing to follow this rule will result in deadlocks.

The file system implementation presented in Section 39.4 on page 1521 demonstrates the use of a servant initializer.

39.3.9 Background-Save Evictor Features

Creation

You create a background-save evictor in C++ with the global function `Freeze::createBackgroundSaveEvictor`, and in Java with the static method `Freeze.Util.createBackgroundSaveEvictor`.

For C++, the signatures are as follows:

```
BackgroundSaveEvictorPtr
createBackgroundSaveEvictor(
    const ObjectAdapterPtr& adapter,
    const string& envName,
    const string& filename,
    const ServantInitializerPtr& initializer = 0,
    const vector<IndexPtr>& indexes = vector<IndexPtr>(),
    bool createDb = true);

BackgroundSaveEvictorPtr
    createBackgroundSaveEvictor(
    const ObjectAdapterPtr& adapter,
    const string& envName,
    DbEnv& dbEnv,
    const string& filename,
    const ServantInitializerPtr& initializer = 0,
    const vector<IndexPtr>& indexes = vector<IndexPtr>(),
    bool createDb = true);
```

For Java, the method signatures are:

```
public static BackgroundSaveEvictor
createBackgroundSaveEvictor(
    Ice.ObjectAdapter adapter,
    String envName,
```

```
    String filename,
    ServantInitializer initializer,
    Index[] indexes,
    boolean createDb);

public static BackgroundSaveEvictor
createBackgroundSaveEvictor(
    Ice.ObjectAdapter adapter,
    String envName,
    com.sleepycat.db.Environment dbEnv,
    String filename,
    ServantInitializer initializer,
    Index[] indexes,
    boolean createDb);
```

Both C++ and Java provide two overloaded functions: in one case, Freeze opens and manages the underlying Berkeley DB environment; in the other case, you provide a `DbEnv` object that represents a Berkeley DB environment you opened yourself. (Usually, it is easiest to let Freeze take care of all interactions with Berkeley DB.)

The `envName` parameter represents the name of the underlying Berkeley DB environment, and is also used as the default Berkeley DB home directory. (See `Freeze.Evictor.env-name.DbHome` in the Ice properties reference.)

The `filename` parameter represents the name of the Berkeley DB database file associated with this evictor. The persistent state of all your servants is stored in this file.

The `initializer` parameter represents the servant initializer. It is an optional parameter in C++; in Java, pass `null` if you do not need a servant initializer.

The `indexes` parameter is a vector or array of evictor indexes. It is an optional parameter in C++; in Java, pass `null` if your evictor does not define an index.

Finally, the `createDb` parameter tells Freeze what to do when the corresponding Berkeley DB database does not exist. When true, Freeze creates a new database; when false, Freeze raises a `Freeze::DatabaseException`.

Saving Thread

All persistence activity of a background-save evictor is handled in a background thread created by the evictor. This thread wakes up periodically and saves the state of all newly-registered, modified, and destroyed servants in the evictor's queue.

For applications that experience bursts of activity that result in a large number of modified servants in a short period of time, you can also configure the evictor's thread to begin saving as soon as the number of modified servants reaches a certain threshold.

Synchronization

When the saving thread takes a snapshot of a servant it is about to save, it is necessary to prevent the application from modifying the servant's persistent data members at the same time.

The Freeze evictor and the application need to use a common synchronization to ensure correct behavior. In Java, this common synchronization is the servant itself: the Freeze evictor synchronizes the servant (a Java object) while taking the snapshot. In C++, the servant is required to inherit from the class `IceUtil::AbstractMutex`: the background-save evictor locks the servant through this interface while taking a snapshot. On the application side, the servant's implementation is required to use the same mechanism to synchronize all operations that access the servant's Slice-defined data members.

Keeping Servants in Memory

Occasionally, automatically evicting and reloading all servants can be inefficient. You can remove a servant from the evictor's queue by locking this servant "in memory" using the `keep` or `keepFacet` operation on the evictor:

```
local interface BackgroundSaveEvictor extends Evictor {
    void keep(Ice::Identity id);
    void keepFacet(Ice::Identity id, string facet);
    void release(Ice::Identity id);
    void releaseFacet(Ice::Identity id, string facet);
};
```

`keep` and `keepFacet` are recursive: you need to call `release` or `releaseFacet` for this servant the same number of times to put it back in the evictor queue and make it eligible again for eviction.

Servants kept in memory (using `keep` or `keepFacet`) do not consume a slot in the evictor queue. As a result, the maximum number of servants in memory is approximately the number of kept servants plus the evictor size. (It can be larger while you have many evictable objects that are modified but not yet saved.)

39.3.10 Transactional Evictor Features

Creation

You create a transactional evictor in C++ with the global function `Freeze::createTransactionalEvictor`, and in Java with the static method `Freeze.Util.createTransactionalEvictor`.

For C++, the signatures are as follows:

```
typedef map<string, string> FacetTypeMap;

TransactionalEvictorPtr
createTransactionalEvictor(
    const ObjectAdapterPtr& adapter,
    const string& envName,
    const string& filename,
    const FacetTypeMap& facetTypes = FacetTypeMap(),
    const ServantInitializerPtr& initializer = 0,
    const vector<IndexPtr>& indexes = vector<IndexPtr>(),
    bool createDb = true);

TransactionalEvictorPtr
createTransactionalEvictor(
    const ObjectAdapterPtr& adapter,
    const string& envName,
    DbEnv& dbEnv,
    const string& filename,
    const FacetTypeMap& facetTypes = FacetTypeMap(),
    const ServantInitializerPtr& initializer = 0,
    const vector<IndexPtr>& indexes = vector<IndexPtr>(),
    bool createDb = true);
```

For Java, the method signatures are:

```
public static TransactionalEvictor
createTransactionalEvictor(
    Ice.ObjectAdapter adapter,
    String envName,
    String filename,
    java.util.Map facetTypes,
    ServantInitializer initializer,
    Index[] indexes,
    boolean createDb);

public static TransactionalEvictor
createTransactionalEvictor(
```

```
    Ice.ObjectAdapter adapter,
    String envName,
    com.sleepycat.db.Environment dbEnv,
    String filename,
    java.util.Map facetTypes,
    ServantInitializer initializer,
    Index[] indexes,
    boolean createDb);
```

Both C++ and Java provide two overloaded functions: in one case, Freeze opens and manages the underlying Berkeley DB environment; in the other case, you provide a `DbEnv` object that represents a Berkeley DB environment you opened yourself. (Usually, it is easier to let Freeze take care of all interactions with Berkeley DB.)

The `envName` parameter represents the name of the underlying Berkeley DB environment, and is also used as the default Berkeley DB home directory. (See `Freeze.Evictor.`*`env-name`*`.DbHome` in the Ice properties reference.)

The `filename` parameter represents the name of the Berkeley DB database file associated with this evictor. The persistent state of all your servants is stored in this file.

The `facetTypes` parameter allows you to specify a single class type (Ice type ID string) for each facet in your new evictor (see below). Most applications use only the default facet, represented by an empty string. This parameter is optional in C++; in Java, pass `null` if you do not want to specify such a facet-to-type mapping.

The `initializer` parameter represents the servant initializer. It is an optional parameter in C++; in Java, pass `null` if you do not need a servant initializer.

The `indexes` parameter is a vector or array of evictor indexes. It is an optional parameter in C++; in Java, pass `null` if your evictor does not define an index.

Finally, the `createDb` parameter tells Freeze what to do when the corresponding Berkeley DB database does not exist. When true, Freeze creates a new database; when false, Freeze raises a `Freeze::DatabaseException`.

Homogeneous Databases

When a transactional evictor processes an incoming request without an associated transaction, it first needs to find out whether the corresponding operation is read-only or read-write (as specified by the `"freeze:read"` and `"freeze:write"` operation metadata). This is straightforward if the evictor knows the target's type;

in this case, it simply instantiates and keeps a "dummy" servant to look up the attributes of each operation.

However, if the target type can vary, the evictor needs to look up and sometimes load a read-only servant to find this information. For read-write requests, it will then load the servant from disk a second time (within a new transaction). Once the transaction commits, the read-only servant—sometimes freshly loaded from disk—is discarded.

When you create a transactional evictor with `createTransactionalEvictor`, you can pass a facet name to type ID map to associate a single servant type with each facet and speed up the operation attribute lookups.

Synchronization

With a transactional evictor, there is no need to perform any synchronization on the servants managed by the evictor:

- For read-only operations, the application must not modify any data member, and hence there is no need to synchronize. (Many threads can safely read the same data members concurrently.)
- For read-write operations, each operation dispatch gets its own private servant or servants (see transaction propagation below).

Not having to worry about synchronization can dramatically simplify your application code.

Transaction Propagation

Without a distributed transaction service, it is not possible to invoke several remote operations within the same transaction. Nevertheless, Freeze supports transaction propagation for collocated calls: when a request is dispatched within a transaction, the transaction is associated with the dispatch thread and will propagate to any other servant reached through a collocated call. If the target of a collocated call is managed by a transactional evictor associated with the same database environment, Freeze reuses the propagated transaction to load the servant and dispatch the request. This allows you to group updates to several servants within a single transaction.

You can also control how a transactional evictor handles an incoming transaction through optional metadata added after `"freeze:write"` and `"freeze:read"`. There are six valid directives:

- `"freeze:read:never"`

  Verify that no transaction is propagated to this operation. If a transaction is present, the transactional evictor raises a `Freeze::DatabaseException`.
- `"freeze:read:supports"`

  Accept requests with or without a transaction, and re-use the transaction if present. `"supports"` is the default for `"freeze:read"` operations.
- `"freeze:read:mandatory"` and `"freeze:write:mandatory"`

  Verify that a transaction is propagated to this operation. If there is no transaction, the transactional evictor raises a `Freeze::DatabaseException`.
- `"freeze:read:required"` and `"freeze:write:required"`

  Accept requests with or without a transaction, and re-use the transaction if present. If no transaction is propagated, the transactional evictor creates a new transaction before dispatching the request. `"required"` is the default for `"freeze:write"` operations.

Commit or Rollback on User Exception

When a transactional evictor processes an incoming read-write request, it starts a new database transaction, loads a servant within the transaction, dispatches the request, and then either commits or rolls back the transaction depending on the outcome of this dispatch. If the dispatch does not raise an exception, the transaction is committed just before the response is sent back to the client. If the dispatch raises a system exception, the transaction is rolled back. If the dispatch raises a user exception, by default, the transaction is committed. However, you can configure Freeze to rollback on user-exceptions by setting `Freeze.Evictor.`*`env-name`*`.`*`fileName`*`.RollbackOnUserException` to a value other than 0.

Deadlocks and Automatic Retries

When reading and writing in separate concurrent transactions, deadlocks are likely to occur. For example, one transaction may lock pages in a particular order while another transaction locks the same pages in a different order; the outcome is a deadlock. Berkeley DB automatically detects such deadlocks, and “kills” one of the transactions.

With a Freeze transactional evictor, the application does not need to catch any deadlock exceptions or retry when deadlock occurs because the transactional evictor automatically retries its transactions whenever it encounters a deadlock situation.

However, this can affect how you implement your operations: for any operation called within a transaction (mainly read-write operations), you must anticipate the possibility of several calls for the same request, all in the same dispatch thread.

Asynchronous Method Dispatch

When a transactional evictor dispatches a read-write operation implemented using AMD, it starts a transaction before dispatching the request, and commits or rolls back the transaction when the dispatch is done. Two threads are involved here: the *dispatch thread* and the *callback thread*. The dispatch thread is a thread from an Ice thread pool tasked with dispatching a request, and the callback thread is the thread that invokes the AMD callback to send the response to the client. These threads may be one and the same if the servant invokes the AMD callback from the dispatch thread.

It is important to understand the threading semantics of an AMD request with respect to the transaction:

- If a successful AMD response is sent from the dispatch thread, the transaction is committed *after* the response is sent. If a deadlock occurs during this commit, the request is not retried and the client receives no indication of the failure.
- If a successful AMD response is sent from another thread, the evictor commits its transaction when the dispatch thread completes, regardless of whether the servant has sent the AMD response. The callback thread waits until the transaction has been committed by the dispatch thread before sending the response.
- If a commit results in a deadlock and the AMD response has not yet been sent, the evictor cancels the original AMD callback and automatically retries the request again with a new AMD callback. Invocations on the original AMD callback are ignored (`ice_response` and `ice_exception` on this callback do nothing).
- Otherwise, if the servant sends an exception via the AMD callback, the response is sent directly to the client.

Transactions and Freeze Maps

A transactional evictor uses the same transaction objects as Freeze maps, which allows you to update a Freeze map within a transaction managed by a transactional evictor.

You can get the current transaction created by a transactional evictor by calling `getCurrentTransaction`. Then, you would typically retrieve the associated Freeze connection (with `getConnection`) and construct a Freeze map using this connection:

```
local interface TransactionalEvictor extends Evictor {
    Transaction getCurrentTransaction();
    void setCurrentTransaction(Transaction tx);
};
```

A transactional evictor also gives you the ability to associate your own transaction with the current thread, using `setCurrentTransaction`. This is useful if you want to perform many updates within a single transaction, for example to add or remove many servants in the evictor. (A less convenient alternative is to implement all such updates within a read-write operation on some object.)

39.3.11 Application Design Considerations

The Freeze evictor creates a snapshot of a servant's state for persistent storage by marshaling the servant, just as if the servant were being sent "over the wire" as a parameter to a remote invocation. Therefore, the Slice definitions for an object type must include the data members comprising the object's persistent state.

For example, we could define a Slice class as follows:

```
class Stateless {
    void calc();
};
```

However, without data members, there will not be any persistent state in the database for objects of this type, and hence there is little value in using the Freeze evictor for this type.

Obviously, Slice object types need to define data members, but there are other design considerations as well. For example, suppose we define a simple application as follows:

```
class Account {
    ["freeze:write"] void withdraw(int amount);
    ["freeze:write"] void deposit(int amount);

    int balance;
};
```

```
interface Bank {
    Account* createAccount();
};
```

In this application, we would use a Freeze evictor to manage `Account` objects that have a data member `balance` representing the persistent state of an account.

From an object-oriented design perspective, there is a glaring problem with these Slice definitions: implementation details (the persistent state) are exposed in the client–server contract. The client cannot directly manipulate the `balance` member because the `Bank` interface returns `Account` proxies, not `Account` instances. However, the presence of the data member may cause unnecessary confusion for client developers.

A better alternative is to clearly separate the persistent state as shown below:

```
interface Account {
    ["freeze:write"] void withdraw(int amount);
    ["freeze:write"] void deposit(int amount);
};

interface Bank {
    Account* createAccount();
};

class PersistentAccount implements Account {
    int balance;
};
```

Now the Freeze evictor can manage `PersistentAccount` objects, while clients interact with `Account` proxies. (Ideally, `PersistentAccount` would be defined in a different source file and inside a separate Slice module.)

39.4 Using the Freeze Evictor in a File System Server

In this section, we present file system implementations that use a transactional evictor. The implementations are based on the ones discussed in Chapter 34, and in this section we only discuss code that illustrates use of the Freeze evictor.

In general, incorporating a Freeze evictor into your application requires the following steps:

1. Evaluate your existing Slice definitions for a suitable persistent object type.

2. If no suitable type is found, you typically define a new derived class that captures your persistent state requirements. Consider placing these definitions in a separate file: they are only used by the server for persistence, and therefore do not need to appear in the "public" definitions required by clients. Also consider placing your persistent types in a separate module to avoid name clashes.
3. Generate code (using **`slice2freeze`** or **`slice2freezej`**) for your new definitions.
4. Create an evictor and register it as a servant locator with an object adapter.
5. Create instances of your persistent type and register them with the evictor.

39.4.1 Slice Definitions

Fortunately, it is unnecessary for us to change any of the existing file system Slice definitions to incorporate the Freeze evictor. However, we do need to add metadata definitions to inform the evictor which operations modify object state (see Section 39.3.5):

```
module Filesystem {
    // ...

    interface Node {
        idempotent string name();

        ["freeze:write"]
        void destroy() throws PermissionDenied;
    };

    // ...

    interface File extends Node {
        idempotent Lines read();

        ["freeze:write"]
        idempotent void write(Lines text) throws GenericError;
    };

    // ...

    interface Directory extends Node {
        idempotent NodeDescSeq list();
```

```
        idempotent NodeDesc find(string name) throws NoSuchName;

        ["freeze:write"]
        File* createFile(string name) throws NameInUse;

        ["freeze:write"]
        Directory* createDirectory(string name) throws NameInUse;
    };
};
```

This definitions are identical to the original ones, with the exception of the added `["freeze:write"]` directives.

The remaining definitions are in derived classes:

```
#include <Filesystem.ice>

module Filesystem {
    class PersistentDirectory;

    class PersistentNode implements Node {
        string nodeName;
        PersistentDirectory* parent;
    };

    class PersistentFile
            extends PersistentNode implements File {
        Lines text;
    };

    dictionary<string, NodeDesc> NodeDict;

    class PersistentDirectory
            extends PersistentNode implements Directory {
        ["freeze:write"]
        void removeNode(string name);

        NodeDict nodes;
    };
};
```

As you can see, we have sub-classed all of the node interfaces. Let us examine each one in turn.

The `PersistentNode` class adds two data members: `nodeName`[1] and `parent`. The file system implementation requires that a child node knows its parent node in order to properly implement the `destroy` operation. Previous implementations

had a state member of type `DirectoryI`, but that is not workable here. It is no longer possible to pass the parent node to the child node's constructor because the evictor may be instantiating the child node (via a factory), and the parent node will not be known. Even if it were known, another factor to consider is that there is no guarantee that the parent node will be active when the child invokes on it, because the evictor may have evicted it. We solve these issues by storing a proxy to the parent node. If the child node invokes on the parent node via the proxy, the evictor automatically activates the parent node if necessary.

The `PersistentFile` class is very straightforward, simply adding a `text` member representing the contents of the file. Notice that the class extends `PersistentNode`, and therefore inherits the state members declared by the base class.

Finally, the `PersistentDirectory` class defines the `removeNode` operation, and adds the `nodes` state member representing the immediate children of the directory node. Since a child node contains only a proxy for its `PersistentDirectory` parent, and not a reference to an implementation class, there must be a Slice-defined operation that can be invoked when the child is destroyed.

If we had followed the advice at the beginning of Section 39.3, we would have defined `Node`, `File`, and `Directory` classes in a separate `PersistentFilesystem` module, but in this example we use the existing `Filesystem` module for the sake of simplicity.

39.4.2 Implementing the File System Server in C++

The Server `main` Program

The server's `main` program is responsible for creating the evictor and initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the class `Ice::Application` as described in Section 8.3.1. Our server `main` program has now become the following:

```
#include <PersistentFilesystemI.h>

using namespace std;
using namespace Filesystem;
```

1. We used `nodeName` instead of `name` because `name` is already used as an operation in the `Node` interface.

```
class FilesystemApp : virtual public Ice::Application
{
public:

    FilesystemApp(const string& envName) :
        _envName(envName)
    {
    }

    virtual int run(int, char*[])
    {
        Ice::ObjectFactoryPtr factory = new NodeFactory;
        communicator()->addObjectFactory(
            factory, PersistentFile::ice_staticId());
        communicator()->addObjectFactory(
            factory, PersistentDirectory::ice_staticId());

        Ice::ObjectAdapterPtr adapter = communicator()->
                    createObjectAdapter("EvictorFilesystem");

        Freeze::EvictorPtr evictor =
            Freeze::createTransactionalEvictor(
                adapter, _envName, "evictorfs");
        FileI::_evictor = evictor;
        DirectoryI::_evictor = evictor;

        adapter->addServantLocator(evictor, "");

        Ice::Identity rootId;
        rootId.name = "RootDir";
        if(!evictor->hasObject(rootId))
        {
            PersistentDirectoryPtr root = new DirectoryI;
            root->nodeName = "/";
            evictor->add(root, rootId);
        }

        adapter->activate();

        communicator()->waitForShutdown();
        if(interrupted())
        {
            cerr << appName()
                 << ": received signal, shutting down" << endl;
        }
```

```
        return 0;
    }

private:

    string _envName;
};

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv, "config.server");
}
```

Let us examine the changes in detail. First, we are now including `PersistentFilesystemI.h`. This header file includes all of the other Freeze (and Ice) header files this source file requires.

Next, we define the class `FilesystemApp` as a subclass of `Ice::Application`, and provide a constructor taking a string argument:

```
    FilesystemApp(const string& envName) :
        _envName(envName) { }
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

One of the first tasks `run` performs is installing the Ice object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. A single instance of `NodeFactory` is installed for both types.

```
        Ice::ObjectFactoryPtr factory = new NodeFactory;
        communicator()->addObjectFactory(
            factory,
            PersistentFile::ice_staticId());
        communicator()->addObjectFactory(
            factory,
            PersistentDirectory::ice_staticId());
```

After creating the object adapter, the program initializes a Freeze evictor by invoking `createTransactionalEvictor`. The third argument to `createTransactionalEvictor` is the name of the database file, which by

default is created if it does not exist. The new evictor is then added to the object adapter as a servant locator for the default category.

```
NodeI::_evictor = Freeze::createTransactionalEvictor(
                    adapter, _envName, "evictorfs");
adapter->addServantLocator(NodeI::_evictor, "");
```

Next, the program creates the root directory node if it is not already being managed by the evictor.

```
Ice::Identity rootId;
rootId.name = "RootDir";
if(!evictor->hasObject(rootId))
{
    PersistentDirectoryPtr root = new DirectoryI;
    root->nodeName = "/";
    evictor->add(root, rootId);
}
```

Finally, the `main` function instantiates the `FilesystemApp`, passing `db` as the name of the database environment.

```
int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv, "config.server");
}
```

The Servant Class Definitions

The servant classes must also be changed to incorporate the Freeze evictor. We no longer derive the servants from a common base class. Instead, `FileI` and `DirectoryI` each have their own `_destroyed` and `_mutex` members, as well as a static `_evictor` smart pointer that points at the transactional evictor.

```
#include <PersistentFilesystem.h>
#include <IceUtil/IceUtil.h>
#include <Freeze/Freeze.h>

namespace Filesystem {

class FileI : virtual public PersistentFile {
public:

    FileI();
```

```
    // Slice operations...

    static Freeze::EvictorPtr _evictor;

private:

    bool _destroyed;
    IceUtil::Mutex _mutex;
};

class DirectoryI : virtual public PersistentDirectory {
public:

    DirectoryI();

    // Slice operations...

    virtual void removeNode(const std::string&,
                            const Ice::Current&);

    static Freeze::EvictorPtr _evictor;

public:
    bool _destroyed;
    IceUtil::Mutex _mutex;
};
```

In addition to the node implementation classes, we have also declared an object factory:

```
namespace Filesystem {
    class NodeFactory : virtual public Ice::ObjectFactory {
    public:
        virtual Ice::ObjectPtr create(const std::string&);
        virtual void destroy();
    };
```

Implementing `FileI`

The `FileI` methods are mostly trivial, because the Freeze evictor handles persistence for us.

```
Filesystem::FileI::FileI() : _destroyed(false)
{
}

string
```

```
Filesystem::FileI::name(const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(
            __FILE__, __LINE__, c.id, c.facet, c.operation);
    }

    return nodeName;
}

void
Filesystem::FileI::destroy(const Ice::Current& c)
{
    {
        IceUtil::Mutex::Lock lock(_mutex);

        if (_destroyed) {
            throw Ice::ObjectNotExistException(
                __FILE__, __LINE__, c.id, c.facet, c.operation);
        }
        _destroyed = true;
    }

    //
    // Because we use a transactional evictor,
    // these updates are guaranteed to be atomic.
    //
    parent->removeNode(nodeName);
    _evictor->remove(c.id);
}

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(
            __FILE__, __LINE__, c.id, c.facet, c.operation);
    }

    return text;
}
```

```
void
Filesystem::FileI::write(const Filesystem::Lines& text,
                         const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(
            __FILE__, __LINE__, c.id, c.facet, c.operation);
    }

    this->text = text;
}
```

The code checks that the node has not been destroyed before acting on the invocation by updating or returning state. Note that `destroy` must update two separate nodes: as well as removing itself from the evictor, the node must also update the parent's node map. Because we are using a transactional evictor, the two updates are guaranteed to be atomic, so it is impossible to the leave the file system in an inconsistent state.

Implementing `DirectoryI`

The `DirectoryI` implementation requires more substantial changes. We begin our discussion with the `createDirectory` operation.

```
Filesystem::DirectoryPrx
Filesystem::DirectoryI::createDirectory(const string& name,
                                        const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(
            __FILE__, __LINE__, c.id, c.facet, c.operation);
    }

    if (name.empty() || nodes.find(name) != nodes.end()) {
        throw NameInUse(name);
    }

    Ice::Identity id;
    id.name = IceUtil::generateUUID();
    PersistentDirectoryPtr dir = new DirectoryI;
    dir->nodeName = name;
    dir->parent = PersistentDirectoryPrx::uncheckedCast(
```

```
                                c.adapter->createProxy(c.id));
    DirectoryPrx proxy = DirectoryPrx::uncheckedCast(
                                _evictor->add(dir, id));

    NodeDesc nd;
    nd.name = name;
    nd.type = DirType;
    nd.proxy = proxy;
    nodes[name] = nd;

    return proxy;
}
```

After validating the node name, the operation obtains a unique identity for the child directory, instantiates the servant, and registers it with the Freeze evictor. Finally, the operation creates a proxy for the child and adds the child to its node table.

The implementation of the `createFile` operation has the same structure as `createDirectory`.

```
Filesystem::FilePrx
Filesystem::DirectoryI::createFile(const string& name,
                                   const Ice::Current& c)
{
    IceUtil::Mutex::Lock lock(_mutex);

    if (_destroyed) {
        throw Ice::ObjectNotExistException(
            __FILE__, __LINE__, c.id, c.facet, c.operation);
    }

    if (name.empty() || nodes.find(name) != nodes.end()) {
        throw NameInUse(name);
    }

    Ice::Identity id;
    id.name = IceUtil::generateUUID();
    PersistentFilePtr file = new FileI;
    file->nodeName = name;
    file->parent = PersistentDirectoryPrx::uncheckedCast(
                                c.adapter->createProxy(c.id));
    FilePrx proxy = FilePrx::uncheckedCast(
                                _evictor->add(file, id));

    NodeDesc nd;
```

```
    nd.name = name;
    nd.type = FileType;
    nd.proxy = proxy;
    nodes[name] = nd;

    return proxy;
}
```

Implementing `NodeFactory`

We use a single factory implementation for creating two types of Ice objects: `PersistentFile` and `PersistentDirectory`. These are the only two types that the Freeze evictor will be restoring from its database.

```
Ice::ObjectPtr
Filesystem::NodeFactory::create(const string& type)
{
    if (type == PersistentFile::ice_staticId())
        return new FileI;
    else if (type == PersistentDirectory::ice_staticId())
        return new DirectoryI;
    else {
        assert(false);
        return 0;
    }
}

void
Filesystem::NodeFactory::destroy()
{
}
```

The remaining Slice operations have trivial implementations, so we do not show them here.

39.4.3 Implementing the File System Server in Java

The Server `main` Method

The server's `main` method is responsible for creating the evictor and initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the class `Ice.Application` as described in Section 12.3.1. Our server `main` program has now become the following:

```
import Filesystem.*;

public class Server extends Ice.Application
{
    public
    Server(String envName)
    {
        _envName = envName;
    }

    public int
    run(String[] args)
    {
        Ice.ObjectFactory factory = new NodeFactory();
        communicator().addObjectFactory(
            factory, PersistentFile.ice_staticId());
        communicator().addObjectFactory(
            factory, PersistentDirectory.ice_staticId());

        Ice.ObjectAdapter adapter =
            communicator().createObjectAdapter(
                "EvictorFilesystem");

        Freeze.Evictor evictor =
            Freeze.Util.createTransactionalEvictor(
                adapter, _envName, "evictorfs",
                    null, null, null, true);
        DirectoryI._evictor = evictor;
        FileI._evictor = evictor;

        adapter.addServantLocator(evictor, "");

        Ice.Identity rootId = new Ice.Identity();
        rootId.name = "RootDir";
        if(!evictor.hasObject(rootId))
        {
            PersistentDirectory root = new DirectoryI();
            root.nodeName = "/";
            root.nodes =
                new java.util.HashMap<
                    java.lang.String, NodeDesc>();
            evictor.add(root, rootId);
        }

        adapter.activate();
```

```
        communicator().waitForShutdown();

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server("db");
        int status = app.main("Server", args, "config.server");
        System.exit(status);
    }

    private String _envName;
}
```

Let us examine the changes in detail. First, we define the class `Server` as a subclass of `Ice.Application`, and provide a constructor taking a string argument:

```
    public
    Server(String envName)
    {
        _envName = envName;
    }
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

One of the first tasks `run` performs is installing the Ice object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. A single instance of `NodeFactory` is installed for both types.

```
        Ice.ObjectFactory factory = new NodeFactory();
        communicator().addObjectFactory(
            factory, PersistentFile.ice_staticId());
        communicator().addObjectFactory(
            factory, PersistentDirectory.ice_staticId());
```

After creating the object adapter, the program initializes a transactional evictor by invoking `createTransactionalEvictor`. The third argument to `createTransactionalEvictor` is the name of the database, the fourth is null to indicate that we do not use facets, the fifth is null to indicate that we do not use a servant initializer, the sixth argument (`null`) indicates no indexes are in use, and

the `true` argument requests that the database be created if it does not exist. The evictor is then added to the object adapter as a servant locator for the default category.

```
Freeze.Evictor evictor =
    Freeze.Util.createTransactionalEvictor(
        adapter, _envName, "evictorfs",
            null, null, null, true);
DirectoryI._evictor = evictor;
FileI._evictor = evictor;

adapter.addServantLocator(evictor, "");
```

Next, the program creates the root directory node if it is not already being managed by the evictor.

```
Ice.Identity rootId = new Ice.Identity();
rootId.name = "RootDir";
if(!evictor.hasObject(rootId))
{
    PersistentDirectory root = new DirectoryI();
    root.nodeName = "/";
    root.nodes =
        new java.util.HashMap<
            java.lang.String, NodeDesc>();
    evictor.add(root, rootId);
}
```

Finally, the `main` function instantiates the `Server` class, passing `db` as the name of the database environment.

```
public static void
main(String[] args)
{
    Server app = new Server("db");
    int status = app.main("Server", args, "config.server");
    System.exit(status);
}
```

The Servant Class Definitions

The servant classes must also be changed to incorporate the Freeze evictor. The `FileI` class now has a static state member `_evictor`.

```
import Filesystem.*;

public final class FileI extends PersistentFile
{
    public
    FileI()
    {
        _destroyed = false;
    }

    // Slice operations...

    public static Freeze.Evictor _evictor;
    private boolean _destroyed;
}
```

The `DirectoryI` class has undergone a similar transformation.

```
import Filesystem.*;

public final class DirectoryI extends PersistentDirectory
{
    public
    DirectoryI()
    {
        _destroyed = false;
        nodes = new java.util.HashMap
                    <java.lang.String, NodeDesc>();
    }

    // Slice operations...

    public static Freeze.Evictor _evictor;
    private boolean _destroyed;
}
```

Implementing `FileI`

The `FileI` methods are mostly trivial, because the Freeze evictor handles persistence for us.

```
    public synchronized String
    name(Ice.Current current)
    {
        if (_destroyed) {
            throw new Ice.ObjectNotExistException(
                current.id, current.facet, current.operation);
```

```
        }

        return nodeName;
    }

    public void
    destroy(Ice.Current current)
        throws PermissionDenied
    {
        synchronized(this) {
            if (_destroyed) {
                throw new Ice.ObjectNotExistException(
                    current.id, current.facet, current.operation);
            }
            _destroyed = true;
        }

        //
        // Because we use a transactional evictor,
        // these updates are guaranteed to be atomic.
        //
        parent.removeNode(nodeName);
        _evictor.remove(current.id);
    }

    public synchronized String[]
    read(Ice.Current current)
    {
        if (_destroyed) {
            throw new Ice.ObjectNotExistException(
                current.id, current.facet, current.operation);
        }

        return (String[])text.clone();
    }

    public synchronized void
    write(String[] text, Ice.Current current)
        throws GenericError
    {
        if (_destroyed) {
            throw new Ice.ObjectNotExistException(
                current.id, current.facet, current.operation);
```

```
        }

        this.text = text;
    }
```

The code checks that the node has not been destroyed before acting on the invocation by updating or returning state. Note that `destroy` must update two separate nodes: as well as removing itself from the evictor, the node must also update the parent's node map. Because we are using a transactional evictor, the two updates are guaranteed to be atomic, so it is impossible to the leave the file system in an inconsistent state.

Implementing `DirectoryI`

The `DirectoryI` implementation requires more substantial changes. We begin our discussion with the `createDirectory` operation.

```
    public synchronized DirectoryPrx
    createDirectory(String name, Ice.Current current)
        throws NameInUse
    {
        if (_destroyed) {
            throw new Ice.ObjectNotExistException(
                current.id, current.facet, current.operation);
        }

        if (name.length() == 0 || nodes.containsKey(name)) {
            throw new NameInUse(name);
        }

        Ice.Identity id =
            current.adapter.getCommunicator().
                stringToIdentity(
                    java.util.UUID.randomUUID().toString());
        PersistentDirectory dir = new DirectoryI();
        dir.nodeName = name;
        dir.parent = PersistentDirectoryPrxHelper.uncheckedCast(cu
rrent.adapter.createProxy(current.id));
        DirectoryPrx proxy = DirectoryPrxHelper.uncheckedCast(_evi
ctor.add(dir, id));

        NodeDesc nd = new NodeDesc();
        nd.name = name;
        nd.type = NodeType.DirType;
        nd.proxy = proxy;
```

```
        nodes.put(name, nd);

        return proxy;
    }
```

After validating the node name, the operation obtains a unique identity for the child directory, instantiates the servant, and registers it with the Freeze evictor. Finally, the operation creates a proxy for the child and adds the child to its node table.

The implementation of the `createFile` operation has the same structure as `createDirectory`.

```
    public synchronized FilePrx
    createFile(String name, Ice.Current current)
        throws NameInUse
    {
        if (_destroyed) {
            throw new Ice.ObjectNotExistException(
                current.id, current.facet, current.operation);
        }

        if (name.length() == 0 || nodes.containsKey(name)) {
            throw new NameInUse(name);
        }

        Ice.Identity id = current.adapter.getCommunicator().
            stringToIdentity(
                java.util.UUID.randomUUID().toString());
        PersistentFile file = new FileI();
        file.nodeName = name;
        file.parent = PersistentDirectoryPrxHelper.uncheckedCast(
            current.adapter.createProxy(current.id));
        FilePrx proxy =
            FilePrxHelper.uncheckedCast(_evictor.add(file, id));

        NodeDesc nd = new NodeDesc();
        nd.name = name;
        nd.type = NodeType.FileType;
        nd.proxy = proxy;
        nodes.put(name, nd);

        return proxy;
    }
```

The remaining Slice operations have trivial implementations, so we do not show them here.

Implementing `NodeFactory`

We use a single factory implementation for creating two types of Ice objects: `PersistentFile` and `PersistentDirectory`. These are the only two types that the Freeze evictor will be restoring from its database.

```
package Filesystem;

public class NodeFactory implements Ice.ObjectFactory
{
    public Ice.Object
    create(String type)
    {
        if (type.equals(PersistentFile.ice_staticId()))
            return new FileI();
        else if (type.equals(PersistentDirectory.ice_staticId()))
            return new DirectoryI();
        else {
            assert(false);
            return null;
        }
    }

    public void
    destroy()
    {
    }
}
```

39.5 The Freeze Map

A Freeze map is a persistent, associative container in which the key and value types can be any primitive or user-defined Slice types. For each pair of key and value types, the developer uses a code-generation tool to produce a language-specific class that conforms to the standard conventions for maps in that language. For example, in C++, the generated class resembles a `std::map`, and in Java it implements the `java.util.SortedMap` interface. Most of the logic for storing and retrieving state to and from the database is implemented in a Freeze base class. The generated map classes derive from this base class, so they contain little code and therefore are efficient in terms of code size.

You can only store data types that are defined in Slice in a Freeze map. Types without a Slice definition (that is, arbitrary C++ or Java types) cannot be stored

because a Freeze map reuses the Ice-generated marshaling code to create the persistent representation of the data in the database. This is especially important to remember when defining a Slice class whose instances will be stored in a Freeze map; only the "public" (Slice-defined) data members will be stored, not the private state members of any derived implementation class.

39.5.1 Freeze Connections

In order to create a Freeze map object, you first need to obtain a Freeze `Connection` object by connecting to a database environment.

As illustrated in Figure 39.3, a Freeze map is associated with a single connection and a single database file. Connection and map objects are not thread-safe: if you want to use a connection or any of its associated maps from multiple threads, you must serialize access to them. If your application requires concurrent access to the same database file (persistent map), you must create several connections and associated maps.

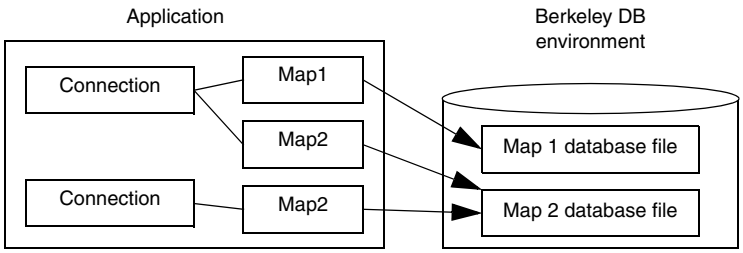


Figure 39.3. Freeze connections and maps.

Freeze connections provide operations that allow you to begin a transaction, access the current transaction, get the communicator associated with a connection, close a connection, and remove a map index. See the online Slice reference for more information on these operations.

39.5.2 Transactions

You may optionally use transactions with Freeze maps. Freeze transactions provide the usual ACID (atomicity, concurrency, isolation, durability) properties. For example, a transaction allows you to group several database updates in one atomic unit: either all or none of the updates within the transaction occur.

You start a transaction by calling `beginTransaction` on the `Connection` object. Once a connection has an associated transaction, all operations on the map objects associated with this connection use this transaction. Eventually, you end the transaction by calling `commit` or `rollback`: `commit` saves all your updates while `rollback` undoes them. The `currentTransaction` operation returns the transaction associated with a connection, if any; otherwise, it returns nil.

```
module Freeze {

local interface Transaction {
    void commit();
    void rollback();
};

local interface Connection {
    Transaction beginTransaction();
    idempotent Transaction currentTransaction();
    // ...
};
};
```

If you do not use transactions, every non-iterator update is enclosed in its own internal transaction, and every read-write iterator has an associated internal transaction that is committed when the iterator is closed.

Using Transactions with C++

You must ensure that you either commit or roll back each transaction that you begin (otherwise, locks will be held by the database until they time out):

```
ConnectionPtr connection = ...;

TransactionPtr tx = connection->beginTransaction();
try {

    // DB updates that might throw here...

    tx->commit();

    // More code that might throw here...

} catch (...) {
    try {
        tx->rollback();
```

```
    } catch (...) {
    }
    throw;
}
```

The outer try-catch blocks are necessary because, if the code encounters an exception, we must roll back any updates that were made. In turn, the attempt to roll back might throw itself, namely, if the code following the commit throws an exception (in which case the transaction cannot be rolled back because it is already committed).

Code such as this is difficult to maintain: for example, an early return statement can cause the transaction to be neither committed nor rolled back. The `TransactionHolder` class ensures that such errors cannot happen:

```
namespace Freeze {
    class TransactionHolder {
    public:
        TransactionHolder(const ConnectionPtr&);
        ~TransactionHolder();

        void commit();
        void rollback();

    private:
        // Copy and assignment are forbidden.
        TransactionHolder(const TransactionHolder&);
        TransactionHolder& operator=(const TransactionHolder&);
    };
}
```

The constructor calls `beginTransaction` if the connection does not already have a transaction in progress, so instantiating the holder also starts a transaction. When the holder instance goes out of scope, its destructor calls `rollback` on the transaction and suppresses any exceptions that the rollback attempt might throw. This ensures that the transaction is rolled back if it was not previously committed or rolled back and ensures that an early return or an exception cannot cause the transaction to remain open:

```
ConnectionPtr connection = ...;

{ // Open scope

    TransactionHolder tx(connection); // Begins transaction

    // DB updates that might throw here...
```

```
    tx.commit();

    // More code that might throw here...

} // Transaction rolled back here if not previously
  // committed or rolled back.
```

If you instantiate a `TransactionHolder` when a transaction is already in progress, it does nothing: the constructor notices that it could not begin a new transaction and turns `commit`, `rollback`, and the destructor into no-ops. For example, the nested `TransactionHolder` instance in the following code is benign and does nothing:

```
ConnectionPtr connection = ...;

{ // Open scope

    TransactionHolder tx(connection); // Begins transaction

    // DB updates that might throw here...

    { // Open nested scope

        TransactionHolder tx2(connection); // Does nothing

        // DB updates that might throw here...

        tx2.commit(); // Does nothing

        // More code that might throw here...

    } // Destructor of tx2 does nothing

    tx.commit();

    // More code that might throw here...

} // Transaction rolled back here if not previously
  // committed or rolled back.
```

Using Transactions with Java

You must ensure that you either commit or roll back each transaction that you begin (otherwise, locks will be held by the database until they time out):

```
Connection connection = ...;

Transaction tx = connection.beginTransaction();
try {

    // DB updates that might throw here...

    tx.commit();

    // More code that might throw here...

} catch (java.lang.RuntimeException ex) {
    try {
        tx.rollback();
    } catch (DatabaseException e) {
    }
    throw ex;
}
```

The catch handler ensures that the transaction is rolled back before re-throwing the exception. Note that the nested try-catch blocks are necessary: if the transaction committed successfully but the code following the commit throws an exception, the rollback attempt will fail therefore we need to suppress the corresponding `DatabaseException` that is raised in that case.

Also use caution with early `return` statements:

```
Connection connection = ...;

Transaction tx = connection.beginTransaction();
try {

    // DB updates that might throw here...

    if (error) {
        // ...
        return; // Oops, bad news!
    }

    // ...

    tx.commit();

    // More code that might throw here...

} catch (java.lang.RuntimeException ex) {
```

```
    try {
        tx.rollback();
    } catch (DatabaseException e) {
    }
    throw ex;
}
```

The early `return` statement in the preceding code causes the transaction to be neither committed nor rolled back. To deal with this situation, avoid early return statements or ensure that you either commit or roll back the transaction before returning. Alternatively, you can use a `finally` block to ensure that the transaction is rolled back:

```
Connection connection = ...;

try {

    Transaction tx = connection.beginTransaction();

    // DB updates that might throw here...

    if (error) {
        // ...
        return; // No problem, see finally block.
    }

    // ...

    tx.commit();

    // More code that might throw here...

} finally {
    if (connection.currentTransaction() != null)
        connection.currentTransaction().rollback();
}
```

39.5.3 Iterators

Iterators allow you to traverse the contents of a Freeze map. Iterators are implemented using Berkeley DB cursors and acquire locks on the underlying database page files. In C++, both read-only (`const_iterator`) and read-write iterators (`iterator`) are available. In Java, an iterator is read-write if it is obtained in the context of a transaction and read-only if it is obtained outside a transaction.

Locks held by an iterator are released when the iterator is closed (if you do not use transactions) or when the enclosing transaction ends. Releasing locks held by iterators is very important to let other threads access the database file through other connection and map objects. Occasionally, it is even necessary to release locks to avoid self-deadlock (waiting forever for a lock held by an iterator created by the same thread).

To improve ease of use and make self-deadlocks less likely, Freeze often closes iterators automatically. If you close a map or connection, associated iterators are closed. Similarly, when you start or end a transaction, Freeze closes all the iterators associated with the corresponding maps. If you do not use transactions, any write operation on a map (such as inserting a new element) automatically closes all iterators opened on the same map object, except for the current iterator when the write operation is performed through that iterator. In Java, Freeze also closes a read-only iterator when no more elements are available.

There is, however, one situation in C++ where an explicit iterator close is needed to avoid self-deadlock:

- you do not use transactions, and
- you have an open iterator that was used to update a map (it holds a write lock), and
- in the same thread, you read that map.

Read operations in C++ never close iterators automatically: you need to either use transactions or explicitly close the iterator that holds the write lock. This is not an issue in Java because you cannot use an iterator to update a map outside of a transaction.

39.5.4 Recovering from Deadlock Exceptions

If you use multiple threads to access a database file, Berkeley DB may acquire locks in conflicting orders (on behalf of different transactions or iterators). For example, an iterator could have a read-lock on page P1 and attempt to acquire a write-lock on page P2, while another iterator (on a different map object associated with the same database file) could have a read-lock on P2 and attempt to acquire a write-lock on P1.

When this occurs, Berkeley DB detects a deadlock and resolves it by returning a “deadlock” error to one or more threads. For all non-iterator operations performed outside any transaction, such as an insertion into a map, Freeze catches such errors and automatically retries the operation until it succeeds. (In that case, the most-recently acquired lock is released before retrying.) For other operations,

Freeze reports this deadlock by raising `Freeze::DeadlockException`. In that case, the associated transaction or iterator is also automatically rolled back or closed. A properly written application is expected to catch deadlock exceptions and retry the transaction or iteration.

39.5.5 Key Sorting

Keys in Freeze maps and indexes are always sorted. By default, Freeze sorts keys according to their Ice-encoded binary representation; this is very efficient but the resulting order is rarely meaningful for the application. Starting with Ice 3.0, Freeze offers the ability to specify your own comparator objects so that you can customize the traversal order of your maps. Note however that the comparator of a Freeze map should remain the same throughout the life of the map. Berkeley DB stores records according to the key order provided by this comparator; switching to another comparator will cause undefined behavior.

C++

In C++, you specify the name of your comparator objects during code generation. The generated map provides the standard features of `std::map`, so that iterators return entries according to the order you have defined for the main key with your comparator object. The `lower_bound`, `upper_bound`, and `equal_range` provide range-searches (see the definition of these functions on `std::map`).

Apart from these standard features, the generated map provides additional functions and methods to perform range searches using secondary keys. The additional functions are `lowerBoundFor`*`Member`*, `upperBoundFor`*`Member`*, and `equalRangeFor`*`Member`*, where *`Member`* is the name of the secondary-key member. These functions return regular iterators on the Freeze map.

Java

In Java, you supply comparator objects (instances of the standard Java interface `java.util.Comparator`) at run time when instantiating the generated map class. The map constructor accepts a comparator for the main key and optionally a collection of comparators for secondary keys. The map also provides a number of methods for performing range searches on the main key and on secondary keys (see page 1563).

39.5.6 Indexing a Map

Freeze maps support efficient reverse lookups: if you define an index when you generate your map (with **`slice2freeze`** or **`slice2freezej`**), the generated code provides additional methods for performing reverse lookups. If your value type is a structure or a class, you can also index on a member of the value, and several such indexes can be associated with the same Freeze map.

Indexed searches are easy to use and very efficient. However, be aware that an index adds significant write overhead: with Berkeley DB, every update triggers a read from the database to get the old index entry and, if necessary, replace it.

If you later add an index to an existing map, Freeze automatically populates the index the next time you open the map. Freeze populates the index by instantiating each map entry, so it is important that you register the object factories for any class types in your map before you open the map.

Note that the index key comparator of a Freeze map index should remain the same throughout the life of the index. Berkeley DB stores records according to the key order provided by this comparator; switching to another comparator will cause undefined behavior.

39.5.7 Using a Freeze Map in C++

This section describes the code generator and demonstrates how to use a Freeze map in a C++ program.

`slice2freeze` Command-Line Options

The Slice-to-Freeze compiler, **`slice2freeze`**, creates C++ classes for Freeze maps. The compiler offers the following command-line options in addition to the standard options described in Section 4.20:

- **`--header-ext`** ***`EXT`***

  Changes the file extension for the generated header files from the default `h` to the extension specified by ***`EXT`***.

- **`--source-ext`** ***`EXT`***

  Changes the file extension for the generated source files from the default `cpp` to the extension specified by ***`EXT`***.

- **`--add-header`** ***`HDR`***`[,`***`GUARD`***`]`

  This option adds an include directive for the specified header at the beginning of the generated source file (preceding any other include directives). If ***`GUARD`*** is specified, the include directive is protected by the specified guard. For

example, **`--add-header precompiled.h,__PRECOMPILED_H__`** results in the following directives at the beginning of the generated source file:

```
#ifndef __PRECOMPILED_H__
#define __PRECOMPILED_H__
#include <precompiled.h>
#endif
```

As this example demonstrates, the **`--add-header`** option is useful mainly for integrating the generated code with a compiler's precompiled header mechanism.

This option can be repeated to create include directives for several files.

- **`--include-dir`** ***`DIR`***

  Modifies `#include` directives in source files to prepend the path name of each header file with the directory ***`DIR`***. See Section 6.16.1 for more information.

- **`--dll-export`** ***`SYMBOL`***

  Use ***`SYMBOL`*** to control DLL exports or imports. See the **`slice2cpp`** description for details.

- **`--dict`** ***`NAME`***`,`***`KEY`***`,`***`VALUE`***`[,sort[,`***`COMPARE`***`]]`**

  Generate a Freeze map class named ***`NAME`*** using ***`KEY`*** as key and ***`VALUE`*** as value. This option may be specified multiple times to generate several Freeze maps. ***`NAME`*** may be a scoped C++ name, such as `Demo::Struct1ObjectMap`. By default, keys are sorted using their binary Ice-encoded representation. Include **`sort`** to sort with the ***`COMPARE`*** functor class. If ***`COMPARE`*** is not specified, the default value is `std::less<`***`KEY`***`>`.

- **`--dict-index`** ***`MAP`***`[,`***`MEMBER`***`]`
  `[,case-sensitive|case-insensitive][,sort[,`*`COMPARE`***`]]`**

  Add an index to the Freeze map named ***`MAP`***. If ***`MEMBER`*** is specified, the map value type must be a structure or a class, and ***`MEMBER`*** must be a member of this structure or class. Otherwise, the entire value is indexed. When the indexed member (or entire value) is a string, the index can be case-sensitive (default) or case-insensitive. An index adds additional member functions to the generated C++ map:

  - `iterator findBy`*`MEMBER`*`(`*`MEMBER_TYPE`*`, bool = true);`
  - `const_iterator findBy`*`MEMBER`*`(`*`MEMBER_TYPE`*`,`
    `bool = true) const;`

- `iterator beginFor`*`MEMBER`*`();`
- `const_iterator beginFor`*`MEMBER`*`() const;`
- `iterator endFor`*`MEMBER`*`();`
- `const_iterator endFor`*`MEMBER`*`() const;`
- `iterator lowerBoundFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`);`
- `const_iterator lowerBoundFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`) const;`
- `iterator upperBoundFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`);`
- `const_iterator upperBoundFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`) const;`
- `std::pair<iterator, iterator>`
  `equalRangeFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`);`
- `std::pair<const_iterator, const_iterator>`
  `equalRangeFor`*`MEMBER`*`(`*`MEMBER_TYPE`*`) const;`
- `int `*`MEMBER`*`Count(`*`MEMBER_TYPE`*`) const;`

When ***MEMBER*** is not specified, these functions are `findByValue` (const and non-const), `lowerBoundForValue` (const and non-const), `valueCount`, and so on. When ***MEMBER*** is specified, its first letter is capitalized in the `findBy` function name. *`MEMBER_TYPE`* corresponds to an in-parameter of the type of ***MEMBER*** (or the type of the value when ***MEMBER*** is not specified). For example, if ***MEMBER*** is a string, *`MEMBER_TYPE`* is a `const std::string&`.

By default, keys are sorted using their binary Ice-encoded representation. Include **`sort`** to sort with the ***COMPARE*** functor class. If ***COMPARE*** is not specified, the default value is `std::less<`***MEMBER_TYPE***`>`.

`findBy`*`MEMBER`* returns an iterator to the first element in the Freeze map that matches the given index value. It returns `end()` if there is no match. When the second parameter is true (the default), the returned iterator provides only the elements with an exact match (and then skips to `end()`). Otherwise, the returned iterator sets a starting position and then provides all elements until the end of the map, sorted according to the index comparator.

`lowerBoundFor`*`MEMBER`* returns an iterator to the first element in the Freeze map whose index value is not less than the given index value. It returns `end()` if there is no such element. The returned iterator provides all elements until the end of the map, sorted according to the index comparator.

`upperBoundFor`*`MEMBER`* returns an iterator to the first element in the Freeze map whose index value is greater than the given index value. It returns

`end()` if there is no such element. The returned iterator provides all elements until the end of the map, sorted according to the index comparator.

`beginFor`*`MEMBER`* returns an iterator to the first element in the map.

`endFor`*`MEMBER`* returns an iterator to the last element in the map.

`equalRangeFor`*`MEMBER`* returns a range (pair of iterators) of all the elements whose index value matches the given index value. This function is similar to `findBy`*`MEMBER`* (see above).

`MEMBER``Count` returns the number of elements in the Freeze map whose index value matches the given index value.

Please note that index-derived iterators do not allow you to set new values in the underlying map.

- **`--index`** ***`CLASS`***`,`***`TYPE`***`,`***`MEMBER`***
  `[,case-sensitive|case-insensitive]`

  Generate an index class for a Freeze evictor (see Section 39.3.7). ***`CLASS`*** is the name of the class to be generated. ***`TYPE`*** denotes the type of class to be indexed (objects of different classes are not included in this index). ***`MEMBER`*** is the name of the data member in ***`TYPE`*** to index. When ***`MEMBER`*** has type `string`, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

Section 6.16.1 provides a discussion of the semantics of `#include` directives that is also relevant for users of **`slice2freeze`**.

Generating a Simple Map

As an example, the following command generates a simple map:

```
$ slice2freeze --dict StringIntMap,string,int StringIntMap
```

This command directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. The final argument is the base name for the output files, to which the compiler appends the `.h` and `.cpp` suffixes. As a result, this command produces two C++ source files, `StringIntMap.h` and `StringIntMap.cpp`.

The Map Class

If you examine the contents of the header file created by the example in the previous section, you will discover that a Freeze map is an instance of the template class `Freeze::Map`:

```
// StringIntMap.h
typedef Freeze::Map<std::string, Ice::Int, ...> StringIntMap;
```

The `Freeze::Map` template class closely resembles the STL container class `std::map`, as shown in the following class definition:

```
namespace Freeze {
template<...> class Map {
public:
    typedef ... value_type;
    typedef ... iterator;
    typedef ... const_iterator;

    typedef size_t size_type;
    typedef ptrdiff_t difference_type;

    Map(const Freeze::ConnectionPtr& connection,
        const std::string& dbName,
        bool createDb = true,
        const Compare& compare = Compare());

    template<class _InputIterator>
    Map(const Freeze::ConnectionPtr& connection,
        const std::string& dbName,
        bool createDb,
        _InputIterator first, _InputIterator last,
        const Compare& compare = Compare());

    static void recreate(const Freeze::ConnectionPtr& connection,
                         const std::string& dbName,
                         const Compare& compare = Compare());

    bool operator==(const Map& rhs) const;
    bool operator!=(const Map& rhs) const;

    void swap(Map& rhs);

    iterator begin();
    const_iterator begin() const;

    iterator end();
    const_iterator end() const;

    bool empty() const;
    size_type size() const;
    size_type max_size() const;
```

```
    iterator insert(iterator /*position*/,
                    const value_type& elem);

    std::pair<iterator, bool> insert(const value_type& elem);

    template <typename InputIterator>
    void insert(InputIterator first, InputIterator last);

    void put(const value_type& elem);

    template <typename InputIterator>
    void put(InputIterator first, InputIterator last);

    void erase(iterator position);
    size_type erase(const key_type& key);
    void erase(iterator first, iterator last);

    void clear();

    void destroy(); // Non-standard.

    iterator find(const key_type& key);
    const_iterator find(const key_type& key) const;

    size_type count(const key_type& key) const;

    iterator lower_bound(const key_type& key);
    const_iterator lower_bound(const key_type& key) const;
    iterator upper_bound(const key_type& key);
    const_iterator upper_bound(const key_type& key) const;

    std::pair<iterator, iterator>
    equal_range(const key_type& key);

    std::pair<const_iterator, const_iterator>
    equal_range(const key_type& key) const;

    const Ice::CommunicatorPtr&
    communicator() const;

    ...
};
}
```

The semantics of the `Freeze::Map` methods are identical to those of `std::map` unless otherwise noted. In particular, the overloaded `insert` method shown below ignores the `position` argument:

```
iterator insert(iterator /*position*/,
                const value_type& elem);
```

A Freeze map class supports only those methods shown above; other features of `std::map`, such as allocators and overloaded array operators, are not available.

Non-standard methods that are specific to Freeze maps are discussed below:

- Constructors

  The following overloaded constructors are provided:

  ```
  Map(const Freeze::ConnectionPtr& connection,
      const std::string& dbName,
      bool createDb = true,
      const Compare& compare = Compare());

  template<class _InputIterator>
  Map(const Freeze::ConnectionPtr& connection,
      const std::string& dbName,
      bool createDb,
      _InputIterator first, _InputIterator last,
      const Compare& compare = Compare());
  ```

  The first constructor accepts a connection, the database name, a flag indicating whether to create the database if it does not exist, and an object used to compare keys. The second constructor accepts all of the parameters of the first, with the addition of iterators from which elements are copied into the map.

  Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.

- Map copy

  The `recreate` function copies an existing database:

  ```
  static void recreate(const Freeze::ConnectionPtr& connection,
                       const std::string& dbName,
                       const Compare& compare = Compare())
  ```

  The `dbName` parameter specifies an existing database name. The copy has the name*<dbName>*`.old-`*<uuid>*. For example, if the database name is `MyDB`, the copy might be named

`MyDB.old-edefd55a-e66a-478d-a77b-f6d53292b873`. (Obviously, a different UUID is used each time you recreate a database).

- `destroy`

  This method deletes the database from its environment and from the Freeze catalog (see Section 39.7). If a transaction is not currently open, the method creates its own transaction in which to perform this task.

- `communicator`

  This method returns the communicator with which the map's connection is associated.

Iterators

A Freeze map's iterator works like its counterpart in `std::map`. The iterator class supports one convenient (but nonstandard) method:

```
void set(const mapped_type& value)
```

Using this method, a program can replace the value at the iterator's current position.

Sample Program

The program below demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit `read` or `write` operations called by the program; instead, simply using the map has the side effect of accessing the database.

```
#include <Freeze/Freeze.h>
#include <StringIntMap.h>

int
main(int argc, char* argv[])
{
    // Initialize the Communicator.
    //
    Ice::CommunicatorPtr communicator =
        Ice::initialize(argc, argv);

    // Create a Freeze database connection.
    //
    Freeze::ConnectionPtr connection =
        Freeze::createConnection(communicator, "db");

    // Instantiate the map.
```

```
    //
    StringIntMap map(connection, "simple");

    // Clear the map.
    //
    map.clear();

    Ice::Int i;
    StringIntMap::iterator p;

    // Populate the map.
    //
    for (i = 0; i < 26; i++) {
        std::string key(1, 'a' + i);
        map.insert(make_pair(key, i));
    }

    // Iterate over the map and change the values.
    //
    for (p = map.begin(); p != map.end(); ++p)
        p.set(p->second + 1);

    // Find and erase the last element.
    //
    p = map.find("z");
    assert(p != map.end());
    map.erase(p);

    // Clean up.
    //
    connection->close();
    communicator->destroy();

    return 0;
}
```

Prior to instantiating a Freeze map, the application must connect to a Berkeley DB database environment:

```
Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator, "db");
```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files. Note that properties with the prefix `Freeze.DbEnv` can modify a number of environment settings (see page 1950),

including the file system directory. For the preceding example, you could change the directory to `FreezeDir` by setting the property `Freeze.DbEnv.db.DbHome` to `FreezeDir`.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, which by default is created if it does not exist:

```
StringIntMap map(connection, "simple");
```

After instantiating the map, we clear it to make sure it is empty in case the program is run more than once:

```
map.clear();
```

Next, we populate the map using a single-character string as the key:

```
for (i = 0; i < 26; i++) {
    std::string key(1, 'a' + i);
    map.insert(make_pair(key, i));
}
```

Iterating over the map will look familiar to `std::map` users. However, to modify a value at the iterator's current position, we use the nonstandard `set` method:

```
for (p = map.begin(); p != map.end(); ++p)
    p.set(p->second + 1);
```

Next, the program obtains an iterator positioned at the element with key `z`, and erases it:

```
p = map.find("z");
assert(p != map.end());
map.erase(p);
```

Finally, the program closes the database connection:

```
connection->close();
```

It is not necessary to explicitly close the database connection, but we demonstrate it here for the sake of completeness.

39.5.8 Using a Freeze Map in Java

This section describes the code generator and demonstrates how to use a Freeze map in a Java program.

`slice2freezej` Command-Line Options

The Slice-to-Freeze compiler, **`slice2freezej`**, creates Java classes for Freeze maps. The compiler offers the following command-line options in addition to the standard options described in Section 4.20:

- **`--dict`** ***`NAME`***,***`KEY`***,***`VALUE`***

  Generate a Freeze map class named ***`NAME`*** using ***`KEY`*** as key and ***`VALUE`*** as value. This option may be specified multiple times to generate several Freeze maps. ***`NAME`*** may be a scoped Java name, such as `Demo.Struct1ObjectMap`.

- **`--dict-index`** ***`MAP`***[,***`MEMBER`***] [,**`case-sensitive`**|**`case-insensitive`**]**

  Add an index to the Freeze map named ***`MAP`***. If ***`MEMBER`*** is specified, the map value type must be a structure or a class, and ***`MEMBER`*** must be the name of a member of that type. If ***`MEMBER`*** is not specified, the entire value is indexed. When the indexed member (or entire value) is a string, the index can be case-sensitive (default) or case-insensitive. See page 1569 for more information on the Freeze map API for indices.

- **`--index`** ***`CLASS`***,***`TYPE`***,***`MEMBER`*** [,**`case-sensitive`**|**`case-insensitive`**]**

  Generate an index class for a Freeze evictor (see Section 39.3.7). ***`CLASS`*** is the name of the index class to be generated. ***`TYPE`*** denotes the type of class to be indexed (objects of different classes are not included in this index). ***`MEMBER`*** is the name of the data member in ***`TYPE`*** to index. When ***`MEMBER`*** has type `string`, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

- **`--meta`** ***`META`***

  Define the global metadata directive ***`META`***. Using this option is equivalent to defining the global metadata ***`META`*** in each named Slice file, as well as in any file included by a named Slice file.

Generating a Simple Map

As an example, the following command generates a simple map:

```
$ slice2freezej --dict StringIntMap,string,int
```

This command directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. The compiler produces one Java source file: `StringIntMap.java`.

Ant Task

In addition to the ant task for executing **`slice2java`**, Ice also includes an ant task for executing **`slice2freezej`**. The classes for `Slice2FreezeJTask` are stored in the same JAR file (`ant-ice.jar`) as `Slice2JavaTask`. Both tasks also share the same logic for locating a compiler in your execution environment and for managing dependencies between Slice files; refer to Section 10.18.2 for more information.

The `Slice2FreezeJTask` supports the parameters listed in Table 39.1:

Table 39.1. Ant task parameters

Attribute	Description	Required
`dependencyfile`	Specifies an alternate name for the dependency file. If you specify a relative filename, it is relative to ant's current working directory. If not specified, the task uses the name `.depend` by default. If you do not define this attribute and `outputdir` is defined, the task creates the `.depend` file in the designated output directory (see `outputdir`).	No
`ice`	Instructs the Slice compiler to permit symbols that have the reserved prefix `Ice`. This parameter is used in the Ice build system and is not normally required by applications.	No
`outputdir`	Specifies the directory in which the Slice compiler generates Java source files. If not specified, the task uses ant's current working directory.	No
`translator`	Specifies the path name of the Slice compiler. If not specified, the task locates the Slice compiler in its execution environment as described in Section 10.18.2.	No

Several Slice compiler options must be defined as nested elements of the task:

- `define`

  Defines a preprocessor macro. The element supports the attributes `name` and (optionally) `value`, as shown below:

  ```
  <define name="FOO">
  <define name="BAR" value="5">
  ```

  These definitions are equivalent to the command-line options `-DFOO` and `-DBAR=5`, respectively.

- `dict`

  Generates a Freeze map. This element is equivalent to the **`--dict`** option and supports three attributes: `name`, `key`, and `value`. Refer to page 1560 for more information on this option.

- `dictindex`

  Generates an index for a Freeze map. This element is equivalent to the **`--dict-index`** option and supports three attributes: `name`, `member`, and `casesensitive`. Refer to page 1560 for more information on this option.

- `fileset`

  Specifies the set of Slice files to be compiled. Refer to the ant documentation of its `FileSet` type for more information.

- `includepath`

  Specifies the include file search path for Slice files. In ant terminology, `includepath` is a *path-like structure*. Refer to the ant documentation of its `Path` type for more information.

- `index`

  Generates an index for a Freeze evictor. This element is equivalent to the **`--index`** option and supports four attributes: `name`, `type`, `member`, and `casesensitive`. Refer to page 1560 for more information on this option.

- `meta`

  Defines a global metadata directive in each Slice file as well as in each included Slice file. The element supports the attributes `name` and `value`.

To enable the `Slice2FreezeJTask` in your ant project, define the following `taskdef` element in your project's build file:

```
<taskdef name="slice2freezej" classname="Slice2FreezeJTask"/>
```

This configuration assumes that `ant-ice.jar` is already present in ant's class path. Alternatively, you can specify the JAR explicitly as follows:

```
<taskdef name="slice2freezej" classpath="/opt/Ice/lib/ant-ice.jar"
    classname="Slice2FreezeJTask"/>
```

Once activated, you can invoke the task to translate your Slice files. The example shown below is a simplified version of the ant project for the `library` demo:

```
<target name="generate" depends="init">
    <mkdir dir="generated"/>
    <slice2java outputdir="generated">
        <fileset dir="." includes="Library.ice"/>
    </slice2java>
    <slice2freezej ice="on" outputdir="generated">
        <fileset dir="/opt/Ice/slice/Ice"
            includes="BuiltinSequences.ice"/>
        <fileset dir="." includes="Library.ice"/>
        <dict name="StringIsbnSeqDict" key="string"
            value="Ice::StringSeq"/>
    </slice2freezej>
</target>
```

This invocation of the `slice2freezej` task enables the `ice` option because the generated Freeze map relies on a type that is defined in an Ice namespace and therefore loads the Slice file `BuiltinSequences.ice` directly.

The Map Class

The class generated by **`slice2freezej`** implements the `Freeze.Map` interface, as shown below:

```
package Freeze;

public interface Map<K, V> extends NavigableMap<K, V>
{
    void fastPut(K key, V value);
    void close();
    int closeAllIterators();
    void destroy();

    public interface EntryIterator<T>
        extends java.util.Iterator<T>
    {
        void close();
        void destroy(); // an alias for close
    }
}
```

The `Map` interface implements standard Java interfaces and provides nonstandard methods that improve efficiency and support database-oriented features. `Map` defines the following methods:

- `fastPut`

  Inserts a new key-value pair. This method is more efficient than the standard `put` method because it avoids the overhead of reading and decoding the previous value associated with the key (if any).

- `close`

  Closes the database associated with this map along with all open iterators. A map must be closed when it is no longer needed, either by closing the map directly or by closing the Freeze `Connection` object with which this map is associated.

- `closeAllIterators`

  Closes all open iterators and returns the number of iterators that were closed. We discuss iterators in more detail in the next section.

- `destroy`

  Removes the database associated with this map along with any indices.

`Map` inherits much of its functionality from the `Freeze.NavigableMap` interface, which derives from the standard Java interface `java.util.SortedMap` and also supports a subset of the `java.util.NavigableMap`[2] interface from Java6:

```
package Freeze;

public interface NavigableMap<K, V>
    extends java.util.SortedMap<K, V>
{
    java.util.Map.Entry<K, V> firstEntry();
    java.util.Map.Entry<K, V> lastEntry();

    java.util.Map.Entry<K, V> ceilingEntry(K key);
    java.util.Map.Entry<K, V> floorEntry(K key);
    java.util.Map.Entry<K, V> higherEntry(K key);
    java.util.Map.Entry<K, V> lowerEntry(K key);
```

2. The generated class does not implement `java.util.NavigableMap` because Freeze maps must remain compatible with Java5.

```
    K ceilingKey(K key);
    K floorKey(K key);
    K higherKey(K key);
    K lowerKey(K key);

    java.util.Set<K> descendingKeySet();
    NavigableMap<K, V> descendingMap();

    NavigableMap<K, V> headMap(K toKey, boolean inclusive);
    NavigableMap<K, V> tailMap(K fromKey, boolean inclusive);
    NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive,
                              K toKey, boolean toInclusive);

    java.util.Map.Entry<K, V> pollFirstEntry();
    java.util.Map.Entry<K, V> pollLastEntry();

    boolean fastRemove(K key);
}
```

The `NavigableMap` interface provides a number of useful methods:

- `firstEntry`
  `lastEntry`

  Returns the first and last key-value pair, respectively.

- `ceilingEntry`

  Returns the key-value pair associated with the least key greater than or equal to the given key, or null if there is no such key.

- `floorEntry`

  Returns the key-value pair associated with the greatest key less than or equal to the given key, or null if there is no such key.

- `higherEntry`

  Returns the key-value pair associated with the least key greater than the given key, or null if there is no such key.

- `lowerEntry`

  Returns the key-value pair associated with the greatest key less than the given key, or null if there is no such key.

- `ceilingKey`
  `floorKey`

`higherKey`
`lowerKey`

These methods have the same semantics as those described above, except they return only the key portion of the matching key-value pair or null if there is no such key.

- `descendingKeySet`

  Returns a set representing a reverse-order view of the keys in this map.

- `descendingMap`

  Returns a reverse-order view of the entries in this map.

- `headMap`

  Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) the given key.

- `tailMap`

  Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) the given key.

- `subMap`

  Returns a view of the portion of this map whose keys are within the given range.

- `pollFirstEntry`
  `pollLastEntry`

  Removes and returns the first and last key-value pair, respectively.

- `fastRemove`

  Removes an existing key-value pair. As for `fastPut`, this method is a more efficient alternative to the standard `remove` method that returns true if a key-value pair was removed, or false if no match was found.

Many of these methods raise `UnsupportedOperationException` if you fail to construct the Freeze map using a custom comparator object. The only exceptions are `firstEntry`, `lastEntry`, `pollFirstEntry`, `pollLastEntry`, and `fastRemove`. (The same applies to `NavigableMap` objects created for secondary keys.)

Note that `NavigableMap` also inherits overloaded methods named `headMap`, `tailMap`, and `subMap` from the `SortedMap` interface. These methods have the same semantics as the ones defined in `NavigableMap` but they omit the boolean arguments (refer to the JDK documentation for complete details). Although these methods are declared as returning a `SortedMap`, the

actual type of the returned object is a `NavigableMap` that you can downcast if necessary.

There are some limitations in the sub maps returned by the `headMap`, `tailMap` and `subMap` methods:

- A new entry in the Freeze map cannot be added via a sub map, therefore calling `put` raises `UnsupportedOperationException`
- An existing entry in the Freeze map cannot be removed via a sub map or iterator for a secondary key (see page 1569).

Now let us examine the contents of the source file created by the example in the previous section:

```
public class StringIntMap extends ...
    // implements Freeze.Map<String, Integer>
{
    public StringIntMap(
        Freeze.Connection connection,
        String dbName,
        boolean createDb,
        java.util.Comparator<String> comparator);

    public StringIntMap(
        Freeze.Connection connection,
        String dbName,
        boolean createDb);

    public StringIntMap(
        Freeze.Connection connection,
        String dbName);
}
```

`StringIntMap` derives from an internal Freeze base class that implements the interface `Freeze.Map<String, Integer>`. The generated class defines several overloaded constructors whose arguments are described below:

- `connection`

  The Freeze connection object (see Section 39.5.1).

- `dbName`

  The name of the database in which to store this map's persistent state. Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.

- `createDb`

  A flag indicating whether the map should create the database if it does not already exist. If this argument is not specified, the default value is `true`.

- `comparator`

  An object used to compare the map's keys. If this argument is not specified, the default behavior compares the encoded form of the keys.

Iterators

You can iterate over a Freeze map just as you can with any container that implements the `java.util.Map` interface. For example, the code below displays the key and value of each element:

```
StringIntMap m = new StringIntMap(...);
java.util.Iterator<java.util.Map.Entry<String, Integer>> i =
    m.entrySet().iterator();
while (i.hasNext()) {
    java.util.Map.Entry<String, Integer> e = i.next();
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
}
```

Generally speaking, a program should close an iterator when it is no longer necessary for the reasons given in Section 39.5.3. (An iterator that is garbage collected without being closed emits a warning message.) However, an explicit close was not necessary in the preceding example because Freeze automatically closes a read-only iterator when it reaches the last element (a read-only iterator is one that is opened outside of any transaction). If instead our program had stopped using the iterator prior to reaching the last element, an explicit close would have been necessary:

```
StringIntMap m = new StringIntMap(...);
java.util.Iterator<java.util.Map.Entry<String, Integer>> i =
    m.entrySet().iterator();
while (i.hasNext()) {
    java.util.Map.Entry<String, Integer> e = i.next();
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
    if (e.getValue().intValue() == 5)
        break;
}
((Freeze.Map.EntryIterator)i).close();
```

Closing the iterator requires downcasting it to a Freeze-specific interface named `Freeze.Map.EntryIterator`. The definition of this interface was shown in the previous section.

Freeze maps also support the enhanced `for` loop functionality in Java5. Here is a simpler way to write our original program:

```
StringIntMap m = new StringIntMap(...);
for (java.util.Map.Entry<String, Integer> e : m.entrySet()) {
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
}
```

As in the first example, Freeze automatically closes the iterator when no more elements are available. Although the enhanced `for` loop is convenient, it is not appropriate for all situations because the loop hides its iterator and therefore prevents the program from accessing the iterator in order to close it. In this case, you can use the traditional `while` loop instead of the `for` loop, or you can invoke `closeAllIterators` on the map as shown below:

```
StringIntMap m = new StringIntMap(...);
for (java.util.Map.Entry<String, Integer> e : m.entrySet()) {
    System.out.println("Key: " + e.getKey());
    System.out.println("Value: " + e.getValue());
    if (e.getValue().intValue() == 5)
        break;
}
int num = m.closeAllIterators();
assert(num <= 1); // The iterator may already be closed.
```

The `closeAllIterators` method returns an integer representing the number of iterators that were actually closed. This value can be useful for diagnostic purposes, such as to assert that a program is correctly closing its iterators.

Indices

Using the **`--dict-index`** option to define an index for a secondary key causes **`slice2freezej`** to generate the following additional code in a Freeze map:

- A static nested class named `IndexComparators`, which allows you to supply a custom comparator object for each index in the map.
- An overloading of the map constructor that accepts an instance of `IndexComparators`.
- An overloading of the `recreate` method that accepts an instance of `IndexComparators`.

- Searching, counting, and range-searching methods for finding key-value pairs using the secondary key.

We discuss each of these additions in more detail below. In this discussion, ***MEMBER*** refers to the optional argument of the **`--dict-index`** option, and ***MEMBER_TYPE*** refers to the type of that member. As explained earlier, if ***MEMBER*** is not specified, **`slice2freezej`** creates an index for the value type of the map. The sample code presented in this section assumes we have generated a Freeze map using the following command:

```
$ slice2freezej --dict StringIntMap,string,int \
    --dict-index StringIntMap
```

By default, index keys are sorted using their binary Ice-encoded representation. This is an efficient sorting scheme but does not necessarily provide a meaningful traversal order for applications. You can choose a different order by providing an instance of the `IndexComparators` class to the map constructor. This class has a public data member holding a comparator (an instance of `java.util.Comparator<`***MEMBER_TYPE***`>`) for each index in the map. The class also provides an empty constructor as well as a convenience constructor that allows you to instantiate and initialize the object all at once. The name of each data member is ***MEMBER***`Comparator`. If ***MEMBER*** is not specified, the `IndexComparators` class has a single data member named `valueComparator`. Note that much of the functionality offered by a map index requires that you provide a custom comparator.

Here is the definition of `IndexComparators` for `StringIntMap`:

```
public class StringIntMap ... {
    public static class IndexComparators {
        public IndexComparators() {}

        public IndexComparators(
            java.util.Comparator<Integer> valueComparator);

        public java.util.Comparator<Integer> valueComparator;
    }

    ...
}
```

To instantiate a Freeze map using your custom comparators, you must use the overloaded constructor that accepts the `IndexComparators` object. For our `StringIntMap`, this constructor has the following definition:

```
public class StringIntMap ... {
    public StringIntMap(
        Freeze.Connection connection,
        String dbName,
        boolean createDb,
        java.util.Comparator<String> comparator,
        IndexComparators indexComparators);

    ...
}
```

Now we can instantiate our `StringIntMap` as follows:

```
java.util.Comparator<String> myMainKeyComparator = ...;
StringIntMap.IndexComparators indexComparators =
    new StringIntMap.IndexComparators();
indexComparators.valueComparator = ...;
StringIntMap m =
    new StringIntMap(connection, "stringIntMap", true,
                     myMainKeyComparator, indexComparators);
```

If you later need to change the index configuration of a Freeze map, you can use one of the `recreate` methods to update the database. Here are the definitions from `StringIntMap`:

```
public class StringIntMap ... {
    public static void recreate(
        Freeze.Connection connection,
        String dbName,
        java.util.Comparator<String> comparator);

    public static void recreate(
        Freeze.Connection connection,
        String dbName,
        java.util.Comparator<String> comparator,
        IndexComparators indexComparators);

    ...
}
```

The first overloading is generated for every map, whereas the second overloading is only generated when the map has at least one index. As its name implies, the `recreate` method creates a new copy of the database. More specifically, the method removes any existing indices, copies every key-value pair to a temporary database, and finally replaces the old database with the new one. As a side-effect, this process also populates any remaining indices. The first overloading of

`recreate` is useful when you have regenerated the map to remove the last index and wish to clean up the map's database state.

`slice2freezej` also generates a number of index-specific methods. The names of these methods incorporate the member name (***MEMBER***), or use `value` if ***MEMBER*** is not specified. In each method name, the value of ***MEMBER*** is used unchanged if it appears at the beginning of the method's name. Otherwise, if ***MEMBER*** is used elsewhere in the method name, its first letter is capitalized. The index methods are described below:

- `public Freeze.Map.EntryIterator<Map.Entry<K, V>> findByMEMBER(MEMBER_TYPE index)`

  `public Freeze.Map.EntryIterator<Map.Entry<K, V>> findByMEMBER(MEMBER_TYPE index, boolean onlyDups)`

  Returns an iterator over elements of the Freeze map starting with an element with whose index value matches the given index value. If there is no such element, the returned iterator is empty (`hasNext` always returns false). When the second parameter is true (or is not provided), the returned iterator provides only "duplicate" elements, that is, elements with the very same index value. Otherwise, the iterator sets a starting position in the map, and then provides elements until the end of the map, sorted according to the index comparator. Any attempt to modify the map via this iterator results in an `UnsupportedOperationException`.

- `public int` *`MEMBER`*`Count(`*`MEMBER_TYPE index`*`)`

  Returns the number of elements in the Freeze map whose index value matches the given index value.

- `public NavigableMap<`*`MEMBER_TYPE`*`, Set<Map.Entry<K, V>>> headMapFor`*`MEMBER`*`(`*`MEMBER_TYPE`*` to, boolean inclusive)`

  `public NavigableMap<`*`MEMBER_TYPE`*`, Set<Map.Entry<K, V>>> headMapFor`*`MEMBER`*`(`*`MEMBER_TYPE`*` to)`

  Returns a view of the portion of the Freeze map whose keys are less than (or equal to, if `inclusive` is true) the given key. If `inclusive` is not specified, the method behaves as if `inclusive` is false.

- `public NavigableMap<`*`MEMBER_TYPE`*`, Set<Map.Entry<K, V>>> tailMapFor`*`MEMBER`*`(`*`MEMBER_TYPE`*` from, boolean inclusive)`

```
public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>
tailMapForMEMBER(MEMBER_TYPE from)
```

Returns a view of the portion of the Freeze map whose keys are greater than (or equal to, if `inclusive` is true) the given key. If `inclusive` is not specified, the method behaves as if `inclusive` is true.

- ```
  public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>
  subMapForMEMBER(MEMBER_TYPE from, boolean fromInclusive,
                  MEMBER_TYPE to, boolean toInclusive)

  public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>
  subMapForMEMBER(MEMBER_TYPE from, MEMBER_TYPE to)
  ```

  Returns a view of the portion of the Freeze map whose keys are within the given range. If `fromInclusive` and `toInclusive` are not specified, the method behaves as if `fromInclusive` is true and `toInclusive` is false.

- ```
  public NavigableMap<MEMBER_TYPE, Set<Map.Entry<K, V>>>
  mapForMEMBER()
  ```

  Returns a view of the entire Freeze map ordered by the index key.

For the methods returning a `NavigableMap`, the key type is the secondary key type and the value is the set of matching key-value pairs from the Freeze map. (For the sake of readability, we have omitted the `java.util` prefix from `Set` and `Map.Entry`.) In other words, the returned map is a mapping of the secondary key to all of the entries whose value contains the same key. Any attempt to add, remove, or modify an element via a sub map view or an iterator of a sub map view results in an `UnsupportedOperationException`.

Note that iterators returned by the `findBy`*MEMBER* methods, as well as those created for sub map views, may need to be closed explicitly, just like iterators obtained for the main Freeze map. (See page 1568 for more information.)

Here are the definitions of the index methods for `StringIntMap`:

```
public Freeze.Map.EntryIterator<Map.Entry<String, Integer>>
findByValue(Integer index);

public Freeze.Map.EntryIterator<Map.Entry<String, Integer>>
findByValue(Integer index, boolean onlyDups);

public int valueCount(Integer index);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
headMapForValue(Integer to, boolean inclusive);
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
headMapForValue(Integer to);
```

```
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
tailMapForValue(Integer from, boolean inclusive);
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
tailMapForValue(Integer from);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
subMapForValue(Integer from, boolean fromInclusive,
               Integer to, boolean toInclusive);
public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
subMapForValue(Integer from, Integer to);

public NavigableMap<Integer, Set<Map.Entry<String, Integer>>>
mapForValue();
```

Sample Program

The program below demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit `read` or `write` operations called by the program; instead, simply using the map has the side effect of accessing the database.

```
public class Client
{
    public static void
    main(String[] args)
    {
        // Initialize the Communicator.
        //
        Ice.Communicator communicator = Ice.Util.initialize(args);

        // Create a Freeze database connection.
        //
        Freeze.Connection connection =
            Freeze.Util.createConnection(communicator, "db");

        // Instantiate the map.
        //
        StringIntMap map =
            new StringIntMap(connection, "simple", true);

        // Clear the map.
        //
        map.clear();

        int i;
```

```
        // Populate the map.
        //
        for (i = 0; i < 26; i++) {
            final char[] ch = { (char)('a' + i) };
            map.put(new String(ch), i);
        }

        // Iterate over the map and change the values.
        //
        for (java.util.Map.Entry<String, Integer> e :
             map.entrySet()) {
            Integer in = e.getValue();
            e.setValue(in.intValue() + 1);
        }

        // Find and erase the last element.
        //
        boolean b;
        b = map.containsKey("z");
        assert(b);
        b = map.fastRemove("z");
        assert(b);

        // Clean up.
        //
        map.close();
        connection.close();
        communicator.destroy();

        System.exit(0);
    }
}
```

Prior to instantiating a Freeze map, the application must connect to a Berkeley DB database environment:

```
Freeze.Connection connection =
    Freeze.Util.createConnection(communicator, "db");
```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, and the third argument indicates that the database should be created if it does not exist:

```
StringIntMap map = new StringIntMap(connection, "simple", true);
```

After instantiating the map, we clear it to make sure it is empty in case the program is run more than once:

```
map.clear();
```

We populate the map, using a single-character string as the key. As with `java.util.Map`, the key and value types must be Java objects but the compiler takes care of autoboxing the integer argument:

```
for (i = 0; i < 26; i++) {
    final char[] ch = { (char)('a' + i) };
    map.put(new String(ch), i);
}
```

Iterating over the map is no different from iterating over any other map that implements the `java.util.Map` interface:

```
for (java.util.Map.Entry<String, Integer> e :
     map.entrySet()) {
    Integer in = e.getValue();
    e.setValue(in.intValue() + 1);
}
```

Next, the program verifies that an element exists with key `z`, and then removes it using `fastRemove`:

```
b = map.containsKey("z");
assert(b);
b = map.fastRemove("z");
assert(b);
```

Finally, the program closes the map and its connection.

```
map.close();
connection.close();
```

39.6 Using a Freeze Map in the File System Server

We can use a Freeze map to add persistence to the file system server, and we present C++ and Java implementations in this section. However, as we describe in Section 39.3, a Freeze evictor is often a better choice for applications (such as the file system server) in which the persistent value is an Ice object.

In general, incorporating a Freeze map into your application requires the following steps:

1. Evaluate your existing Slice definitions for suitable key and value types.
2. If no suitable key or value types are found, define new (possibly derived) types that capture your persistent state requirements. Consider placing these definitions in a separate file: these types are only used by the server for persistence, and therefore do not need to appear in the "public" definitions required by clients. Also consider placing your persistent types in a separate module to avoid name clashes.
3. Generate a Freeze map for your persistent types using the Freeze compiler.
4. Use the Freeze map in your operation implementations.

39.6.1 Choosing Key and Value Types

Our goal is to implement the file system using Freeze maps for all persistent storage, including files and their contents. There are various options for how to implement the server. For this example, the server is stateless; whenever a client invokes an operation, the server accesses the database to satisfy the request. Implementing the server in this way has the advantage that it scales very well: we do not need a separate servant for each node; instead two default servants (see Section 32.8), one for directories and one for files, are sufficient. This keeps the memory requirements of the server to a minimum and also allows us to rely on the database for transactions and locking. (This is a very common implementation technique for servers that act as a front end to a database: the server is a simple facade that implements each operation by accessing the database.)

Our first step is to select the Slice types we will use for the key and value types for our maps. For each file, we need to store the name of the file, its parent directory, and the contents of the file. For directories, we also store the name and parent directory, as well as a dictionary that keeps track of the subdirectories and files in that directory. This leads to Slice definitions (in file `FilesystemDB.ice`) as follows:

```
#include <Filesystem.ice>
#include <Ice/Identity.ice>

module FilesystemDB {
    struct FileEntry {
        string name;
        Ice::Identity parent;
        Filesystem::Lines text;
    };
```

```
    dictionary<string, Filesystem::NodeDesc> StringNodeDescDict;

    struct DirectoryEntry {
        string name;
        Ice::Identity parent;
        StringNodeDescDict nodes;
    };
};
```

Note that the definitions are placed into a separate module, so they do not affect the existing definitions of the non-persistent version of the application. For reference, here is the definition of `NodeDesc` once more:

```
module Filesystem {
    // ...

    enum NodeType { DirType, FileType };

    struct NodeDesc {
        string name;
        NodeType type;
        Node* proxy;
    };

    // ...
};
```

To store the persistent state for the file system, we use two Freeze maps: one map for files and one map for directories. For files, we map the identity of the file to its corresponding `FileEntry` structure and, similarly, for directories, we map the identity of the directory to its corresponding `DirectoryEntry` structure.

When a request arrives from a client, the object identity is available in the server. The server uses the identity to retrieve the state of the target node for the request from the database and act on that state accordingly.

39.6.2 Implementing the File System Server in C++

In this section, we present a C++ implementation of the file system as outlined in the preceding section. (See Section 39.6.3 for a Java implementation.)

Generating the Maps

Now that we have selected our key and value types, we can generate the maps as follows:

```
$ slice2freeze -I$(ICE_HOME)/slice -I. --ice --dict \
    FilesystemDB::IdentityFileEntryMap,Ice::Identity,\
    FilesystemDB::FileEntry \
    IdentityFileEntryMap FilesystemDB.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
$ slice2freeze -I$(ICE_HOME)/slice -I. --ice --dict \
    FilesystemDB::IdentityDirectoryEntryMap,Ice::Identity,\
    FilesystemDB::DirectoryEntry \
    IdentityDirectoryEntryMap FilesystemDB.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
```

The resulting map classes are named `IdentityFileEntryMap` and `IdentityDirectoryEntryMap`.

The Server `main` Program

The server's `main` program is very simple:

```
#include <FilesystemI.h>
#include <IdentityFileEntryMap.h>
#include <IdentityDirectoryEntryMap.h>
#include <Ice/Application.h>
#include <Freeze/Freeze.h>

using namespace std;
using namespace Filesystem;
using namespace FilesystemDB;

class FilesystemApp : public virtual Ice::Application
{
public:

    FilesystemApp(const string& envName)
        : _envName(envName)
    {
    }

    virtual int run(int, char*[])
    {
        shutdownOnInterrupt();

        Ice::ObjectAdapterPtr adapter =
            communicator()->createObjectAdapter("MapFilesystem");

        const Freeze::ConnectionPtr connection(
            Freeze::createConnection(communicator(), _envName));
```

```
        const IdentityFileEntryMap fileDB(
                                    connection, FileI::filesDB());
        const IdentityDirectoryEntryMap dirDB(
                                    connection,
                                    DirectoryI::directoriesDB());

        adapter->addDefaultServant(
                    new FileI(communicator(), _envName), "file");
        adapter->addDefaultServant(
                    new DirectoryI(communicator(), _envName), "");

        adapter->activate();

        communicator()->waitForShutdown();

        if(interrupted())
            cerr << appName()
                 << ": received signal, shutting down" << endl;

        return 0;
    }

private:

    string _envName;
};

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv, "config.server");
}
```

Let us examine the code in detail. First, we are now including `IdentityFileEntry.h` and `IdentityDirectoryEntry.h`. These header files includes all of the other Freeze (and Ice) header files we need.

Next, we define the class `FilesystemApp` as a subclass of `Ice::Application`, and provide a constructor taking a string argument:

```
    FilesystemApp(const string& envName)
        : _envName(envName) {}
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

The interesting part of `run` are the few lines of code that create the database connection and the two maps that store files and directories, plus the code to add the two default servants:

```
const Freeze::ConnectionPtr connection(
    Freeze::createConnection(communicator(), _envName));

const IdentityFileEntryMap fileDB(
                            connection, FileI::filesDB());
const IdentityDirectoryEntryMap dirDB(
                            connection,
                            DirectoryI::directoriesDB());

adapter->addDefaultServant(
            new FileI(communicator(), _envName), "file");
adapter->addDefaultServant(
            new DirectoryI(communicator(), _envName), "");
```

`run` keeps the database connection open for the duration of the program for performance reasons. As we will see shortly, individual operation implementations will use their own connections; however, it is substantially cheaper to create second (and subsequent connections) than it is to create the first connection.

For the default servants, we use `file` as the category for files. For directories, we use the empty default category.

The Servant Class Definitions

The class definition for `FileI` is very simple:

```
namespace Filesystem {
    class FileI : public File {
    public:
        FileI(const Ice::CommunicatorPtr& communicator,
              const std::string& envName);

        // Slice operations...

        static std::string filesDB();

    private:
        void halt(const Freeze::DatabaseException& ex) const;

        const Ice::CommunicatorPtr _communicator;
        const std::string _envName;
    };
}
```

The `FileI` class stores the communicator and the environment name. These members are initialized by the constructor. The `filesDB` static member function returns the name of the file map, and the `halt` member function is used to stop the server if it encounters a catastrophic error.

The `DirectoryI` class looks very much the same, also storing the communicator and environment name. The `directoriesDB` static member function returns the name of the directory map.

```
namespace Filesystem {
    class DirectoryI : public Directory {
    public:
        DirectoryI(const Ice::CommunicatorPtr& communicator,
                   const std::string& envName);

        // Slice operations...

        static std::string directoriesDB();

    private:
        void halt(const Freeze::DatabaseException& ex) const;

        const Ice::CommunicatorPtr _communicator;
        const std::string _envName;
    };
}
```

Implementing `FileI`

The `FileI` constructor and the `filesDB` and `halt` member functions have trivial implementations:

```
FileI::FileI(const Ice::CommunicatorPtr& communicator,
             const string& envName)
    : _communicator(communicator), _envName(envName)
{
}

string
FileI::filesDB()
{
    return "files";
}

void
FileI::halt(const Freeze::DatabaseException& ex) const
```

```
{
    Ice::Error error(_communicator->getLogger());
    error << "fatal exception: " << ex
          << "\n*** Aborting application ***";

    abort();
}
```

The Slice operations all follow the same implementation strategy: we create a database connection and the file map and place the body of the operation into an infinite loop:

```
string
FileI::someOperation(/* ... */ const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for (;;) {
        try {

            // Operation implementation here...

        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

Each operation creates its own database connection and map for concurrency reasons: the database takes care of all the necessary locking, so there is no need for any other synchronization in the server. If the database detects a deadlock, the code handles the corresponding `DeadlockException` and simply tries again until the operation eventually succeeds; any other database exception indicates that something has gone seriously wrong and terminates the server.

Here is the implementation of the `name` method:

```
string
FileI::name(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());
```

```
    for (;;) {
        try {
            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if (p == fileDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__,
                                                   __LINE__);
            }
            return p->second.name;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

The implementation could hardly be simpler: the default servant uses the identity in the `Current` object to index into the file map. If a record with this identity exists, it returns the name of the file as stored in the `FileEntry` structure in the map. Otherwise, if no such entry exists, it throws `ObjectNotExistException`. This happens if the file existed at some time in the past but has since been destroyed.

The `read` implementation is almost identical. It returns the text that is stored by the `FileEntry`:

```
Lines
FileI::read(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for (;;) {
        try {
            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if (p == fileDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__,
                                                   __LINE__);
            }
            return p->second.text;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
```

```
            halt(ex);
        }
    }
}
```

The `write` implementation updates the file contents and calls `set` on the iterator to update the map with the new contents:

```
void
FileI::write(const Filesystem::Lines& text, const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());

    for (;;) {
        try {
            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if (p == fileDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__,
                                                   __LINE__);
            }
            FileEntry entry = p->second;
            entry.text = text;
            p.set(entry);
            break;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

Finally, the `destroy` implementation for files must update two maps: it needs to remove its own entry in the file map as well as update the `nodes` map in the parent to remove itself from the parent's map of children. This raises a potential problem: if one update succeeds but the other one fails, we end up with an inconsistent file system: either the parent still has an entry to a non-existent file, or the parent lacks an entry to a file that still exists.

To make sure that the two updates happen atomically, `destroy` performs them in a transaction:

```
void
FileI::destroy(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityFileEntryMap fileDB(connection, filesDB());
    IdentityDirectoryEntryMap dirDB(connection,
                                    DirectoryI::directoriesDB());

    for (;;) {
        try {
            Freeze::TransactionHolder txn(connection);

            IdentityFileEntryMap::iterator p = fileDB.find(c.id);
            if (p == fileDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__,
                                                   __LINE__);
            }
            FileEntry entry = p->second;

            IdentityDirectoryEntryMap::iterator pp =
                dirDB.find(entry.parent);
            if (pp == dirDB.end()) {
                halt(Freeze::DatabaseException(
                       __FILE__, __LINE__,
                       "consistency error: file without parent"));
            }

            DirectoryEntry dirEntry = pp->second;
            dirEntry.nodes.erase(entry.name);
            pp.set(dirEntry);

            fileDB.erase(p);
            txn.commit();
            break;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

As you can see, the code first establishes a transaction and then locates the file in the parent directory's map of nodes. After removing the file from the parent, the

code updates the parent's persistent state by calling `set` on the parent iterator and then removes the file from the file map before committing the transaction.

Implementing `DirectoryI`

The `DirectoryI::directoriesDB` implementation returns the string `directories`, and the `halt` implementation is the same as for `FileI`, so we do not show them here.

Turning to the constructor, we must cater for two different scenarios:

- The server is started with a database that already contains a number of nodes.
- The server is started for the very first time with an empty database.

This means that the root directory (which must always exist) may or may not be present in the database. Accordingly, the constructor looks for the root directory (with the fixed identity `RootDir`); if the root directory does not exist in the database, it creates it:

```
DirectoryI::DirectoryI(const Ice::CommunicatorPtr& communicator,
                       const string& envName)
    : _communicator(communicator), _envName(envName)
{
    const Freeze::ConnectionPtr connection =
        Freeze::createConnection(_communicator, _envName);
    IdentityDirectoryEntryMap dirDB(connection, directoriesDB());

    for (;;) {
        try {
            Ice::Identity rootId;
            rootId.name = "RootDir";
            IdentityDirectoryEntryMap::const_iterator p =
                dirDB.find(rootId);
            if (p == dirDB.end()) {
                DirectoryEntry d;
                d.name = "/";
                dirDB.put(make_pair(rootId, d));
            }
            break;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

Next, let us examine the implementation of `createDirectory`. Similar to the `FileI::destroy` operation, `createDirectory` must update both the parent's nodes map and create a new entry in the directory map. These updates must happen atomically, so we perform them in a separate transaction:

```
DirectoryPrx
DirectoryI::createDirectory(const string& name,
                            const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityDirectoryEntryMap directoryDB(connection,
                                          directoriesDB());

    for (;;) {
        try {
            Freeze::TransactionHolder txn(connection);

            IdentityDirectoryEntryMap::iterator p =
                directoryDB.find(c.id);
            if (p == directoryDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__,
                                                   __LINE__);
            }

            DirectoryEntry entry = p->second;
            if (name.empty()
               || entry.nodes.find(name) != entry.nodes.end()) {
                throw NameInUse(name);
            }

            DirectoryEntry d;
            d.name = name;
            d.parent = c.id;

            Ice::Identity id;
            id.name = IceUtil::generateUUID();
            DirectoryPrx proxy = DirectoryPrx::uncheckedCast(
                                    c.adapter->createProxy(id));

            NodeDesc nd;
            nd.name = name;
            nd.type = DirType;
            nd.proxy = proxy;
            entry.nodes.insert(make_pair(name, nd));
```

```
            p.set(entry);
            directoryDB.put(make_pair(id, d));

            txn.commit();

            return proxy;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

After establishing the transaction, the code ensures that the directory does not already contain an entry with the same name and then initializes a new `DirectoryEntry`, setting the name to the name of the new directory, and the parent to its own identity. The identity of the new directory is a UUID, which ensures that all directories have unique identities. In addition, the UUID prevents the "accidental rebirth" of a file or directory in the future (see Section 34.8).

The code then initializes a new `NodeDesc` structure with the details of the new directory and, finally, updates its own map of children as well as adding the new directory to the map of directories before committing the transaction.

The `createFile` implementation is almost identical, so we do not show it here. Similarly, the `name` and `destroy` implementations are almost identical to the ones for `FileI`, so let us move to `list`:

```
NodeDescSeq
DirectoryI::list(const Ice::Current& c)
{
    const Freeze::ConnectionPtr connection(
        Freeze::createConnection(_communicator, _envName));
    IdentityDirectoryEntryMap directoryDB(connection,
                                          directoriesDB());

    for (;;) {
        try {
            IdentityDirectoryEntryMap::iterator p =
                directoryDB.find(c.id);
            if (p == directoryDB.end()) {
                throw Ice::ObjectNotExistException(__FILE__,
                                                   __LINE__);
            }
```

```
            NodeDescSeq result;
            for (StringNodeDescDict::const_iterator q =
                                        p->second.nodes.begin();
                 q != p->second.nodes.end();
                 ++q) {
                result.push_back(q->second);
            }
            return result;
        } catch (const Freeze::DeadlockException&) {
            continue;
        } catch (const Freeze::DatabaseException& ex) {
            halt(ex);
        }
    }
}
```

Again, the code is very simple: it iterates over the `nodes` map, adding each `NodeDesc` structure to the returned sequence.

The `find` implementation is even simpler, so we do not show it here.

39.6.3 Implementing the File System Server in Java

In this section, we present a Java implementation of the file system as outlined in Section 39.6.1. (See Section 39.6.2 for a C++ implementation.)

Generating the Maps

Now that we have selected our key and value types, we can generate the maps as follows:

```
$ slice2freezej -I$(ICE_HOME)/slice -I. --ice --dict \
    FilesystemDB.IdentityFileEntryMap,Ice.Identity,\
    FilesystemDB.FileEntry \
    IdentityFileEntryMap FilesystemDB.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
$ slice2freezej -I$(ICE_HOME)/slice -I. --ice --dict \
    FilesystemDB.IdentityDirectoryEntryMap,Ice.Identity,\
    FilesystemDB.DirectoryEntry \
    IdentityDirectoryEntryMap FilesystemDB.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
```

The resulting map classes are named `IdentityFileEntryMap` and `IdentityDirectoryEntryMap`.

The Server Main Program

The server's main program is very simple:

```
import Filesystem.*;
import FilesystemDB.*;

public class Server extends Ice.Application
{
    public
    Server(String envName)
    {
        _envName = envName;
    }

    public int
    run(String[] args)
    {
        Ice.ObjectAdapter adapter =
            communicator().createObjectAdapter("MapFilesystem");

        Freeze.Connection connection = null;
        try {
            connection =
                Freeze.Util.createConnection(communicator(),
                                             _envName);
            IdentityFileEntryMap fileDB =
                new IdentityFileEntryMap(
                    connection, FileI.filesDB(), true);
            IdentityDirectoryEntryMap dirDB =
                new IdentityDirectoryEntryMap(
                    connection, DirectoryI.directoriesDB(), true);

            adapter.addDefaultServant(
                new FileI(communicator(), _envName), "file");
            adapter.addDefaultServant(
                new DirectoryI(communicator(), _envName), "");

            adapter.activate();

            communicator().waitForShutdown();
        } finally {
            connection.close();
        }

        return 0;
```

```
    }

    public static void
    main(String[] args)
    {
        Server app = new Server("db");
        app.main("MapServer", args, "config.server");
        System.exit(0);
    }

    private String _envName;
}
```

First, we import the `Filesystem` and `FilesystemDB` packages.

Next, we define the class `FilesystemApp` as a subclass of `Ice.Application`, and provide a constructor taking a string argument:

```
FilesystemApp(const string& envName)
    : _envName(envName) {}
```

The string argument represents the name of the database environment, and is saved for later use in `run`.

The interesting part of `run` are the few lines of code that create the database connection and the two maps that store files and directories, plus the code to add the two default servants:

```
connection =
    Freeze.Util.createConnection(communicator(),
                                 _envName);
IdentityFileEntryMap fileDB =
    new IdentityFileEntryMap(
        connection, FileI.filesDB(), true);
IdentityDirectoryEntryMap dirDB =
    new IdentityDirectoryEntryMap(
        connection, DirectoryI.directoriesDB(), true);

adapter.addDefaultServant(
    new FileI(communicator(), _envName), "file");
adapter.addDefaultServant(
    new DirectoryI(communicator(), _envName), "");
```

`run` keeps the database connection open for the duration of the program for performance reasons. As we will see shortly, individual operation implementations will use their own connections; however, it is substantially cheaper to create second (and subsequent connections) than it is to create the first connection.

For the default servants, we use `file` as the category for files. For directories, we use the empty default category.

Implementing `FileI`

The class definition for `FileI` is very simple:

```
public class FileI extends _FileDisp
{
    public
    FileI(Ice.Communicator communicator,
                String envName)
    {
        _communicator = communicator;
        _envName = envName;
    }

    // Slice operations...

    public static String
    filesDB()
    {
        return "files";
    }

    private void
    halt(Freeze.DatabaseException e)
    {
        java.io.StringWriter sw = new java.io.StringWriter();
        java.io.PrintWriter pw = new java.io.PrintWriter(sw);
        e.printStackTrace(pw);
        pw.flush();
        _communicator.getLogger().error(
            "fatal database error\n" + sw.toString() +
            "\n*** Halting JVM ***");
        Runtime.getRuntime().halt(1);
    }

    private Ice.Communicator _communicator;
    private String _envName;
}
```

The `FileI` class stores the communicator and the environment name. These members are initialized by the constructor. The `filesDB` static method returns the name of the file map, and the `halt` member function is used to stop the server if it encounters a catastrophic error.

The Slice operations all follow the same implementation strategy: we create a database connection and the file map and place the body of the operation into an infinite loop:

```
public String
someOperation(/* ... */ Ice.Current c)
{
    Freeze.Connection connection =
        Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityFileEntryMap fileDB =
            new IdentityFileEntryMap(connection, filesDB());

        for (;;) {
            try {

                // Operation implementation here...

            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            }
        }
    } finally {
        connection.close();
    }
}
```

Each operation creates its own database connection and map for concurrency reasons: the database takes care of all the necessary locking, so there is no need for any other synchronization in the server. If the database detects a deadlock, the code handles the corresponding `DeadlockException` and simply tries again until the operation eventually succeeds; any other database exception indicates that something has gone seriously wrong and terminates the server.

Here is the implementation of the `name` method:

```
public String
name(Ice.Current c)
{
    Freeze.Connection connection =
        Freeze.Util.createConnection(_communicator, _envName);
    try
    {
        IdentityFileEntryMap fileDB =
```

```
                new IdentityFileEntryMap(connection, filesDB());

            for (;;) {
                try {
                    FileEntry entry = fileDB.get(c.id);
                    if (entry == null) {
                        throw new Ice.ObjectNotExistException();
                    }
                    return entry.name;
                } catch (Freeze.DeadlockException ex) {
                    continue;
                } catch (Freeze.DatabaseException ex) {
                    halt(ex);
                }
            }
        } finally {
            connection.close();
        }
    }
```

The implementation could hardly be simpler: the default servant uses the identity in the `Current` object to index into the file map. If a record with this identity exists, it returns the name of the file as stored in the `FileEntry` structure in the map. Otherwise, if no such entry exists, it throws `ObjectNotExistException`. This happens if the file existed at some time in the past but has since been destroyed.

The `read` implementation is almost identical. It returns the text that is stored by the `FileEntry`:

```
    public String[]
    read(Ice.Current c)
    {
        Freeze.Connection connection =
            Freeze.Util.createConnection(_communicator, _envName);
        try {
            IdentityFileEntryMap fileDB = new
                IdentityFileEntryMap(connection, filesDB());

            for (;;) {
                try {
                    FileEntry entry = fileDB.get(c.id);
                    if (entry == null) {
                        throw new Ice.ObjectNotExistException();
                    }
                    return entry.text;
```

```
                } catch (Freeze.DeadlockException ex) {
                    continue;
                } catch (Freeze.DatabaseException ex) {
                    halt(ex);
                }
            }
        } finally {
            connection.close();
        }
    }
```

The `write` implementation updates the file contents and calls `put` on the iterator to update the map with the new contents:

```
public void
write(String[] text, Ice.Current c)
    throws GenericError
{
    Freeze.Connection connection =
        Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityFileEntryMap fileDB =
            new IdentityFileEntryMap(connection, filesDB());

        for (;;) {
            try {
                FileEntry entry = fileDB.get(c.id);
                if (entry == null) {
                    throw new Ice.ObjectNotExistException();
                }
                entry.text = text;
                fileDB.put(c.id, entry);
                break;
            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            }
        }
    } finally {
        connection.close();
    }
}
```

Finally, the `destroy` implementation for files must update two maps: it needs to remove its own entry in the file map as well as update the `nodes` map in the

parent to remove itself from the parent's map of children. This raises a potential problem: if one update succeeds but the other one fails, we end up with an inconsistent file system: either the parent still has an entry to a non-existent file, or the parent lacks an entry to a file that still exists.

To make sure that the two updates happen atomically, `destroy` performs them in a transaction:

```
public void
destroy(Ice.Current c)
    throws PermissionDenied
{
    Freeze.Connection connection =
        Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityFileEntryMap fileDB =
            new IdentityFileEntryMap(connection, filesDB());
        IdentityDirectoryEntryMap dirDB =
            new IdentityDirectoryEntryMap(
                connection, DirectoryI.directoriesDB());

        for (;;) {
            Freeze.Transaction txn = null;
            try {
                txn = connection.beginTransaction();

                FileEntry entry = fileDB.get(c.id);
                if (entry == null) {
                    throw new Ice.ObjectNotExistException();
                }

                DirectoryEntry dirEntry =
                    (DirectoryEntry)dirDB.get(entry.parent);
                if (dirEntry == null) {
                    halt(new Freeze.DatabaseException(
                                    "consistency error: " +
                                    "file without parent"));
                }

                dirEntry.nodes.remove(entry.name);
                dirDB.put(entry.parent, dirEntry);

                fileDB.remove(c.id);

                txn.commit();
                txn = null;
```

```
                    break;
                } catch (Freeze.DeadlockException ex) {
                    continue;
                } catch (Freeze.DatabaseException ex) {
                    halt(ex);
                } finally {
                    if (txn != null) {
                        txn.rollback();
                    }
                }
            }
        }
        finally
        {
            connection.close();
        }
    }
```

As you can see, the code first establishes a transaction and then locates the file in the parent directory's map of nodes. After removing the file from the parent, the code updates the parent's persistent state by calling `put` on the parent iterator and then removes the file from the file map before committing the transaction.

Implementing `DirectoryI`

The `DirectoryI.directoriesDB` implementation returns the string `directories`, and the `halt` implementation is the same as for `FileI`, so we do not show them here.

Turning to the constructor, we must cater for two different scenarios:

- The server is started with a database that already contains a number of nodes.
- The server is started for the very first time with an empty database.

This means that the root directory (which must always exist) may or may not be present in the database. Accordingly, the constructor looks for the root directory (with the fixed identity `RootDir`); if the root directory does not exist in the database, it creates it:

```
public
DirectoryI(Ice.Communicator communicator, String envName)
{
    _communicator = communicator;
    _envName = envName;

    Freeze.Connection connection =
        Freeze.Util.createConnection(_communicator, _envName);
```

```
        try {
            IdentityDirectoryEntryMap dirDB =
                new IdentityDirectoryEntryMap(connection,
                                              directoriesDB());

            for (;;) {
                try {
                    Ice.Identity rootId =
                        new Ice.Identity("RootDir", "");
                    DirectoryEntry entry = dirDB.get(rootId);
                    if (entry == null) {
                        dirDB.put(rootId,
                            new DirectoryEntry("/",
                                new Ice.Identity("", ""), null));
                    }
                    break;
                } catch (Freeze.DeadlockException ex) {
                    continue;
                } catch (Freeze.DatabaseException ex) {
                    halt(ex);
                }
            }
        } finally {
            connection.close();
        }
    }
```

Next, let us examine the implementation of `createDirectory`. Similar to the `FileI::destroy` operation, `createDirectory` must update both the parent's nodes map and create a new entry in the directory map. These updates must happen atomically, so we perform them in a separate transaction:

```
    public DirectoryPrx
    createDirectory(String name, Ice.Current c)
        throws NameInUse
    {
        Freeze.Connection connection =
            Freeze.Util.createConnection(_communicator, _envName);
        try {
            IdentityDirectoryEntryMap dirDB =
                new IdentityDirectoryEntryMap(connection,
                                              directoriesDB());

            for (;;) {
                Freeze.Transaction txn = null;
                try {
```

```
                    txn = connection.beginTransaction();

                    DirectoryEntry entry = dirDB.get(c.id);
                    if (entry == null) {
                        throw new Ice.ObjectNotExistException();
                    }
                    if(name.length() == 0
                       || entry.nodes.get(name) != null) {
                        throw new NameInUse(name);
                    }

                    DirectoryEntry newEntry =
                        new DirectoryEntry(name, c.id, null);
                    Ice.Identity id = new Ice.Identity(
                        java.util.UUID.randomUUID().toString(),
                        "");
                    DirectoryPrx proxy =
                        DirectoryPrxHelper.uncheckedCast(
                                    c.adapter.createProxy(id));

                    entry.nodes.put(name,
                                    new NodeDesc(name,
                                                 NodeType.DirType,
                                                 proxy));
                    dirDB.put(c.id, entry);
                    dirDB.put(id, newEntry);

                    txn.commit();
                    txn = null;

                    return proxy;
                } catch (Freeze.DeadlockException ex) {
                    continue;
                } catch (Freeze.DatabaseException ex) {
                    halt(ex);
                } finally {
                    if(txn != null) {
                        txn.rollback();
                    }
                }
            }
        } finally {
            connection.close();
        }
    }
```

After establishing the transaction, the code ensures that the directory does not already contain an entry with the same name and then initializes a new `DirectoryEntry`, setting the name to the name of the new directory, and the parent to its own identity. The identity of the new directory is a UUID, which ensures that all directories have unique identities. In addition, the UUID prevents the "accidental rebirth" of a file or directory in the future (see Section 34.8).

The code then initializes a new `NodeDesc` structure with the details of the new directory and, finally, updates its own map of children as well as adding the new directory to the map of directories before committing the transaction.

The `createFile` implementation is almost identical, so we do not show it here. Similarly, the `name` and `destroy` implementations are almost identical to the ones for `FileI`, so let us move to `list`:

```
public NodeDesc[]
list(Ice.Current c)
{
    Freeze.Connection connection =
        Freeze.Util.createConnection(_communicator, _envName);
    try {
        IdentityDirectoryEntryMap dirDB =
            new IdentityDirectoryEntryMap(connection,
                                          directoriesDB());

        for (;;) {
            try {
                DirectoryEntry entry = dirDB.get(c.id);
                if (entry == null) {
                    throw new Ice.ObjectNotExistException();
                }
                NodeDesc[] result =
                    new NodeDesc[entry.nodes.size()];
                java.util.Iterator<NodeDesc> p =
                    entry.nodes.values().iterator();
                for (int i = 0; i < entry.nodes.size(); ++i) {
                    result[i] = p.next();
                }
                return result;
            } catch (Freeze.DeadlockException ex) {
                continue;
            } catch (Freeze.DatabaseException ex) {
                halt(ex);
            }
        }
```

```
        } finally {
            connection.close();
        }
    }
```

Again, the code is very simple: it iterates over the `nodes` map, adding each `NodeDesc` structure to the returned sequence.

The `find` implementation is even simpler, so we do not show it here.

39.7 The Freeze Catalog

In each database environment, Freeze maintains an internal table that contains type information describing all the databases in the environment. This table is an instance of a Freeze map in which the key is a string representing the database name and the value is an instance of `Freeze::CatalogData`:

```
module Freeze {
    struct CatalogData {
        bool evictor;
        string key;
        string value;
    };
};
```

An entry describes an evictor database if the `evictor` member is true, in which case the `key` and `value` members are empty strings. An entry that describes a Freeze map sets `evictor` to false; the `key` and `value` members contain the Slice types used when the map was defined.

Tools such as **`transformdb`** and **`dumpdb`** (see Chapter 40) access the catalog to obtain type information when none is supplied by the user. You can also use **`dumpdb`** to display the catalog of a database environment.

Freeze applications may access the catalog in the same manner as any other Freeze map. For example, the following C++ code displays the contents of a catalog:

```
#include <Freeze/Catalog.h>
...
string envName = ...;
Freeze::ConnectionPtr conn =
    Freeze::createConnection(communicator, envName);
Freeze::Catalog catalog(conn, Freeze::catalogName());
for (Freeze::Catalog::const_iterator p = catalog.begin();
```

```
    p != catalog.end(); ++p) {
    if (p->second.evictor)
        cout << p->first << ": evictor" << endl;
    else
        cout << p->first << ": map<" << p->second.key
             << ", " << p->second.value << ">" << endl;
}
conn->close();
```

The equivalent Java code is shown below:

```
String envName = ...;
Freeze.Connection conn =
    Freeze.Util.createConnection(communicator, envName);
Freeze.Catalog catalog =
    new Freeze.Catalog(conn, Freeze.Util.catalogName(), true);
for (java.util.Map.Entry<String, Freeze.CatalogData> e :
     catalog.entrySet()) {
    String name = e.getKey();
    Freeze.CatalogData data = e.getValue();
    if (data.evictor)
        System.out.println(name + ": evictor");
    else
        System.out.println(name + ": map<" + data.key + ", " +
                           data.value + ">");
}
conn.close();
```

39.8 Backups

When you store important information in a Freeze database environment, you should consider regularly backing up the database environment.

There are two forms of backups: cold backups, where you just copy your database environment directory while no application is using these files (very straightforward), and hot backups, where you backup a database environment while an application is actively reading and writing data.

In order to perform a hot backup on a Freeze environment, you need to configure this Freeze environment with two non-default settings:

- `Freeze.DbEnv.`*`envName`*`.OldLogsAutoDelete=0`
  This instructs Freeze to keep old log files instead of periodically deleting them. This setting is necessary for proper hot backups; it implies that you will

need to take care of deleting old files yourself (typically as part of your periodic backup procedure).

- `Freeze.DbEnv.`*`envName`*`.DbPrivate=0`
  By default, Freeze is configured with `DbPrivate` set to 1, which means only one process at a time can safely access the database environment. When performing hot backups, you need to access this database environment concurrently from various Berkeley DB utilities (such as **`db_archive`** or **`db_hotbackup`**), so you need to set this property to 0.

The `Freeze/backup` C++ demo in your Ice distribution shows one way to perform such backups and recovery. Please consult the Berkeley DB documentation for further details.

39.9 Summary

Freeze is a collection of services that simplify the use of persistence in Ice applications. The Freeze map is an associative container mapping any Slice key and value types, providing a convenient and familiar interface to a persistent map. Freeze evictors are an especially powerful facility for supporting persistent Ice objects in a highly-scalable implementation.

Chapter 40
FreezeScript

40.1 Chapter Overview

This chapter describes the FreezeScript tools for migrating and inspecting the databases created by Freeze maps and evictors. The discussion of database migration begins in Section 40.3 and continues through Section 40.5. Database inspection is presented in Section 40.6 and Section 40.7. Finally, Section 40.8 describes the expression language supported by the FreezeScript tools.

40.2 Introduction

As described in Chapter 39, Freeze supplies a valuable set of services for simplifying the use of persistence in Ice applications. However, while Freeze makes it easy for an application to manage its persistent state, there are additional administrative responsibilities that must also be addressed:

- Migration

  As an application evolves, it is not unusual for the types describing its persistent state to evolve as well. When these changes occur, a great deal of time can be saved if existing databases can be migrated to the new format while preserving as much information as possible.

- Inspection

  The ability to examine a database can be helpful during every stage of the application's lifecycle, from development to deployment.

FreezeScript provides tools for performing both of these activities on Freeze map and evictor databases. These databases have a well-defined structure because the key and value of each record consist of the marshaled bytes of their respective Slice types. This design allows the FreezeScript tools to operate on any Freeze database using only the Slice definitions for the database types.

40.3 Database Migration

The FreezeScript tool **`transformdb`** migrates a database created by a Freeze map or evictor. It accomplishes this by comparing the "old" Slice definitions (i.e., the ones that describe the current contents of the database) with the "new" Slice definitions, and making whatever modifications are necessary to ensure that the transformed database is compatible with the new definitions.

This would be difficult to achieve by writing a custom transformation program because that program would require static knowledge of the old and new types, which frequently define many of the same symbols and would therefore prevent the program from being loaded. The **`transformdb`** tool avoids this issue using an interpretive approach: the Slice definitions are parsed and used to drive the migration of the database records.

The tool supports two modes of operation:

1. automatic migration, in which the database is migrated in a single step using only the default set of transformations, and
2. custom migration, in which you supply a script to augment or override the default transformations.

40.3.1 Default Transformations

The default transformations performed by **`transformdb`** preserve as much information as possible. However, there are practical limits to the tool's capabilities, since the only information it has is obtained by performing a comparison of the Slice definitions.

For example, suppose our old definition for a structure is the following:

```
struct AStruct {
    int i;
};
```

We want to migrate instances of this struct to the following revised definition:

```
struct AStruct {
    int j;
};
```

As the developers, we know that the `int` member has been renamed from `i` to `j`, but to **`transformdb`** it appears that member `i` was removed and member `j` was added. The default transformation results in exactly that behavior: the value of `i` is lost, and `j` is initialized to a default value. If we need to preserve the value of `i` and transfer it to `j`, then we need to use custom migration (see Section 40.3.5).

The changes that occur as a type system evolves can be grouped into three categories:

- Data members

  The data members of class and structure types are added, removed, or renamed. As discussed above, the default transformations initialize new and renamed data members to default values (see Section 40.3.3).

- Type names

  Types are added, removed, or renamed. New types do not pose a problem for database migration when used to define a new data member; the member is initialized with default values as usual. On the other hand, if the new type replaces the type of an existing data member, then type compatibility becomes a factor (see the following item).

  Removed types generally do not cause problems either, because any uses of that type must have been removed from the new Slice definitions (e.g., by removing data members of that type). There is one case, however, where removed types become an issue, and that is for polymorphic classes (see Section 40.5.10).

  Renamed types are a concern, just like renamed data members, because of the potential for losing information during migration. This is another situation for which custom migration is recommended.

- Type content

  Examples of changes of type content include the key type of a dictionary, the element type of a sequence, or the type of a data member. If the old and new types are not compatible (as defined in Section 40.3.2), then the default trans-

formation emits a warning, discards the current value, and reinitializes the value as described in Section 40.3.3.

40.3.2 Type Compatibility

Changes in the type of a value are restricted to certain sets of compatible changes. This section describes the type changes supported by the default transformations. All incompatible type changes result in a warning indicating that the current value is being discarded and a default value for the new type assigned in its place. Additional flexibility is provided by custom migration, as described in Section 40.3.5.

Boolean

A value of type `bool` can be transformed to and from `string`. The legal string values for a `bool` value are `"true"` and `"false"`.

Integer

The integer types `byte`, `short`, `int`, and `long` can be transformed into each other, but only if the current value is within range of the new type. These integer types can also be transformed into `string`.

Floating Point

The floating-point types `float` and `double` can be transformed into each other, as well as to `string`. No attempt is made to detect a loss of precision during transformation.

String

A `string` value can be transformed into any of the primitive types, as well as into enumeration and proxy types, but only if the value is a legal string representation of the new type. For example, the string value `"Pear"` can be transformed into the enumeration `Fruit`, but only if `Pear` is an enumerator of `Fruit`.

Enum

An enumeration can be transformed into an enumeration with the same type id, or into a string. Transformation between enumerations is performed symbolically. For example, consider our old type below:

```
enum Fruit { Apple, Orange, Pear };
```

Suppose the enumerator `Pear` is being transformed into the following new type:

```
enum Fruit { Apple, Pear };
```

The transformed value in the new enumeration is also `Pear`, despite the fact that `Pear` has changed positions in the new type. However, if the old value had been `Orange`, then the default transformation emits a warning because that enumerator no longer exists, and initializes the new value to `Apple` (the default value).

If an enumerator has been renamed, then custom migration is required to convert enumerators from the old name to the new one.

Sequence

A sequence can be transformed into another sequence type, even if the new sequence type does not have the same type id as the old type, but only if the element types are compatible. For example, `sequence<short>` can be transformed into `sequence<int>`, regardless of the names given to the sequence types.

Dictionary

A dictionary can be transformed into another dictionary type, even if the new dictionary type does not have the same type id as the old type, but only if the key and value types are compatible. For example, `dictionary<int, string>` can be transformed into `dictionary<long, string>`, regardless of the names given to the dictionary types.

Caution is required when changing the key type of a dictionary, because the default transformation of keys could result in duplication. For example, if the key type changes from `int` to `short`, any `int` value outside the range of `short` results in the key being initialized to a default value (namely zero). If zero is already used as a key in the dictionary, or another out-of-range key is encountered, then a duplication occurs. The transformation handles key duplication by removing the duplicate element from the transformed dictionary. (Custom migration can be useful in these situations if the default behavior is not acceptable.)

Structure

A `struct` type can only be transformed into another `struct` type with the same type id. Data members are transformed as appropriate for their types.

Proxy

A proxy value can be transformed into another proxy type, or into `string`. Transformation into another proxy type is done with the same semantics as in a

language mapping: if the new type does not match the old type, then the new type must be a base type of the old type (that is, the proxy is widened).

Class

A `class` type can only be transformed into another `class` type with the same type id. A data member of a `class` type is allowed to be widened to a base type. Data members are transformed as appropriate for their types. See Section 40.5.10 for more information on transforming classes.

40.3.3 Default Values

Data types are initialized with default values, as shown in Table 40.1.

Table 40.1. Default values for Slice types.

Type	Default Value
Boolean	`false`
Numeric	Zero (`0`)
String	Empty string
Enumeration	The first enumerator
Sequence	Empty sequence
Dictionary	Empty dictionary
Struct	Data members are initialized recursively
Proxy	Nil
Class	Nil

40.3.4 Running an Automatic Transformation

In order to use automatic transformation, we need to supply the following information to **`transformdb`**:

- The old and new Slice definitions
- The old and new types for the database key and value
- The database environment directory, the database file name, and the name of a new database environment directory to hold the transformed database

Here is an example of a **`transformdb`** command:

```
$ transformdb --old old/MyApp.ice --new new/MyApp.ice \
--key int,string --value ::Employee db emp.db newdb
```

Briefly, the **`--old`** and **`--new`** options specify the old and new Slice definitions, respectively. These options can be specified as many times as necessary in order to load all of the relevant definitions. The **`--key`** option indicates that the database key is evolving from `int` to `string`. The **`--value`** option specifies that `::Employee` is used as the database value type in both old and new type definitions, and therefore only needs to be specified once. Finally, we provide the pathname of the database environment directory (**`db`**), the file name of the database (**`emp.db`**), and the pathname of the database environment directory for the transformed database (**`newdb`**).

See Section 40.5 for more information on using **`transformdb`**.

40.3.5 Custom Migration

Custom migration is useful when your types have changed in ways that make automatic migration difficult or impossible. It is also convenient to use custom migration when you have complex initialization requirements for new types or new data members, because custom migration enables you to perform many of the same tasks that would otherwise require you to write a throwaway program.

Custom migration operates in conjunction with automatic migration, allowing you to inject your own transformation rules at well-defined intercept points in the automatic migration process. These rules are called *transformation descriptors*, and are written in XML.

A Simple Example

We can use a simple example to demonstrate the utility of custom migration. Suppose our application uses a Freeze map whose type is `string` and whose value is an enumeration, defined as follows:

```
enum BigThree { Ford, DaimlerChrysler, GeneralMotors };
```

We now wish to rename the enumerator `DaimlerChrysler`, as shown in our new definition:

```
enum BigThree { Ford, Daimler, GeneralMotors };
```

As explained in Section 40.3.2, the default transformation results in all occurrences of the `DaimlerChrysler` enumerator being transformed into `Ford`, because

`Chrysler` no longer exists in the new definition and therefore the default value `Ford` is used instead.

To remedy this situation, we use the following transformation descriptors:

```
<transformdb>
    <database key="string" value="::BigThree">
        <record>
            <if test="oldvalue == ::Old::DaimlerChrysler>
                <set target="newvalue"
                    value="::New::Daimler"/>
            </if>
        </record>
    </database>
</transformdb>
```

When executed, these descriptors convert occurrences of `DaimlerChrysler` in the old type system into `Daimler` in the transformed database's new type system. Transformation descriptors are described in detail in Section 40.4.

40.4 Transformation Descriptors

This section describes the XML elements comprising the FreezeScript transformation descriptors.

40.4.1 Overview

A transformation descriptor file has a well-defined structure. The top-level descriptor in the file is `<transformdb>`. A `<database>` descriptor must be present within `<transformdb>` to define the key and value types used by the database. Inside `<database>`, the `<record>` descriptor triggers the transformation process. See Section 40.3.5 for an example that demonstrates the structure of a minimal descriptor file.

During transformation, type-specific actions are supported by the `<transform>` and `<init>` descriptors, both of which are children of `<transformdb>`. One `<transform>` descriptor and one `<init>` descriptor may be defined for each type in the new Slice definitions. Each time **`transformdb`** creates a new instance of a type, it executes the `<init>` descriptor for that type, if one is defined. Similarly, each time **`transformdb`** transforms an instance of an old type into a new type, the `<transform>` descriptor for the new type is executed.

The `<database>`, `<record>`, `<transform>`, and `<init>` descriptors may contain general-purpose action descriptors such as `<if>`, `<set>`, and `<echo>`. These actions resemble statements in programming languages like C++ and Java, in that they are executed in the order of definition and their effects are cumulative. Actions make use of the expression language described in Section 40.8.

40.4.2 Flow of Execution

The transformation descriptors are executed as described below.

- `<database>` is executed first. Each child descriptor of `<database>` is executed in the order of definition. If a `<record>` descriptor is present, database transformation occurs at that point. Any child descriptors of `<database>` that follow `<record>` are not executed until transformation completes.
- During transformation of each record, **`transformdb`** creates instances of the new key and value types, which includes the execution of the `<init>` descriptors for those types. Next, the old key and value are transformed into the new key and value, in the following manner:
  1. Locate the `<transform>` descriptor for the type.
  2. If no descriptor is found, or the descriptor exists and it does not preclude default transformation, then transform the data as described in Section 40.3.1.
  3. If the `<transform>` descriptor exists, execute it.
  4. Finally, execute the child descriptors of `<record>`.

See Section 40.4.4 for detailed information on the transformation descriptors.

40.4.3 Scopes

The `<database>` descriptor creates a global scope, allowing child descriptors of `<database>` to define symbols that are accessible in any descriptor[1]. Furthermore, certain other descriptors create local scopes that exist only for the duration of the descriptor's execution. For example, the `<transform>` descriptor creates

1. In order for a global symbol to be available to a `<transform>` or `<init>` descriptor, the symbol must be defined before the `<record>` descriptor is executed.

a local scope and defines the symbols `old` and `new` to represent a value in its old and new forms. Child descriptors of `<transform>` can also define new symbols in the local scope, as long as those symbols do not clash with an existing symbol in that scope. It is legal to add a new symbol with the same name as a symbol in an outer scope, but the outer symbol will not be accessible during the descriptor's execution.

The global scope is useful in many situations. For example, suppose you want to track the number of times a certain value was encountered during transformation. This can be accomplished as shown below:

```
<transformdb>
    <database key="string" value="::Ice::Identity">
        <define name="categoryCount" type="int" value="0"/>
        <record/>
        <echo message="categoryCount = " value="categoryCount"/>
    </database>
    <transform type="::Ice::Identity">
        <if test="new.category == 'Accounting'">
            <set target="categoryCount"
                value="categoryCount + 1"/>
        </if>
    </transform>
</transformdb>
```

In this example, the `<define>` descriptor introduces the symbol `categoryCount` into the global scope, defining it as type `int` with an initial value of zero. Next, the `<record>` descriptor causes transformation to proceed. Each occurrence of the type `Ice::Identity` causes its `<transform>` descriptor to be executed, which examines the `category` member and increases `categoryCount` if necessary. Finally, after transformation completes, the `<echo>` descriptor displays the final value of `categoryCount`.

To reinforce the relationships between descriptors and scopes, consider the diagram in Figure 40.1. Several descriptors are shown, including the symbols they define in their local scopes. In this example, the `<iterate>` descriptor has a dictionary target and therefore the default symbol for the element value, `value`, hides the symbol of the same name in the parent `<init>` descriptor's scope[2]. In

2. This situation can be avoided by assigning a different symbol name to the element value.

addition to symbols in the `<iterate>` scope, child descriptors of `<iterate>` can also refer to symbols from the `<init>` and `<database>` scopes.

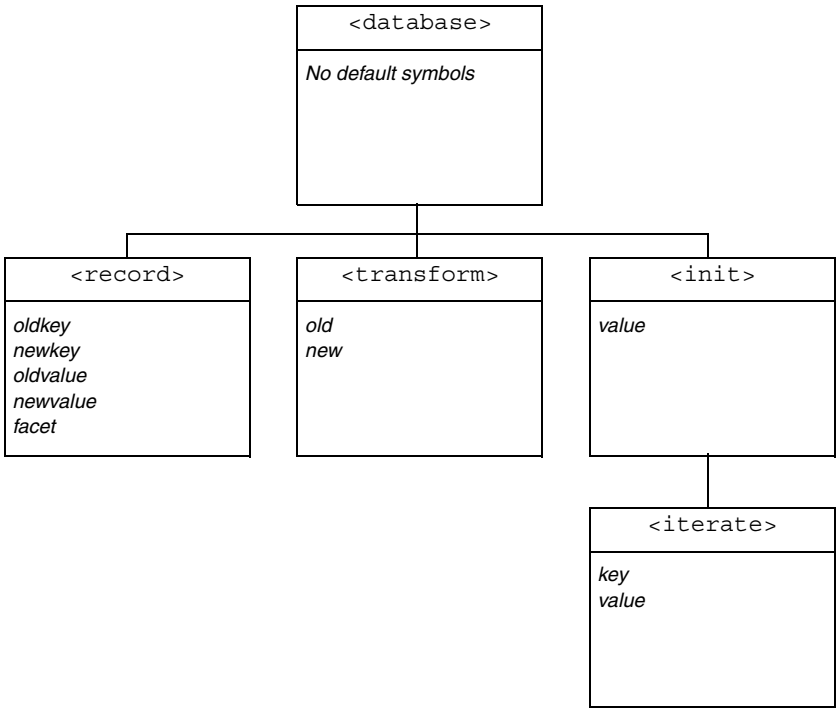


Figure 40.1. Relationship between descriptors and scopes.

40.4.4 Descriptor Reference

`<transformdb>`

The top-level descriptor in a descriptor file. It requires at least one `<database>` descriptor, and supports any number of `<transform>` and `<init>` descriptors. This descriptor has no attributes.

`<database>`

The attributes of this descriptor define the old and new key and value types for the database to be transformed, and optionally the name of the database to which

these types apply. It supports any number of child descriptors, but at most one `<record>` descriptor. The `<database>` descriptor also creates a global scope for user-defined symbols (see Section 40.4.3).

The attributes supported by the `<database>` descriptor are described in Table 40.2.

Table 40.2. Attributes for `<database>` descriptor.

Name	Description
`name`	Specifies the name of the database defined by this descriptor. (Optional)
`key`	Specifies the Slice types of the old and new keys. If the types are the same, only one needs to be specified. Otherwise, the types are separated by a comma.
`value`	Specifies the Slice types of the old and new values. If the types are the same, only one needs to be specified. Otherwise, the types are separated by a comma.

As an example, consider the following `<database>` descriptor. In this case, the Freeze map to be transformed currently has key type `int` and value type `::Employee`, and is migrating to a key type of `string`:

```
<database key="int,string" value="::Employee">
```

`<record>`

Commences the transformation. Child descriptors are executed for each record in the database, providing the user with an opportunity to examine the record's old key and value, and optionally modify the new key and value. Default transformations, as well as `<transform>` and `<init>` descriptors, are executed before the child descriptors. The `<record>` descriptor introduces the following symbols into a local scope: `oldkey`, `newkey`, `oldvalue`, `newvalue`, `facet`. These symbols are accessible to child descriptors, but not to `<transform>` or `<init>` descriptors. The `oldkey` and `oldvalue` symbols are read-only. The `facet` symbol is a string indicating the facet name of the object in the current record, and is only relevant for Freeze evictor databases.

Use caution when modifying database keys to ensure that duplicate keys do not occur. If a duplicate database key is encountered, transformation fails immediately.

Note that database transformation only occurs if a `<record>` descriptor is present.

`<transform>`

Customizes the transformation for all instances of a type in the new Slice definitions. The children of this descriptor are executed after the optional default transformation has been performed, as described in Section 40.3.1. Only one `<transform>` descriptor can be specified for a type, but a `<transform>` descriptor is not required for every type. The symbols `old` and `new` are introduced into a local scope and represent the old and new values, respectively. The `old` symbol is read-only. The attributes supported by this descriptor are described in Table 40.3.

Table 40.3. Attributes for `<transform>` descriptor.

Name	Description
`type`	Specifies the type id in the new Slice definitions.
`default`	If `false`, no default transformation is performed on values of this type. If not specified, the default value is `true`.
`base`	This attribute determines whether `<transform>` descriptors of base class types are executed. If `true`, the `<transform>` descriptor of the immediate base class is invoked. If no descriptor is found for the immediate base class, the class hierarchy is searched until a descriptor is found. The execution of any base class descriptors occurs after execution of this descriptor's children. If not specified, the default value is `true`.
`rename`	Indicates that a type in the old Slice definitions has been renamed to the new type identified by the `type` attribute. The value of this attribute is the type id of the old Slice definition. Specifying this attribute relaxes the strict compatibility rules defined in Section 40.3.2 for `enum`, `struct` and `class` types.

Below is an example of a `<transform>` descriptor that initializes a new data member:

```
<transform type="::Product">
    <set target="new.salePrice"
         value="old.listPrice * old.discount"/>
</transform>
```

For class types, **transformdb** first attempts to locate a `<transform>` descriptor for the object's most-derived type. If no descriptor is found, **transformdb** proceeds up the class hierarchy in an attempt to find a descriptor. The base object type, `Object`, is the root of every class hierarchy and is included in the search for descriptors. It is therefore possible to define a `<transform>` descriptor for type `Object`, which will be invoked for every class instance.

Note that `<transform>` descriptors are executed recursively. For example, consider the following Slice definitions:

```
struct Inner {
    int sum;
};
struct Outer {
    Inner i;
};
```

When **transformdb** is performing the default transformation on a value of type `Outer`, it recursively performs the default transformation on the `Inner` member, then executes the `<transform>` descriptor for `Inner`, and finally executes the `<transform>` descriptor for `Outer`. However, if default transformation is disabled for `Outer`, then no transformation is performed on the `Inner` member and therefore the `<transform>` descriptor for `Inner` is not executed.

`<init>`

Defines custom initialization rules for all instances of a type in the new Slice definitions. Child descriptors are executed each time the type is instantiated. The typical use case for this descriptor is for types that have been introduced in the new Slice definitions and whose instances require default values different than what **transformdb** supplies. The symbol `value` is introduced into a local

scope to represent the instance. The attributes supported by this descriptor are described in Table 40.4.

Table 40.4. Attributes for `<init>` descriptor.

Name	Description
`type`	Specifies the type id of the new Slice definition.

Here is a simple example of an `<init>` descriptor:

```
<init type="::Player">
    <set target="value.currency" value="100"/>
</init>
```

Note that, like `<transform>`, `<init>` descriptors are executed recursively. For example, if an `<init>` descriptor is defined for a `struct` type, the `<init>` descriptors of the `struct`'s members are executed before the `struct`'s descriptor.

`<iterate>`

Iterates over a dictionary or sequence, executing child descriptors for each element. The symbol names selected to represent the element information may conflict with existing symbols in the enclosing scope, in which case those outer symbols are not accessible to child descriptors. The attributes supported by this descriptor are described in Table 40.5.

Table 40.5. Attributes for `<iterate>` descriptor.

Name	Description
`target`	The sequence or dictionary.
`index`	The symbol name used for the sequence index. If not specified, the default symbol is `i`.
`element`	The symbol name used for the sequence element. If not specified, the default symbol is `elem`.
`key`	The symbol name used for the dictionary key. If not specified, the default symbol is `key`.
`value`	The symbol name used for the dictionary value. If not specified, the default symbol is `value`.

Shown below is an example of an `<iterate>` descriptor that sets the new data member `reviewSalary` to `true` if the employee's salary is greater than $3000.

```
<iterate target="new.employeeMap" key="id" value="emp">
    <if test="emp.salary > 3000">
        <set target="emp.reviewSalary" value="true"/>
    </if>
</iterate>
```

`<if>`

Conditionally executes child descriptors. The attributes supported by this descriptor are described in Table 40.6.

Table 40.6. Attributes for `<if>` descriptor.

Name	Description
`test`	A boolean expression.

See Section 40.8 for more information on the descriptor expression language.

`<set>`

Modifies a value. The `value` and `type` attributes are mutually exclusive. If `target` denotes a dictionary element, that element must already exist (i.e., `<set>` cannot be used to add an element to a dictionary). The attributes supported by this descriptor are described in Table 40.7.

Table 40.7. Attributes for `<set>` descriptor.

Name	Description
`target`	An expression that must select a modifiable value.
`value`	An expression that must evaluate to a value compatible with the target's type.
`type`	If specified, set the target to be an instance of the given Slice class. The value is a type id from the new Slice definitions. The class must be compatible with the target's type.

Table 40.7. Attributes for `<set>` descriptor.

Name	Description
`length`	An integer expression representing the desired new length of a sequence. If the new length is less than the current size of the sequence, elements are removed from the end of the sequence. If the new length is greater than the current size, new elements are added to the end of the sequence. If `value` or `type` is also specified, it is used to initialize each new element.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is `false`.

The `<set>` descriptor below modifies a member of a dictionary element:

```
<set target="new.parts['P105J3'].cost"
      value="new.parts['P105J3'].cost * 1.05"/>
```

This `<set>` descriptor adds an element to a sequence and initializes its value:

```
<set target="new.partsList" length="new.partsList.length + 1"
     value="'P105J3'"/>
```

As another example, the following <set> descriptor changes the value of an enumeration. Notice that the value refers to a symbol in the new Slice definitions (see Section 40.8.3 for more information).

```
<set target="new.ingredient" value="::New::Apple"/>
```

`<add>`

Adds a new element to a sequence or dictionary. It is legal to add an element while traversing the sequence or dictionary using `<iterate>`, however the traversal order after the addition is undefined. The `key` and `index` attributes are mutually exclusive, as are the `value` and `type` attributes. If neither `value` nor `type` is

specified, the new element is initialized with a default value. The attributes supported by this descriptor are described in Table 40.8.

Table 40.8. Attributes for `<add>` descriptor.

Name	Description
`target`	An expression that must select a modifiable sequence or dictionary.
`key`	An expression that must evaluate to a value compatible with the target dictionary's key type.
`index`	An expression that must evaluate to an integer value representing the insertion position. The new element is inserted before `index`. The value must not exceed the length of the target sequence.
`value`	An expression that must evaluate to a value compatible with the target dictionary's value type, or the target sequence's element type.
`type`	If specified, set the target value or element to be an instance of the given Slice class. The value is a type id from the new Slice definitions. The class must be compatible with the target dictionary's value type, or the target sequence's element type.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is `false`.

Below is an example of an `<add>` descriptor that adds a new dictionary element and then initializes its member:

```
<add target="new.parts" key="'P105J4'"/>
<set target="new.parts['P105J4'].cost" value="3.15"/>
```

<define>

Defines a new symbol in the current scope. The attributes supported by this descriptor are described in Table 40.9.

Table 40.9. Attributes for `<define>` descriptor.

Name	Description
`name`	The name of the new symbol. An error occurs if the name matches an existing symbol in the current scope.
`type`	The name of the symbol's formal Slice type. For user-defined types, the name should be prefixed with `::Old` or `::New` to indicate the source of the type. The prefix can be omitted for primitive types.
`value`	An expression that must evaluate to a value compatible with the symbol's type.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Execution fails immediately if a range error occurs. If not specified, the default value is `false`.

Below are two examples of the `<define>` descriptor. The first example defines the symbol `identity` to have type `Ice::Identity`, and proceeds to initialize its members using `<set>`:

```
<define name="identity" type="::New::Ice::Identity"/>
<set target="identity.name" value="steve"/>
<set target="identity.category" value="Admin"/>
```

The second example uses the enumeration we first saw in Section 40.3.5 to define the symbol `manufacturer` and assign it a default value:

```
<define name="manufacturer" type="::New::BigThree"
        value="::New::Daimler"/>
```

<remove>

Removes an element from a sequence or dictionary. It is legal to remove an element while traversing a sequence or dictionary using `<iterate>`, however

the traversal order after removal is undefined. The attributes supported by this descriptor are described in Table 40.10.

Table 40.10. Attributes for `<remove>` descriptor.

Name	Description
`target`	An expression that must select a modifiable sequence or dictionary.
`key`	An expression that must evaluate to a value compatible with the key type of the target dictionary.
`index`	An expression that must evaluate to an integer value representing the index of the sequence element to be removed.

`<fail>`

Causes transformation to fail immediately. If `test` is specified, transformation fails only if the expression evaluates to `true`. The attributes supported by this descriptor are described in Table 40.11.

Table 40.11. Attributes for `<fail>` descriptor.

Name	Description
`message`	A message to display upon transformation failure.
`test`	A boolean expression.

The following `<fail>` descriptor terminates the transformation if a range error is detected:

```
<fail message="range error occurred in ticket count!"
      test="old.ticketCount > 32767"/>
```

`<delete>`

Causes transformation of the current database record to cease, and removes the record from the transformed database. This descriptor has no attributes.

<echo>

Displays values and informational messages. If no attributes are specified, only a newline is printed. The attributes supported by this descriptor are described in Table 40.12.

Table 40.12. Attributes for <echo> descriptor.

Name	Description
message	A message to display.
value	An expression. The value of the expression is displayed in a structured format.

Shown below is an <echo> descriptor that uses both message and value attributes:

```
<if test="old.ticketCount > 32767">
    <echo message="deleting record with invalid ticket count: "
          value="old.ticketCount"/>
    <delete/>
</if>
```

40.4.5 Descriptor Guidelines

There are three points at which you can intercept the transformation process: when transforming a record (<record>), when transforming an instance of a type (<transform>), and when creating an instance of a type (<init>).

In general, <record> is used when your modifications require access to both the key and value of the record. For example, if the database key is needed as a factor in an equation, or to identify an element in a dictionary, then <record> is the only descriptor in which this type of modification is possible. The <record> descriptor is also convenient to use when the number of changes to be made is small, and does not warrant the effort of writing separate <transform> or <init> descriptors.

The <transform> descriptor has a more limited scope than <record>. It is used when changes must potentially be made to all instances of a type (regardless of the context in which that type is used) and access to the old value is necessary. The <transform> descriptor does not have access to the database key and

value, therefore decisions can only be made based on the old and new instances of the type in question.

Finally, the `<init>` descriptor is useful when access to the old instance is not required in order to properly initialize a type. In most cases, this activity could also be performed by a `<transform>` descriptor that simply ignored the old instance, so `<init>` may seem redundant. However, there is one situation where `<init>` is required: when it is necessary to initialize an instance of a type that is introduced by the new Slice definitions. Since there are no instances of this type in the current database, a `<transform>` descriptor for that type would never be executed.

40.5 Using `transformdb`

This section describes the invocation of the **`transformdb`** tool, and provides advice on how best to use it. The tool operates in one of three modes:

- Automatic migration
- Custom migration
- Analysis

Section 40.3 provided an overview of the tool's automatic and custom migration modes. The only difference between these two modes is the source of the transformation descriptors: for automatic migration, **`transformdb`** internally generates and executes a default set of descriptors, whereas for custom migration the user specifies an external file containing the transformation descriptors to be executed.

In analysis mode, **`transformdb`** creates a file containing the default transformation descriptors it would have used during automatic migration. You would normally review this file and possibly customize it prior to executing the tool again in its custom migration mode.

40.5.1 Database Catalogs

As explained in Section 39.7, Freeze maintains schema information in a catalog for each database environment. If necessary, **`transformdb`** will use the catalog to determine the names of the databases in the environment, and to determine the key and value types of a particular database. There are two advantages to the tool's use of the catalog:

1. it allows **transformdb** to operate on all of the databases in a single invocation
2. it eliminates the need for you to specify type information for a database.

For example, you can use automatic migration to transform all of the databases at one time, as shown below:

```
$ transformdb [options] old-env new-env
```

Since we omitted the name of a database to be migrated, **transformdb** uses the catalog in the environment **old-env** to discover all of the databases and their types, generates default transformations for each database, and performs the migration. However, we must still ensure that **transformdb** has loaded the old and new Slice types used by *all* of the databases in the environment.

40.5.2 Slice Options

The tool supports the standard command-line options common to all Slice processors listed in Section 4.20, with the exception of the include directory (**-I**) option. The options specific to **transformdb** are described below.

- **--old *SLICE***
  --new *SLICE*

  Loads the old or new Slice definitions contained in the file ***SLICE***. These options may be specified multiple times if several files must be loaded. However, it is the user's responsibility to ensure that duplicate definitions do not occur (which is possible when two files are loaded that share a common include file). One strategy for avoiding duplicate definitions is to load a single Slice file that contains only `#include` statements for each of the Slice files to be loaded. No duplication is possible in this case if the included files use include guards correctly.

- **--include-old *DIR***
  --include-new *DIR*

  Adds the directory ***DIR*** to the set of include paths for the old or new Slice definitions.

40.5.3 Type Options

In invocation modes for which **transformdb** requires that you define the types used by a database, you must specify one of the following options:

- **`--key` *`TYPE`*`[,`*`TYPE`*`]`**
  `--value` *`TYPE`*`[,`*`TYPE`*`]`

  Specifies the Slice type(s) of the database key and value. If the type does not change, then the type only needs to be specified once. Otherwise, the old type is specified first, followed by a comma and the new type. For example, the option **`--key int,string`** indicates that the database key is migrating from `int` to `string`. On the other hand, the option **`--key int,int`** indicates that the key type does not change, and could be given simply as **`--key int`**. Type changes are restricted to those allowed by the compatibility rules defined in Section 40.3.2, but custom migration provides additional flexibility.

- **`-e`**

  Indicates that a Freeze evictor database is being migrated. As a convenience, this option automatically sets the database key and value types to those appropriate for the Freeze evictor, and therefore the **`--key`** and **`--value`** options are not necessary. Specifically, the key type of a Freeze evictor database is `Ice::Identity`, and the value type is `Freeze::ObjectRecord`. The latter is defined in the Slice file `Freeze/EvictorStorage.ice`; however, this file does not need to be loaded into your old and new Slice definitions.

40.5.4 General Options

These options may be specified during analysis or migration, as indicated below:

- **`-i`**

  Requests that **`transformdb`** ignore type changes that violate the compatibility rules defined in Section 40.3.2. If this option is not specified, **`transformdb`** fails immediately if such a violation occurs. With this option, a warning is displayed but **`transformdb`** continues the requested action. The **`-i`** option can be specified in analysis or automatic migration modes.

- **`-p`**

  During migration, this option requests that **`transformdb`** purge object instances whose type is no longer found in the new Slice definitions. See Section 40.5.10 for more information.

- **`-c`**

  Use catastrophic recovery on the old BerkeleyDB database environment prior to migration.

- **-w**

  Suppress duplicate warnings during migration. This option is especially useful to minimize diagnostic messages when **transformdb** would otherwise emit the same warning many times, such as when it detects the same issue in every record of a database.

40.5.5 Database Arguments

In addition to the options described above, **transformdb** accepts as many as three arguments that specify the names of databases and database environments:

- **dbenv**

  The pathname of the old database environment directory.

- **db**

  The name of an existing database file in **dbenv**. **transformdb** never modifies this database.

- **newdbenv**

  The pathname of the database environment directory to contain the transformed database(s). This directory must exist and must not contain an existing database whose name matches a database being migrated.

40.5.6 Automatic Migration

You can use **transformdb** to automatically migrate one database or all databases in an environment.

Migrating a Single Database

Use the following command line to migrate one database:

```
$ transformdb [slice-opts] [type-opts] [gen-opts] \
dbenv db newdbenv
```

If you omit **type-opts**, the tool obtains type information for database **db** from the catalog (see Section 40.5.1). For example, consider the following command, which uses automatic migration to transform a database with a key type of `int` and value type of `string` into a database with the same key type and a value type of `long`:

```
$ transformdb --key int --value string,long \
dbhome data.db newdbhome
```

Note that we did not need to specify the Slice options **--old** or **--new** because our key and value types are primitives. Upon successful completion, the file **newdbhome/data.db** contains our transformed database.

Migrating All Databases

To migrate all databases in the environment, use a command like the one shown below:

```
$ transformdb [slice-opts] [gen-opts] dbenv newdbenv
```

In this invocation mode, you must ensure that **transformdb** has loaded the old and new Slice definitions for all of the types it will encounter among the databases in the environment.

40.5.7 Analysis

Custom migration is a two-step process: you first write the transformation descriptors, and then execute them to transform a database. To assist you in the process of creating a descriptor file, **transformdb** can generate a default set of transformation descriptors by comparing your old and new Slice definitions. This feature is enabled by specifying the following option:

- **-o *FILE***

  Specifies the descriptor file ***FILE*** to be created during analysis. No migration occurs in this invocation mode.

Generated File

The generated file contains a `<transform>` descriptor for each type that appears in both old and new Slice definitions, and an `<init>` descriptor for types that appear only in the new Slice definitions. In most cases, these descriptors are empty. However, they can contain XML comments describing changes detected by **transformdb** that may require action on your part.

For example, let us revisit the enumeration we defined in Section 40.3.5:

```
enum BigThree { Ford, DaimlerChrysler, GeneralMotors };
```

This enumeration has evolved into the one shown below. In particular, the `DaimlerChrysler` enumerator has been renamed to reflect a corporate name change:

```
enum BigThree { Ford, Daimler, GeneralMotors };
```

Next we run **transformdb** in analysis mode:

```
$ transformdb --old old/BigThree.ice \
--new new/BigThree.ice --key string \
--value ::BigThree -o transform.xml
```

The generated file **transform.xml** contains the following descriptor for the enumeration BigThree:

```
<transform type="::BigThree">
    <!-- NOTICE: enumerator `DaimlerChrysler' has been removed -->
</transform>
```

The comment indicates that enumerator DaimlerChrysler is no longer present in the new definition, reminding us that we need to add logic in this <transform> descriptor to change all occurrences of DaimlerChrysler to Daimler.

The descriptor file generated by **transformdb** is well-formed and does not require any manual intervention prior to being executed. However, executing an unmodified descriptor file is simply the equivalent of using automatic migration.

Invocation Modes

The sample command line shown in the previous section specified the key and value types of the database explicitly. This invocation mode has the following general form:

```
$ transformdb [slice-opts] [type-opts] [gen-opts] \
-o FILE
```

Upon successful completion, the generated file contains a <database> descriptor that records the type information supplied by **type-opts**, in addition to the <transform> and <init> descriptors described earlier.

For your convenience, you can omit **type-opts** and allow **transformdb** to obtain type information from the catalog instead:

```
$ transformdb [slice-opts] [gen-opts] -o FILE dbenv
```

In this case, the generated file contains a <database> descriptor for each database in the catalog. Note that in this invocation mode, **transformdb** must assume that the names of the database key and value types have not changed, since the only type information available is the catalog in the old database environment. If the tool is unable to locate a new Slice definition for a database's key or value type, it emits a warning message and generates a placeholder value in the output file that you must modify prior to migration.

40.5.8 Custom Migration

After preparing a descriptor file, either by writing one completely yourself, or modifying one generated by the analysis mode described in the previous section, you are ready to migrate a database. One additional option is provided for migration:

- **-f *FILE***

  Execute the transformation descriptors in the file ***FILE***.

To transform one database, use the following command:

```
$ transformdb [slice-opts] [gen-opts] -f FILE dbenv db \
newdbenv
```

The tool searches the descriptor file for a `<database>` descriptor whose `name` attribute matches **db**. If no match is found, it searches for a `<database>` descriptor that does not have a `name` attribute.

If you want to transform all databases in the environment, you can omit the database name:

```
$ transformdb [slice-opts] [gen-opts] -f FILE dbenv \
newdbenv
```

In this case, the descriptor file must contain a `<database>` element for each database in the environment.

Continuing our enumeration example from the analysis discussion above, assume we have modified **`transform.xml`** to convert the `Chrysler` enumerator, and are now ready to execute the transformation:

```
$ transformdb --old old/BigThree.ice \
--new new/BigThree.ice -f transform.xml \
dbhome bigthree.db newdbhome
```

40.5.9 Usage Strategies

If it becomes necessary for you to transform a Freeze database, we generally recommend that you attempt to use automatic migration first, unless you already know that custom migration is necessary. Since transformation is a non-destructive process, there is no harm in attempting an automatic migration, and it is a good way to perform a sanity check on your **`transformdb`** arguments (for example, to ensure that all the necessary Slice files are being loaded), as well as on the database itself. If **`transformdb`** detects any incompatible type changes, it displays an error message for each incompatible change and terminates without

doing any transformation. In this case, you may want to run **transformdb** again with the **-i** option, which ignores incompatible changes and causes transformation to proceed.

Pay careful attention to any warnings that **transformdb** emits, as these may indicate the need for using custom migration. For example, if we had attempted to transform the database containing the `BigThree` enumeration from previous sections using automatic migration, any occurrences of the `Chrysler` enumerator would display the following warning:

```
warning: unable to convert 'Chrysler' to ::BigThree
```

If custom migration appears to be necessary, use analysis to generate a default descriptor file, then review it for `NOTICE` comments and edit as necessary. Liberal use of the `<echo>` descriptor can be beneficial when testing your descriptor file, especially from within the `<record>` descriptor where you can display old and new keys and values.

40.5.10 Transforming Objects

The polymorphic nature of Slice classes can cause problems for database migration. As an example, the Slice parser can ensure that a set of Slice definitions loaded into **transformdb** is complete for all types but classes (and exceptions, but we ignore those because they are not persistent). **transformdb** cannot know that a database may contain instances of a subclass that is derived from one of the loaded classes but whose definition is not loaded. Alternatively, the type of a class instance may have been renamed and cannot be found in the new Slice definitions.

By default, these situations result in immediate transformation failure. However, the **-p** option is a (potentially drastic) way to handle these situations: if a class instance has no equivalent in the new Slice definitions and this option is specified, **transformdb** removes the instance any way it can. If the instance appears in a sequence or dictionary element, that element is removed. Otherwise, the database record containing the instance is deleted.

Now, the case of a class type being renamed is handled easily enough using custom migration and the `rename` attribute of the `<transform>` descriptor. However, there are legitimate cases where the destructive nature of the **-p** option can be useful. For example, if a class type has been removed and it is simply easier to start with a database that is guaranteed not to contain any instances of that type, then the **-p** option may simplify the broader migration effort.

This is another situation in which running an automatic migration first can help point out the trouble spots in a potential migration. Using the **-p** option, **transformdb** emits a warning about the missing class type and continues, rather than halting at the first occurrence, enabling you to discover whether you have forgotten to load some Slice definitions, or need to rename a type.

40.5.11 Using `transformdb` on an Open Environment

It is possible to use **transformdb** to migrate databases in an environment that is currently open by another process, but if you are not careful you can easily corrupt the environment and cause the other process to fail. To avoid such problems, you must configure both **transformdb** and the other process to set `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0` (refer to Appendix D for a description of this property). This property has a default value of one, therefore you must explicitly set it to zero. Note that **transformdb** makes no changes to the existing database environment, but it requires exclusive access to the new database environment until transformation is complete.

If you run **transformdb** on an open environment but neglect to set `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`, you can expect **transformdb** to terminate immediately with an error message stating that the database environment is locked. Before running **transformdb** on an open environment, we strongly recommend that you first verify that the other process was also configured with `Freeze.DbEnv.`*`env-name`*`.DbPrivate=0`.

40.6 Database Inspection

The FreezeScript tool **dumpdb** is used to examine a Freeze database. Its simplest invocation displays every record of the database, but the tool also supports more selective activities. In fact, **dumpdb** supports a scripted mode that shares many of the same XML descriptors as **transformdb** (see Section 40.4), enabling sophisticated filtering and reporting.

40.6.1 Descriptor Overview

A **dumpdb** descriptor file has a well-defined structure. The top-level descriptor in the file is `<dumpdb>`. A `<database>` descriptor must be present within `<dumpdb>` to define the key and value types used by the database. Inside

`<database>`, the `<record>` descriptor triggers database traversal. Shown below is an example that demonstrates the structure of a minimal descriptor file:

```
<dumpdb>
    <database key="string" value="::Employee">
        <record>
            <echo message="Key: " value="key"/>
            <echo message="Value: " value="value"/>
        </record>
    </database>
</dumpdb>
```

During traversal, type-specific actions are supported by the `<dump>` descriptor, which is a child of `<dumpdb>`. One `<dump>` descriptor may be defined for each type in the Slice definitions. Each time **dumpdb** encounters an instance of a type, the `<dump>` descriptor for the type is executed.

The `<database>`, `<record>`, and `<dump>` descriptors may contain general-purpose action descriptors such as `<if>` and `<echo>`. These actions resemble statements in programming languages like C++ and Java, in that they are executed in the order of definition and their effects are cumulative. Actions make use of the expression language described in Section 40.8.

Although **dumpdb** descriptors are not allowed to modify the database, they can still define local symbols for scripting purposes. Once a symbol is defined by the `<define>` descriptor, other descriptors such as `<set>`, `<add>`, and `<remove>` can be used to manipulate the symbol's value.

40.6.2 Flow of Execution

The descriptors are executed as described below.

- `<database>` is executed first. Each child descriptor of `<database>` is executed in the order of definition. If a `<record>` descriptor is present, database traversal occurs at that point. Any child descriptors of `<database>` that follow `<record>` are not executed until traversal completes.
- For each record, **dumpdb** interprets the key and value, invoking `<dump>` descriptors for each type it encounters. For example, if the value type of the database is a `struct`, then **dumpdb** first attempts to invoke a `<dump>` descriptor for the structure type, and then recursively interprets the structure's members in the same fashion.

See Section 40.6.4 for detailed information on the **dumpdb** descriptors.

40.6.3 Scopes

The `<database>` descriptor creates a global scope, allowing child descriptors of `<database>` to define symbols that are accessible in any descriptor[3]. Furthermore, certain other descriptors create local scopes that exist only for the duration of the descriptor's execution. For example, the `<dump>` descriptor creates a local scope and defines the symbol `value` to represent a value of the specified type. Child descriptors of `<dump>` can also define new symbols in the local scope, as long as those symbols do not clash with an existing symbol in that scope. It is legal to add a new symbol with the same name as a symbol in an outer scope, but the outer symbol will not be accessible during the descriptor's execution.

The global scope is useful in many situations. For example, suppose you want to track the number of times a certain value was encountered during database traversal. This can be accomplished as shown below:

```
<dumpdb>
    <database key="string" value="::Ice::Identity">
        <define name="categoryCount" type="int" value="0"/>
        <record/>
        <echo message="categoryCount = " value="categoryCount"/>
    </database>
    <dump type="::Ice::Identity">
        <if test="value.category == `Accounting'">
            <set target="categoryCount"
                value="categoryCount + 1"/>
        </if>
    </dump>
</dumpdb>
```

In this example, the `<define>` descriptor introduces the symbol `categoryCount` into the global scope, defining it as type `int` with an initial value of zero. Next, the `<record>` descriptor causes traversal to proceed. Each occurrence of the type `Ice::Identity` causes its `<dump>` descriptor to be executed, which examines the `category` member and increases `categoryCount` if necessary. Finally, after traversal completes, the `<echo>` descriptor displays the final value of `categoryCount`.

To reinforce the relationships between descriptors and scopes, consider the diagram in Figure 40.2. Several descriptors are shown, including the symbols they

3. In order for a global symbol to be available to a `<dump>` descriptor, the symbol must be defined before the `<record>` descriptor is executed.

define in their local scopes. In this example, the `<iterate>` descriptor has a dictionary target and therefore the default symbol for the element value, `value`, hides the symbol of the same name in the parent `<dump>` descriptor's scope[4]. In addition to symbols in the `<iterate>` scope, child descriptors of `<iterate>` can also refer to symbols from the `<dump>` and `<database>` scopes.

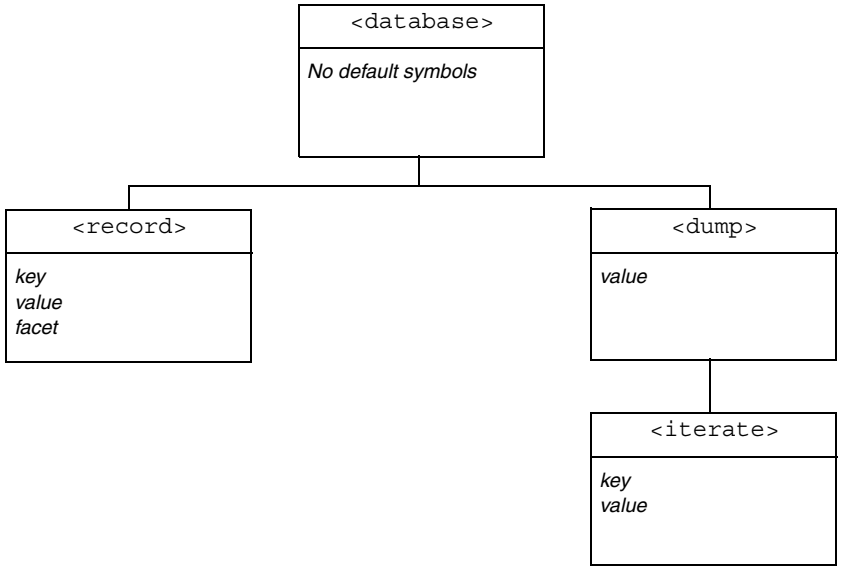


Figure 40.2. Relationship between descriptors and scopes.

40.6.4 Descriptor Reference

`<dumpdb>`

The top-level descriptor in a descriptor file. It requires one child descriptor, `<database>`, and supports any number of `<dump>` descriptors. This descriptor has no attributes.

4. This situation can be avoided by assigning a different symbol name to the element value.

<database>

The attributes of this descriptor define the key and value types of the database. It supports any number of child descriptors, but at most one <record> descriptor. The <database> descriptor also creates a global scope for user-defined symbols (see Section 40.6.3).

The attributes supported by the <database> descriptor are described in Table 40.13.

Table 40.13. Attributes for <database> descriptor.

Name	Description
key	Specifies the Slice type of the database key.
value	Specifies the Slice type of the database value.

As an example, consider the following <database> descriptor. In this case, the Freeze map to be examined has key type int and value type ::Employee:

```
<database key="int" value="::Employee">
```

<record>

Commences the database traversal. Child descriptors are executed for each record in the database, but after any <dump> descriptors are executed. The <record> descriptor introduces the read-only symbols key, value and facet into a local scope. These symbols are accessible to child descriptors, but not to <dump> descriptors. The facet symbol is a string indicating the facet name of the object in the current record, and is only relevant for Freeze evictor databases.

Note that database traversal only occurs if a <record> descriptor is present.

<dump>

Executed for all instances of a Slice type. Only one <dump> descriptor can be specified for a type, but a <dump> descriptor is not required for every type. The

read-only symbol `value` is introduced into a local scope. The attributes supported by this descriptor are described in Table 40.14.

Table 40.14. Attributes for `<dump>` descriptor.

Name	Description
`type`	Specifies the Slice type id.
`base`	If `type` denotes a Slice class, this attribute determines whether the `<dump>` descriptor of the base class is invoked. If `true`, the base class descriptor is invoked after executing the child descriptors. If not specified, the default value is `true`.
`contents`	For `class` and `struct` types, this attribute determines whether descriptors are executed for members of the value. For `sequence` and `dictionary` types, this attribute determines whether descriptors are executed for elements. If not specified, the default value is `true`.

Below is an example of a `<dump>` descriptor that searches for certain products:

```
<dump type="::Product">
    <if test="value.description.find(`scanner') != -1">
        <echo message="Scanner SKU: " value="value.SKU"/>
    </if>
</dump>
```

For class types, **`dumpdb`** first attempts to locate a `<dump>` descriptor for the object's most-derived type. If no descriptor is found, **`dumpdb`** proceeds up the class hierarchy in an attempt to find a descriptor. The base object type, `Object`, is the root of every class hierarchy and is included in the search for descriptors. It is therefore possible to define a `<dump>` descriptor for type `Object`, which will be invoked for every class instance.

Note that `<dump>` descriptors are executed recursively. For example, consider the following Slice definitions:

```
struct Inner {
    int sum;
};
struct Outer {
    Inner i;
};
```

When **dumpdb** is interpreting a value of type `Outer`, it executes the `<dump>` descriptor for `Outer`, then recursively executes the `<dump>` descriptor for the `Inner` member, but only if the `contents` attribute of the `Outer` descriptor has the value `true`.

`<iterate>`

Iterates over a dictionary or sequence, executing child descriptors for each element. The symbol names selected to represent the element information may conflict with existing symbols in the enclosing scope, in which case those outer symbols are not accessible to child descriptors. The attributes supported by this descriptor are described in Table 40.15.

Table 40.15. Attributes for `<iterate>` descriptor.

Name	Description
`target`	The sequence or dictionary.
`index`	The symbol name used for the sequence index. If not specified, the default symbol is `i`.
`element`	The symbol name used for the sequence element. If not specified, the default symbol is `elem`.
`key`	The symbol name used for the dictionary key. If not specified, the default symbol is `key`.
`value`	The symbol name used for the dictionary value. If not specified, the default symbol is `value`.

Shown below is an example of an `<iterate>` descriptor that displays the name of an employee if the employee's salary is greater than $3000.

```
<iterate target="value.employeeMap" key="id" value="emp">
    <if test="emp.salary > 3000">
        <echo message="Employee: " value="emp.name"/>
    </if>
</iterate>
```

<if>

Conditionally executes child descriptors. The attributes supported by this descriptor are described in Table 40.16.

Table 40.16. Attributes for <if> descriptor.

Name	Description
test	A boolean expression.

See Section 40.8 for more information on the descriptor expression language.

<set>

Modifies a value. The value and type attributes are mutually exclusive. If target denotes a dictionary element, that element must already exist (i.e., <set> cannot be used to add an element to a dictionary). The attributes supported by this descriptor are described in Table 40.17.

Table 40.17. Attributes for <set> descriptor.

Name	Description
target	An expression that must select a modifiable value.
value	An expression that must evaluate to a value compatible with the target's type.
type	The Slice type id of a class to be instantiated. The class must be compatible with the target's type.
length	An integer expression representing the desired new length of a sequence. If the new length is less than the current size of the sequence, elements are removed from the end of the sequence. If the new length is greater than the current size, new elements are added to the end of the sequence. If value or type is also specified, it is used to initialize each new element.
convert	If true, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is false.

The `<set>` descriptor below modifies a member of a dictionary element:

```
<set target="new.parts['P105J3'].cost"
      value="new.parts['P105J3'].cost * 1.05"/>
```

This `<set>` descriptor adds an element to a sequence and initializes its value:

```
<set target="new.partsList" length="new.partsList.length + 1"
     value="'P105J3'"/>
```

`<add>`

Adds a new element to a sequence or dictionary. It is legal to add an element while traversing the sequence or dictionary using `<iterate>`, however the traversal order after the addition is undefined. The `key` and `index` attributes are mutually exclusive, as are the `value` and `type` attributes. If neither `value` nor `type` is specified, the new element is initialized with a default value. The attributes supported by this descriptor are described in Table 40.18.

Table 40.18. Attributes for `<add>` descriptor.

Name	Description
`target`	An expression that must select a modifiable sequence or dictionary.
`key`	An expression that must evaluate to a value compatible with the target dictionary's key type.
`index`	An expression that must evaluate to an integer value representing the insertion position. The new element is inserted before `index`. The value must not exceed the length of the target sequence.
`value`	An expression that must evaluate to a value compatible with the target dictionary's value type, or the target sequence's element type.
`type`	The Slice type id of a class to be instantiated. The class must be compatible with the target dictionary's value type, or the target sequence's element type.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is `false`.

Below is an example of an `<add>` descriptor that adds a new dictionary element and then initializes its member:

```
<add target="new.parts" key="'P105J4'"/>
<set target="new.parts['P105J4'].cost" value="3.15"/>
```

`<define>`

Defines a new symbol in the current scope. The attributes supported by this descriptor are described in Table 40.19.

Table 40.19. Attributes for `<define>` descriptor.

Name	Description
`name`	The name of the new symbol. An error occurs if the name matches an existing symbol in the current scope.
`type`	The name of the symbol's formal Slice type.
`value`	An expression that must evaluate to a value compatible with the symbol's type.
`convert`	If `true`, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Execution fails immediately if a range error occurs. If not specified, the default value is `false`.

Below are two examples of the `<define>` descriptor. The first example defines the symbol `identity` to have type `Ice::Identity`, and proceeds to initialize its members using `<set>`:

```
<define name="identity" type="::Ice::Identity"/>
<set target="identity.name" value="steve"/>
<set target="identity.category" value="Admin"/>
```

The second example uses the enumeration we first saw in Section 40.3.5 to define the symbol `manufacturer` and assign it a default value:

```
<define name="manufacturer" type="::BigThree"
    value="::DaimlerChrysler"/>
```

`<remove>`

Removes an element from a sequence or dictionary. It is legal to remove an element while traversing a sequence or dictionary using `<iterate>`, however

the traversal order after removal is undefined. The attributes supported by this descriptor are described in Table 40.20.

Table 40.20. Attributes for `<remove>` descriptor.

Name	Description
`target`	An expression that must select a modifiable sequence or dictionary.
`key`	An expression that must evaluate to a value compatible with the key type of the target dictionary.
`index`	An expression that must evaluate to an integer value representing the index of the sequence element to be removed.

`<fail>`

Causes transformation to fail immediately. If `test` is specified, transformation fails only if the expression evaluates to `true`. The attributes supported by this descriptor are described in Table 40.21.

Table 40.21. Attributes for `<fail>` descriptor.

Name	Description
`message`	A message to display upon transformation failure.
`test`	A boolean expression.

The following `<fail>` descriptor terminates the transformation if a range error is detected:

```
<fail message="range error occurred in ticket count!"
      test="value.ticketCount > 32767"/>
```

<echo>

Displays values and informational messages. If no attributes are specified, only a newline is printed. The attributes supported by this descriptor are described in Table 40.22.

Table 40.22. Attributes for <echo> descriptor.

Name	Description
message	A message to display.
value	An expression. The value of the expression is displayed in a structured format.

Shown below is an <echo> descriptor that uses both message and value attributes:

```
<if test="value.ticketCount > 32767">
    <echo message="range error occurred in ticket count: "
          value="value.ticketCount"/>
</if>
```

40.7 Using dumpdb

This section describes the invocation of **dumpdb** and provides advice on how to best use it.

40.7.1 Options

The tool supports the standard command-line options common to all Slice processors listed in Section 4.20. The options specific to **dumpdb** are described below:

- **--load *SLICE***

  Loads the Slice definitions contained in the file ***SLICE***. This option may be specified multiple times if several files must be loaded. However, it is the user's responsibility to ensure that duplicate definitions do not occur (which is possible when two files are loaded that share a common include file). One strategy for avoiding duplicate definitions is to load a single Slice file that contains only #include statements for each of the Slice files to be loaded. No

duplication is possible in this case if the included files use include guards correctly.

- **--key *TYPE***
  --value *TYPE*

  Specifies the Slice type of the database key and value. If these options are not specified, and the **-e** option is not used, **dumpdb** obtains type information from the catalog.

- **-e**

  Indicates that a Freeze evictor database is being examined. As a convenience, this option automatically sets the database key and value types to those appropriate for the Freeze evictor, and therefore the **--key** and **--value** options are not necessary. Specifically, the key type of a Freeze evictor database is `Ice::Identity`, and the value type is `Freeze::ObjectRecord`. The latter is defined in the Slice file `Freeze/EvictorStorage.ice`, however this file does not need to be explicitly loaded.

  If this option is not specified, and the **--key** and **--value** options are not used, **dumpdb** obtains type information from the catalog.

- **-o *FILE***

  Create a file named ***FILE*** containing sample descriptors for the loaded Slice definitions. If type information is not specified, **dumpdb** obtains it from the catalog. If the **--select** option is used, its expression is included in the sample descriptors. Database traversal does not occur when the **-o** option is used.

- **-f *FILE***

  Execute the descriptors in the file named ***FILE***. The file's `<database>` descriptor specifies the key and value types; therefore it is not necessary to supply type information.

- **--select *EXPR***

  Only display those records for which the expression ***EXPR*** is true. The expression can refer to the symbols `key` and `value`.

- **-c, --catalog**

  Display information about the databases in an environment, or about a particular database. This option presents the type information contained in the catalog (see Section 39.7).

40.7.2 Database Arguments

If **dumpdb** is invoked to examine a database, it requires two arguments:

- **dbenv**

  The pathname of the database environment directory.

- **db**

  The name of the database file. **dumpdb** opens this database as read-only, and traversal occurs within a transaction.

To display catalog information using the **-c** option, the database environment directory **dbenv** is required. If the database file argument **db** is omitted, **dumpdb** displays information about every database in the catalog.

40.7.3 Use Cases

The command line options described in Section 40.7.1 support several modes of operation:

- Dump an entire database.
- Dump selected records of a database.
- Emit a sample descriptor file.
- Execute a descriptor file.
- Examine the catalog.

These use cases are described in the following sections.

Dump an Entire Database

The simplest way to examine a database with **dumpdb** is to dump its entire contents. You must specify the database key and value types, load the necessary Slice definitions, and supply the names of the database environment directory and database file. For example, this command dumps a Freeze map database whose key type is `string` and value type is `Employee`:

```
$ dumpdb --key string --value ::Employee \
--load Employee.ice db emp.db
```

As a convenience, you may omit the key and value types, in which case **dumpdb** obtains them from the catalog (see Section 39.7):

```
$ dumpdb --load Employee.ice db emp.db
```

Dump Selected Records

If only certain records are of interest to you, the **`--select`** option provides a convenient way to filter the output of **dumpdb**. In the following example, we select employees from the accounting department:

```
$ dumpdb --load Employee.ice \
--select "value.dept == 'Accounting'" db emp.db
```

In cases where the database records contain polymorphic class instances, you must be careful to specify an expression that can be successfully evaluated against all records. For example, **dumpdb** fails immediately if the expression refers to a data member that does not exist in the class instance. The safest way to write an expression in this case is to check the type of the class instance before referring to any of its data members.

In the example below, we assume that a Freeze evictor database contains instances of various classes in a class hierarchy, and we are only interested in instances of `Manager` whose employee count is greater than 10:

```
$ dumpdb -e --load Employee.ice \
--select "value.servant.ice_id == '::Manager' and \
value.servant.group.length > 10" db emp.db
```

Alternatively, if `Manager` has derived classes, then the expression can be written in a different way so that instances of `Manager` and any of its derived classes are considered:

```
$ dumpdb -e --load Employee.ice \
  --select "value.servant.ice_isA('::Manager') and \
  value.servant.group.length > 10" db emp.db
```

Creating a Sample Descriptor File

If you require more sophisticated filtering or scripting capabilities, then you must use a descriptor file. The easiest way to get started with a descriptor file is to generate a template using **dumpdb**:

```
$ dumpdb --key string --value ::Employee \
--load Employee.ice -o dump.xml
```

The output file `dump.xml` is complete and can be executed immediately if desired, but typically the file is used as a starting point for further customization. Again, you may omit the key and value types by specifying the database instead:

```
$ dumpdb --load Employee.ice -o dump.xml db emp.db
```

If the **--select** option is specified, its expression is included in the generated <record> descriptor as the value of the test attribute in an <if> descriptor.

dumpdb terminates immediately after generating the output file.

Executing a Descriptor File

Use the **-f** option when you are ready to execute a descriptor file. For example, we can execute the descriptor we generated in the previous section using this command:

```
$ dumpdb -f dump.xml --load Employee.ice db emp.db
```

Examine the Catalog

The **-c** option displays the contents of the database environment's catalog:

```
$ dumpdb -c db
```

The output indicates whether each database in the environment is associated with an evictor or a map. For maps, the output includes the key and value types.

If you specify the name of a database, **dumpdb** only displays the type information for that database:

```
$ dumpdb -c db emp.db
```

40.7.4 Using dumpdb on an Open Environment

It is possible to use **dumpdb** to migrate databases in an environment that is currently open by another process, but if you are not careful you can easily corrupt the environment and cause the other process to fail. To avoid such problems, you must configure both **dumpdb** and the other process to set Freeze.DbEnv.*env-name*.DbPrivate=0 (refer to Appendix D for a description of this property). This property has a default value of one, therefore you must explicitly set it to zero.

If you run **dumpdb** on an open environment but neglect to set Freeze.DbEnv.*env-name*.DbPrivate=0, you can expect **dumpdb** to terminate immediately with an error message stating that the database environment is locked. Before running **dumpdb** on an open environment, we strongly recommend that you first verify that the other process was also configured with Freeze.DbEnv.*env-name*.DbPrivate=0.

40.8 Descriptor Expression Language

An expression language is provided for use in FreezeScript descriptors.

40.8.1 Operators

The language supports the usual complement of operators: `and`, `or`, `not`, `+`, `-`, `/`, `*`, `%`, `<`, `>`, `==`, `!=`, `<=`, `>=`, `(`, `)`. Note that the `<` character must be escaped as `<` in order to comply with XML syntax restrictions.

40.8.2 Literals

Literal values can be specified for integer, floating point, boolean, and string. The expression language supports the same syntax for literal values as that of Slice (see Section 4.9.5), with one exception: string literals must be enclosed in single quotes.

40.8.3 Symbols

Certain descriptors introduce symbols that can be used in expressions. These symbols must comply with the naming rules for Slice identifiers (i.e., a leading letter followed by zero or more alphanumeric characters). Data members are accessed using dotted notation, such as `value.memberA.memberB`.

Expressions can refer to Slice constants and enumerators using scoped names. In a **`transformdb`** descriptor, there are two sets of Slice definitions, therefore the expression must indicate which set of definitions it is accessing by prefixing the scoped name with `::Old` or `::New`. For example, the expression `old.fruitMember == ::Old::Pear` evaluates to `true` if the data member `fruitMember` has the enumerated value `Pear`. In **`dumpdb`**, only one set of Slice definitions is present and therefore the constant or enumerator can be identified without any special prefix.

40.8.4 Nil

The keyword `nil` represents a nil value of type `Object`. This keyword can be used in expressions to test for a nil object value, and can also be used to set an object value to nil.

40.8.5 Elements

Dictionary and sequence elements are accessed using array notation, such as `userMap['joe']` or `stringSeq[5]`. An error occurs if an expression attempts to access a dictionary or sequence element that does not exist. For dictionaries, the recommended practice is to check for the presence of a key before accessing it:

```
<if test="userMap.containsKey('joe') and userMap['joe'].active">
```

See Section 40.8.8 for more information on the `containsKey` function.

Similarly, expressions involving sequences should check the length of the sequence:

```
<if test="stringSeq.length > 5 and stringSeq[5] == 'fruit'">
```

See Section 40.8.7 for details on the `length` member.

40.8.6 Reserved Keywords

The following keywords are reserved: `and`, `or`, `not`, `true`, `false`, `nil`.

40.8.7 Implicit Members

Certain Slice types support implicit data members:

- Dictionary and sequence instances have a member `length` representing the number of elements.
- Object instances have a member `ice_id` denoting the actual type of the object.

40.8.8 Functions

The expression language supports two forms of function invocation: member functions and global functions. A member function is invoked on a particular data value, whereas global functions are not bound to a data value. For instance, here is an expression that invokes the `find` member function of a `string` value:

```
old.stringValue.find('theSubstring') != -1
```

And here is an example that invokes the global function `stringToIdentity`:

```
stringToIdentity(old.stringValue)
```

If a function takes multiple arguments, the arguments must be separated with commas.

String Member Functions

The `string` data type supports the following member functions:

- `int find(string match[, int start])`

  Returns the index of the substring, or `-1` if not found. A starting position can optionally be supplied.

- `string replace(int start, int len, string str)`

  Replaces a given portion of the string with a new substring, and returns the modified string.

- `string substr(int start[, int len])`

  Returns a substring beginning at the given start position. If the optional length argument is supplied, the substring contains at most `len` characters, otherwise the substring contains the remainder of the string.

Dictionary Member Functions

The `dictionary` data type supports the following member function:

- `bool containsKey(key)`

  Returns `true` if the dictionary contains an element with the given key, or `false` otherwise. The `key` argument must have a value that is compatible with the dictionary's key type.

Object Member Functions

Object instances support the following member function:

- `bool ice_isA(string id)`

  Returns `true` if the object implements the given interface type, or `false` otherwise. This function cannot be invoked on a nil object.

Global Functions

The following global functions are provided:

- `string generateUUID()`

  Returns a new UUID.

- `string identityToString(Ice::Identity id)`

  Converts an identity into its string representation.

- `string lowercase(string str)`
  Returns a new string converted to lowercase.
- `string proxyToString(Ice::ObjectPrx prx)`
  Returns the string representation of the given proxy.
- `Ice::Identity stringToIdentity(string str)`
  Converts a string into an `Ice::Identity`.
- `Ice::ObjectPrx stringToProxy(string str)`
  Converts a string into a proxy.
- `string typeOf(val)`
  Returns the formal Slice type of the argument.

40.9 Summary

FreezeScript provides tools that ease the maintenance of Freeze databases. The **`transformdb`** tool simplifies the task of migrating a database when its persistent types have changed, offering an automatic mode requiring no manual intervention, and a custom mode in which scripted changes are possible. Database inspection and reporting is accomplished using the **`dumpdb`** tool, which supports a number of operational modes including a scripting capability.

Chapter 41
IceSSL

41.1 Chapter Overview

In this chapter we present IceSSL, an optional security component for Ice applications. Section 41.2 provides an overview of the SSL protocol and the infrastructure required to support it. Section 41.3 discusses the installation requirements for each supported language mapping, while Section 41.4 describes typical configuration scenarios. For programs that need to interact directly with IceSSL, the application programming interface is covered in Section 41.5 and Section 41.6. To set up your own certificate authority, you can use the utilities described in Section 41.7.

41.2 Introduction

Security is an important consideration for many distributed applications, both within corporate intranets as well as over untrusted networks, such as the Internet. The ability to protect sensitive information, ensure its integrity, and verify the identities of the communicating parties is essential for developing secure applications. With those goals in mind, Ice includes the IceSSL *plug-in* that provides these capabilities using the Secure Socket Layer (SSL) protocol.[1]

41.2.1 SSL Overview

SSL is the protocol that enables Web browsers to conduct secure transactions and therefore is one of the most commonly used protocols for secure network communication. You do not need to know the technical details of the SSL protocol in order to use IceSSL successfully (and those details are outside the scope of this text). However, it would be helpful to have a high-level understanding of how the protocol works and the infrastructure required to support it. (For more information on the SSL protocol, see [24].)

SSL provides a secure environment for communication (without sacrificing too much performance) by combining a number of cryptographic techniques:

- public key encryption
- symmetric (shared key) encryption
- message authentication codes
- digital certificates

When a client establishes an SSL connection to a server, a *handshake* is performed. During a typical handshake, digital certificates that identify the communicating parties are validated, and symmetric keys are exchanged for encrypting the session traffic. Public key encryption, which is too slow to be used for the bulk of a session's data transfer, is used heavily during the handshaking phase. Once the handshake is complete, SSL uses message authentication codes to ensure data integrity, allowing the client and server to communicate at will with reasonable assurance that their messages are secure.

41.2.2 Public Key Infrastructure

Security requires trust, and public key cryptography by itself does nothing to establish trust. SSL addresses the issue of trust using Public Key Infrastructure (PKI), which binds public keys to identities using certificates. A certificate *issuer* creates a certificate for an entity, called the *subject*. The subject is often a person, but it may also be a computer or a specific application. The subject's identity is represented by a *distinguished name*, which includes information such as the subject's name, organization and location. A certificate alone is not sufficient to

1. IceSSL is available for C++, Java and .NET applications. Python, PHP and Ruby applications can use IceSSL for C++ via configuration.

establish the subject's identity, however, as anyone can create a certificate for a particular distinguished name.

In order to authenticate a certificate, we need a third-party to guarantee that the certificate belongs to the subject described by the distinguished name. This third party, called a Certificate Authority (CA), expresses this guarantee by using its own private key to sign the subject's certificate. Combining the CA's certificate with the subject's certificate forms a *certificate chain* that provides SSL with most of the information it needs to authenticate the remote peer. In many cases, the chain contains only the aforementioned two certificates, but it is also possible for the chain to be longer when the *root CA* issues a certificate that the subject may use to sign other certificates. Regardless of the length of the chain, this scheme can only work if we trust that the root CA has sufficiently verified the identity of the subject before issuing the certificate.

An implementation of the SSL protocol also needs to know which root CAs we trust. An application supplies that information as a list of certificates representing the trusted root CAs. With that list in hand, the SSL implementation authenticates a peer by obtaining the peer's certificate chain and examining it carefully for validity. If we view the chain as a hierarchy with the root CA certificate at the top and the peer's certificate at the bottom, we can describe SSL's validation activities as follows:

- The root CA certificate must be self-signed and be present among the application's trusted CA certificates.
- All other certificates in the chain must be signed by the one immediately preceding it.
- The certificates must not be expired or revoked.

These tests certify that the chain is valid, but applications often require the chain to undergo more intensive scrutiny (see Section 41.4.6).

Commercial CAs exist to supply organizations with a reliable source of certificates, but in many cases a private CA is completely sufficient. You can create and manage your CA using freely-available tools, and in fact Ice includes a collection of utilities that simplify this process (see Section 41.7).

Depending on your implementation language, it may also possible to avoid the use of certificates altogether; encryption is still used to obscure the session traffic, but the benefits of authentication are sacrificed in favor of reduced complexity and administration.

For more information on PKI, see [5].

41.2.3 Requirements

Integrating IceSSL into your application often requires no changes to your source code, but does involve the following administrative tasks:

- creating a public key infrastructure (if necessary)
- configuring the IceSSL plug-in
- modifying your application's configuration to install the IceSSL plug-in and use secure connections

The remainder of this chapter discusses plug-in configuration and programming.

41.3 Using IceSSL

Incorporating IceSSL into your application requires installing the plug-in, configuring it according to your security requirements, and creating SSL endpoints.

41.3.1 Installing IceSSL

Ice supports a generic plug-in facility that allows extensions (such as IceSSL) to be installed dynamically without changing the application source code. The `Ice.Plugin` property (see Appendix D) provides language-specific information that enables the Ice run time to install a plug-in.

C++ Applications

The executable code for the IceSSL C++ plug-in resides in a shared library on Unix and a dynamic link library (DLL) on Windows. The format for the `Ice.Plugin` property is shown below:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
```

The last component of the property name (`IceSSL`) becomes the plug-in's official identifier for configuration purposes, but the IceSSL plug-in requires its identifier to be `IceSSL`. The property value `IceSSL:createIceSSL` is sufficient to allow the Ice run time to locate the IceSSL library (on both Unix and Windows) and initialize the plug-in. The only requirement is that the library reside in a directory that appears in the shared library path (`LD_LIBRARY_PATH` on most Unix platforms, `PATH` on Windows).

Additional configuration properties are usually necessary as well; see Section 41.4.1 for more information.

Java Applications

The format for the `Ice.Plugin` property is shown below:

```
Ice.Plugin.IceSSL=IceSSL.PluginFactory
```

The last component of the property name (`IceSSL`) becomes the plug-in's official identifier for configuration purposes, but the IceSSL plug-in requires its identifier to be `IceSSL`. The property value `IceSSL.PluginFactory` is the name of a class that allows the Ice run time to initialize the plug-in. The IceSSL classes are included in `Ice.jar`, therefore no additional changes to your `CLASSPATH` are necessary.

Additional configuration properties are usually necessary as well; see Section 41.4.2 for more information.

.NET Applications

The format for the `Ice.Plugin` property is shown below:

```
Ice.Plugin.IceSSL=C:/Ice/bin/IceSSL.dll:IceSSL.PluginFactory
```

The last component of the property name (`IceSSL`) becomes the plug-in's official identifier for configuration purposes, but the IceSSL plug-in requires its identifier to be `IceSSL`. The property value contains the file name of the IceSSL assembly as well as the name of a class, `IceSSL.PluginFactory`, that allows the Ice run time to initialize the plug-in. As described in Appendix D, you may also specify the full assembly name instead of the file name in an `Ice.Plugin` property.

Additional configuration properties are usually necessary as well; see Section 41.4.3 for more information.

Ice Touch Applications

The IceSSL plug-in is included in the Ice Touch run time and installed automatically, therefore it is not necessary to explicitly load it. See Section 41.4.4 for more information on configuring IceSSL in Ice Touch applications.

41.3.2 Creating SSL Endpoints

Installing the IceSSL plug-in enables you to use a new protocol, `ssl`, in your endpoints. For example, the following endpoint list creates a TCP endpoint, an SSL endpoint, and a UDP endpoint:

```
MyAdapter.Endpoints=tcp -p 4063:ssl -p 4064:udp -p 4063
```

As this example demonstrates, it is possible for a UDP endpoint to use the same port number as a TCP or SSL endpoint, because UDP is a different protocol and therefore has its own set of ports. It is not possible for a TCP endpoint and an SSL endpoint to use the same port number, because SSL is essentially a layer over TCP.

Using SSL in stringified proxies is equally straightforward:

```
MyProxy=MyObject:tcp -p 4063:ssl -p 4064:udp -p 4063
```

For more information on proxies and endpoints, see Appendix E.

41.3.3 Security Considerations

Defining an object adapter's endpoints to use multiple protocols, as shown in the example in Section 41.3.2, has obvious security implications. If your intent is to use SSL to protect session traffic and/or restrict access to the server, then you should only define SSL endpoints.

There can be situations, however, in which insecure endpoint protocols are advantageous. Figure 41.1 illustrates an environment in which TCP endpoints are allowed behind the firewall, but external clients are required to use SSL.

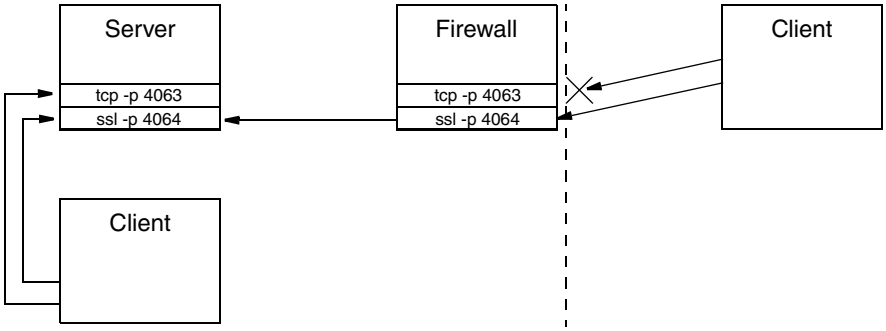


Figure 41.1. An application of multiple protocol endpoints.

The firewall in Figure 41.1 is configured to block external access to TCP port 4063 and to forward connections to port 4064 to the server machine.

One reason for using TCP behind the firewall is that it is more efficient than SSL and requires less administrative work. Of course, this scenario assumes that internal clients can be trusted, which is not true in many environments.

For more information on using SSL in complex network architectures, see Chapter 42.

41.4 Configuring IceSSL

After installing IceSSL as described in Section 41.3.1, an application typically needs to define a handful of additional properties to configure settings such as the location of certificate and key files. This section provides an introduction to configuring the plug-in for each of the supported language mappings. For a complete listing of the IceSSL configuration properties, see Appendix D.

41.4.1 C++

Our first example shows the properties that are sufficient in many situations:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=pubkey.pem
IceSSL.KeyFile=privkey.pem
IceSSL.CertAuthFile=ca.pem
IceSSL.Password=password
```

The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your certificate and key files. The three properties that follow it define the files containing the program's certificate, private key, and trusted CA certificate, respectively. This example assumes the files contain RSA keys, and IceSSL requires the files to use the Privacy Enhanced Mail (PEM) encoding. Finally, the `IceSSL.Password` property specifies the password of the private key.

Note that it is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. See Section 41.6.1 for alternate ways to supply IceSSL with a password.

DSA Example

If you used DSA to generate your keys, one additional property is necessary:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=pubkey_dsa.pem
IceSSL.KeyFile=privkey_dsa.pem
IceSSL.CertAuthFile=ca.pem
IceSSL.Password=password
IceSSL.Ciphers=DEFAULT:DSS
```

The `IceSSL.Ciphers` property adds support for DSS authentication to the plug-in's default set of ciphersuites. See Appendix D for more information on this property.

RSA and DSA Example

It is also possible to specify certificates and keys for both RSA and DSA by including two filenames in the `IceSSL.CertFile` and `IceSSL.KeyFile` properties. The filenames must be separated using the platform's path separator. The example below demonstrates the Unix separator (a colon):

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=pubkey_rsa.pem:pubkey_dsa.pem
IceSSL.KeyFile=privkey_rsa.pem:privkey_dsa.pem
IceSSL.CertAuthFile=ca.pem
IceSSL.Password=password
IceSSL.Ciphers=DEFAULT:DSS
```

On Windows, you would use a semicolon to separate the filenames.

ADH Example

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The configuration properties shown below demonstrate how to use ADH:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.Ciphers=ADH
IceSSL.VerifyPeer=0
```

The `IceSSL.Ciphers` property enables support for ADH, which is disabled by default.

The `IceSSL.VerifyPeer` property changes the plug-in's default behavior with respect to certificate verification. Without this setting, IceSSL rejects a connection if the peer does not supply a certificate (as is the case with ADH).

See Appendix D for more information on these properties.

41.4.2 Java

IceSSL uses Java's native format for storing keys and certificates: the keystore.

A keystore is represented as a file containing key pairs and associated certificates, and is usually administered using the **`keytool`** utility supplied with the Java run time. Keystores serve two roles in Java's SSL architecture:

1. A keystore containing a key pair identifies the peer and is usually closely guarded.
2. A keystore containing public certificates represents the identities of trusted peers and can be freely shared. These keystores are also referred to as "trust-stores" when they are used to store only trusted certificate chains.

A single keystore file can fulfill both of these purposes.

Java supports a pluggable architecture for keystore implementations in which a system property selects a particular implementation as the default keystore type. IceSSL uses the default keystore type unless otherwise specified.

A password is assigned to each key pair in a keystore, as well as to the keystore itself. IceSSL must be provided with the password for the key pair, but the keystore password is optional. If a keystore password is specified, it is used only to verify the keystore's integrity. IceSSL requires that all of the key pairs in a keystore have the same password.

Our first example shows the properties that are sufficient in many situations:

```
Ice.Plugin.IceSSL=IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.Keystore=keys.jks
IceSSL.Truststore=ca.jks
IceSSL.Password=password
```

IceSSL resolves the filenames defined in its configuration properties as follows:

1. Attempt to open the file as a class loader resource. This is especially useful for deploying applications with special security restrictions, such as applets.
2. Attempt to open the file in the local file system.
3. If `IceSSL.DefaultDir` is defined, prepend its value and try steps 1 and 2 again. The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your keystore and truststore files.

The `IceSSL.Password` property specifies the password of the key pair. Note that it is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. Section 41.6.1 describes a more secure way to configure the plug-in.

DSA Example

Java supports both RSA and DSA keys. No additional properties are necessary to use DSA:

```
Ice.Plugin.IceSSL=IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.Keystore=dsakeys.jks
IceSSL.Truststore=ca.jks
IceSSL.Password=password
```

ADH Example

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The configuration properties shown below demonstrate how to use ADH:

```
Ice.Plugin.IceSSL=IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.Ciphers=NONE (DH_anon)
IceSSL.VerifyPeer=0
```

The `IceSSL.Ciphers` property enables support for ADH, which is disabled by default.

The `IceSSL.VerifyPeer` property changes the plug-in's default behavior with respect to certificate verification. Without this setting, IceSSL rejects a connection if the peer does not supply a certificate (as is the case with ADH).

See Appendix D for more information on these properties.

41.4.3 .NET

The Common Language Runtime (CLR) in .NET uses certificate stores as the persistent repositories of certificates and keys. Furthermore, the CLR maintains two distinct sets of certificate stores, one for the current user and another for the local machine. Although it is possible to load a certificate and its corresponding private key from a regular file, the CLR requires trusted CA certificates to reside in an appropriate certificate store.

Managing Certificates with the Microsoft Management Console

On Windows, you can use the Microsoft Management Console (MMC) to browse the contents of the various certificate stores. To start the console, run `MMC.EXE` from a command window, or choose Run from the Start menu and enter `MMC.EXE`.

Once the console is running, you need to install the Certificates "snap-in" by choosing Add/Remove Snap-in from the File menu. Click the Add button, choose Certificates in the popup window and click Add. If you wish to manage certificates for the current user, select My Current Account and click Finish. To manage certificates for the local computer, select Computer Account and click Next, then select Local Computer and click Finish.

When you have finished adding snap-ins, close the Add Standalone Snap-in window and click OK on the Add/Remove Snap-in window. Your Console Root window now contains a tree structure that you can expand to view the available certificate stores. If you have a certificate in a file that you want to add to a store, click on the desired store, then open the Action menu and select All Tasks/Import.

Using Certificate Files

Our first example demonstrates how to configure IceSSL with a file that contains the program's certificate and key:

```
Ice.Plugin.IceSSL=IceSSL.dll:IceSSL.PluginFactory
IceSSL.DefaultDir=/opt/certs
IceSSL.CertFile=cert.pfx
IceSSL.Password=password
```

The `IceSSL.DefaultDir` property is a convenient way to specify the default location of your certificate file. This file must use the Personal Information Exchange (PFX, also known as PKCS#12) format and contain both a certificate and its corresponding private key. The `IceSSL.Password` property specifies the password used to secure the file.

Note that it is a security risk to define a password in a plain text file, such as an Ice configuration file, because anyone who can gain read access to your configuration file can obtain your password. More secure methods of configuring the plug-in are described in the next section and in Section 41.6.1.

This configuration assumes that any trusted CA certificates necessary to authenticate the program's peers are already installed in an appropriate certificate store. You may also use a configuration property to automatically import a certificate from a file, as described in a subsequent section below.

Using Certificate Stores

If the program's certificate and private key are already installed in a certificate store, you can select it using the `IceSSL.FindCert` configuration property as shown in the following example:

```
Ice.Plugin.IceSSL=IceSSL.dll:IceSSL.PluginFactory
IceSSL.FindCert.LocalMachine.My=subject:"Quote Server"
```

An `IceSSL.FindCert` property executes a query in a particular certificate store and selects all of the certificates that match the given criteria. In the example above, the location of the certificate store is `LocalMachine`, and the store's name is `My`. When using MMC to browse the certificate stores, this specification is equivalent to the store "Personal" in the location "Certificates (Local Computer)."

The other legal value for the location component of the property name is `CurrentUser`. Table 41.1 shows the valid values for the store name component and their equivalents in MMC.

Table 41.1. Certificate store names.

Property Name	MMC Name
`AddressBook`	Other People
`AuthRoot`	Third-Party Root Certification Authorities
`CertificateAuthority`	Intermediate Certification Authorities
`Disallowed`	Untrusted Certificates
`My`	Personal
`Root`	Trusted Root Certification Authorities
`TrustedPeople`	Trusted People
`TrustedPublisher`	Trusted Publishers

The search criteria consists of *name*:*value* pairs that perform case-insensitive comparisons against the fields of each certificate in the specified store, and the special property value `*` selects every certificate in the store. Typically, however,

the criteria should select a single certificate. In a server, IceSSL must supply the CLR with the certificate that represents the server's identity; if a configuration matches several certificates, IceSSL chooses one (in an undefined manner) and logs a warning to notify you of the situation.

Selecting a certificate from a store is more secure than using a certificate file via the `IceSSL.CertFile` property because it is not necessary to specify a plain-text password. MMC prompts you for the password when initially importing a certificate into a store, so the password is not required when an application uses that certificate to identify itself.

For complete details about the syntax of the `IceSSL.FindCert` property, see its description in Appendix D.

Importing Certificates

IceSSL can be configured to import a certificate into a particular store. The Ice demos and test suites use this capability to ensure that the CA certificate is present, which avoids the need for a user to manually import the certificate using MMC before she could use the Ice sample programs and tests.

The `IceSSL.ImportCert` property uses the same format for its name as the `IceSSL.FindCert` property described above, in that the certificate store's location and name are part of the property name:

```
IceSSL.ImportCert.LocalMachine.AuthRoot=cacert.pem
```

The property's value is the name of a certificate file and an optional password. If a file is protected with a password, append the password to the property value using a semicolon as the separator. IceSSL uses the value of `IceSSL.DefaultDir` to complete the file name if necessary. The CLR accepts a number of encoding formats for the certificate, including PEM, DER and PFX.

The store name should be chosen with care. When installing a trusted CA certificate, authentication succeeds only when the certificate is installed into one of the following stores:

- `LocalMachine.Root`
- `LocalMachine.AuthRoot`
- `CurrentUser.Root`

Note that administrative privileges are required when installing a certificate into a `LocalMachine` store.

If you specify a store name other than those listed in Table 41.1, IceSSL creates a new store with the given name and adds the certificate to it. Once

installed in the specified store, the application (or the user) is responsible for removing the certificate when it is no longer necessary.

41.4.4 Ice Touch

In Ice Touch, certificate files are loaded from the application's resource bundle. If the application's target platform is Mac OS, certificate files can also be loaded directly from the file system. Consider the following properties:

```
IceSSL.DefaultDir=certs
IceSSL.CertAuthFile=cacert.der
IceSSL.CertFile=cert.pfx
IceSSL.Password=password
```

The `IceSSL.DefaultDir` property is a convenient way to specify the location of your certificate files. Defining `IceSSL.DefaultDir` means IceSSL searches for certificate files relative to the specified directory. For the properties in the example above, IceSSL composes the pathnames `certs/cacert.der` and `certs/cert.pfx`. If `IceSSL.DefaultDir` is not defined, IceSSL uses the certificate file pathnames exactly as they are supplied.

As mentioned earlier, IceSSL has different semantics for locating certificate files depending on the target platform. For the iPhone and iPhone simulator, IceSSL attempts to open a certificate file in the application's resource bundle as `Resources/`*`DefaultDir`*`/`*`file`* if `IceSSL.DefaultDir` is defined, or as simply `Resources/`*`file`* otherwise. If the target platform is Mac OS and the certificate file cannot be found in the resource bundle, IceSSL also attempts to open the file in the file system as *`DefaultDir`*`/`*`file`* if a default directory is specified, or as simply *`file`* otherwise.

IceSSL requires that the CA certificate file specified by `IceSSL.CertAuthFile` use the DER format. The certificate file in `IceSSL.CertFile` must use the Personal Information Exchange (PFX, also known as PKCS#12) format and contain both a certificate and its corresponding private key. The `IceSSL.Password` property specifies the password used to secure the certificate file.

Keychains

IceSSL imports the certificate specified by `IceSSL.CertFile` into a keychain. IceSSL uses the `login` keychain by default unless you choose a different one by defining the `IceSSL.Keychain` property:

```
IceSSL.Keychain=Test Keychain
```

The `login` keychain is the user's default keychain, which is normally unlocked after logging into the system. IceSSL does not usually require a password to import a certificate into the `login` keychain. However, if your `login` keychain is not unlocked automatically, or if you have selected a different keychain, you can supply a password using the `IceSSL.KeychainPassword` property:

```
IceSSL.KeychainPassword=password
```

41.4.5 Ciphersuites

A ciphersuite represents a particular combination of encryption, authentication and hashing algorithms. The IceSSL plug-ins for C++ and Java allow you to configure the ciphersuites that their underlying SSL engines are allowed to negotiate during handshaking with a peer. By default, IceSSL uses the underlying engine's default ciphersuites, but you can define a property to customize the list as we demonstrated earlier in this section with the ADH examples. Normally the default configuration is chosen to eliminate relatively insecure ciphersuites such as ADH, which is the reason it must be explicitly enabled.

Configuring Ciphersuites in C++

The value of the `IceSSL.Ciphers` property is given directly to the low-level OpenSSL library, on which IceSSL is based. Therefore, OpenSSL determines the allowable ciphersuites, which in turn depend on how the OpenSSL distribution was compiled. You can obtain a complete list of the supported ciphersuites using the **`openssl`** command:

```
$ openssl ciphers
```

This command will likely generate a long list. To simplify the selection process, OpenSSL supports several classes of ciphers, as shown in Table 41.2.

Table 41.2. Cipher classes.

Class	Description
ALL	All possible combinations.
ADH	Anonymous ciphers.
LOW	Low bit-strength ciphers.
EXP	Export-crippled ciphers.

Classes and ciphers can be excluded by prefixing them with an exclamation point. The special keyword `@STRENGTH` sorts the cipher list in order of their strength, so that SSL gives preference to the more secure ciphers when negotiating a cipher suite. The `@STRENGTH` keyword must be the last element in the list.

For example, here is a good value for the `cipherlist` attribute:

```
ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH
```

This value excludes the ciphers with low bit strength and known problems, and orders the remaining ciphers according to their strength.

Note that no warning is given if an unrecognized cipher is specified.

Configuring Ciphersuites in Java

IceSSL for Java interprets the value of `IceSSL.Ciphers` as a sequence of expressions that filter the selected ciphersuites using name and pattern matching. If the property is not defined, the Java security provider's default ciphersuites are used. Table 41.3 defines the valid expressions that may appear in the property value.

Table 41.3. Ciphersuite filter expressions.

Expression	Description
`NONE`	Disables all ciphersuites. If specified, it must appear first.
`ALL`	Enables all supported ciphersuites. If specified, it must appear first. This expression should be used with caution, as it may enable low-security ciphersuites
`NAME`	Enables the ciphersuite matching the given name.
`!`*`NAME`*	Disables the ciphersuite matching the given name.
`(`*`EXP`*`)`	Enables ciphersuites whose names contain the regular expression *`EXP`*.
`!(`*`EXP`*`)`	Disables ciphersuites whose names contain the regular expression *`EXP`*.

To determine the set of enabled ciphersuites, the plug-in begins with a list of ciphersuite names containing the default set as determined by the security

provider. The expressions in the property value add and remove ciphersuites from this list and are evaluated in the order of appearance. For example, consider the following property definition:

```
IceSSL.Ciphers=NONE (RSA.*AES) !(EXPORT)
```

The expressions in this property have the following effects:

- `NONE` clears the list of enabled ciphersuites.
- `(RSA.*AES)` is a regular expression that enables ciphersuites whose names contain the string "RSA" followed by "AES", meaning ciphersuites using RSA authentication and AES encryption.
- `!(EXPORT)` is a regular expression that disables any of the selected ciphersuites whose names contain the string "EXPORT", meaning ciphersuites having export-quality strength.

As another example, this property adds anonymous Diffie-Hellman to the default set of ciphersuites and disables export ciphersuites:

```
IceSSL.Client.Ciphers=(DH_anon) !(EXPORT)
```

Finally, this example selects only one ciphersuite:

```
IceSSL.Client.Ciphers=NONE SSL_RSA_WITH_RC4_128_SHA
```

41.4.6 Trust

As described in Section 41.2.2, declaring that you trust a certificate authority implies that you trust any peer whose certificate was signed directly or indirectly by that certificate authority. It is necessary to use this broad definition of trust in some applications, such as a public Web server. In more controlled environments, it is a good idea to restrict access as much as possible, and IceSSL provides a number of ways for you to do that.

Trusted Peers

After the low-level SSL engine has completed its authentication process, IceSSL can be configured to take additional steps to verify whether a peer should be trusted. The `IceSSL.TrustOnly` family of properties (see Appendix D) defines a collection of acceptance and rejection filters that IceSSL applies to the distinguished name of a peer's certificate in order to determine whether to allow the connection to proceed. IceSSL permits the connection if the peer's distinguished name matches any of the acceptance filters and does not match any of the rejection filters.

A distinguished name uniquely identifies a person or entity and is generally represented in the following textual form:

```
C=US, ST=Florida, L=Palm Beach Gardens, O="ZeroC, Inc.",
OU=Servers, CN=Quote Server
```

Suppose we are configuring a client to communicate with the server whose distinguished name is shown above. If we know that the client is allowed to communicate only with this server, we can enforce this rule using the following property:

```
IceSSL.TrustOnly=O="ZeroC, Inc.", OU=Servers, CN=Quote Server
```

With this property in place, IceSSL allows a connection to proceed only if the distinguished name in the server's certificate matches this filter. The property may contain multiple filters, separated by semicolons, if the client needs to communicate with more than one server. Additional variations of the property are also supported, as described in Appendix D.

If the `IceSSL.TrustOnly` properties do not provide the selectivity you require, the next step is to install a custom certificate verifier (see Section 41.5).

Verification Depth

In order to authenticate a peer, SSL obtains the peer's certificate chain, which includes the peer's certificate as well as that of the root CA. SSL verifies that each certificate in the chain is valid, but there still remains a subtle security risk. Suppose that we have identified a trusted root CA (via its certificate), and a peer has supplied a valid certificate chain signed by our trusted root CA. It is possible for an attacker to obtain a special signing certificate that is signed by our root CA and therefore trusted implicitly. The attacker can use this certificate to sign fraudulent certificates with the goal of masquerading as a trusted peer, presumably for some nefarious purpose.

We could use the `IceSSL.TrustOnly` properties described above in an attempt to defend against such an attack. However, the attacker could easily manufacture a certificate containing a distinguished name that satisfies the trust properties.

If you know that all trusted peers present certificate chains of a certain length, set the property `IceSSL.VerifyDepthMax` so that IceSSL automatically rejects longer chains. The default value of this property is two, therefore you may need to set it to a larger value if you expect peers to present longer chains.

In situations where you cannot make assumptions about the length of a peer's certificate chain, yet you still want to examine the chain before allowing the connection, you should install a custom certificate verifier (see Section 41.5).

41.4.7 Secure Proxies

Proxies may contain any combination of secure and insecure endpoints. An application that requires secure communication can guarantee that proxies it manufactures itself, such as those created by calling `stringToProxy`, contain only secure endpoints. However, the application cannot make the same assumption about proxies received as the result of a remote invocation.

The simplest way to guarantee that all proxies use only secure endpoints is to define the following configuration property:

```
Ice.Override.Secure=1
```

Setting this property is equivalent to invoking `ice_secure(true)` on every proxy. When enabled, attempting to establish a connection using a proxy that does not contain a secure endpoint results in `NoEndpointException`.

If you want the default behavior of proxies to give precedence to secure endpoints, you can set this property instead:

```
Ice.Default.PreferSecure=1
```

Note that proxies may still attempt to establish connections to insecure endpoints, but they try all secure endpoints first. This is equivalent to invoking `ice_preferSecure(true)` on a proxy.

Refer to Section 32.11.2 for more information on these proxy methods, and Section 32.11.4 for details on the connection establishment process used by the Ice run time.

41.4.8 Diagnostics

You can use two configuration properties to obtain more information about the plug-in's activities. Setting `IceSSL.Trace.Security=1` enables the plug-in's diagnostic output, which includes a variety of messages regarding events such as ciphersuite selection, peer verification and trust evaluation. The other property, `Ice.Trace.Network`, determines how much information is logged about network events such as connections and packets. Note that the output generated by `Ice.Trace.Network` also includes other transports such as TCP and UDP.

System Logging in Java

In Java, you can use a system property that displays a great deal of information about SSL certificates and connections, including the ciphersuite that is selected

for use by each connection. For example, the following command sets the system property that activates the diagnostics:

```
$ java -Djavax.net.debug=ssl MyProgram
```

System Logging in .NET

Enabling additional tracing output in .NET requires the creation of an XML file such as the one shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    <system.diagnostics>
        <trace autoflush="true"/>
        <sources>
            <source name="System.Net">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
            <source name="System.Net.Sockets">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
            <source name="System.Net.Cache">
                <listeners>
                    <add name="System.Net"/>
                </listeners>
            </source>
        </sources>
        <sharedListeners>
            <add
            name="System.Net"
            type="System.Diagnostics.TextWriterTraceListener"
            initializeData="trace.txt"
            />
        </sharedListeners>
        <switches>
            <add name="System.Net" value="Verbose"/>
            <add name="System.Net.Sockets" value="Verbose"/>

            <add name="System.Net.Cache" value="Verbose"/>
        </switches>
    </system.diagnostics>
</configuration>
```

In this example, the output is stored in the file `trace.txt`. To activate tracing, give the XML file the same name as your executable with a `.config` extension as in `server.exe.config`, and place it in the same directory as the executable.

41.5 Programming with IceSSL

The configuration properties described in Section 41.4 are flexible enough to satisfy the requirements of many applications, and IceSSL supports a public API that offers even more functionality for those applications that need it.

41.5.1 C++

This section describes the C++ API for the IceSSL plug-in.

The `Plugin` Interface

Applications can interact directly with the IceSSL plug-in using the native C++ class `IceSSL::Plugin`. A reference to a `Plugin` object must be obtained from the communicator in which the plug-in is installed:

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr =
    communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin =
    IceSSL::PluginPtr::dynamicCast(plugin);
```

The `Plugin` class supports the following methods:

```
namespace IceSSL
{
class Plugin : public Ice::Plugin
{
public:
    virtual void setContext(SSL_CTX*) = 0;
    virtual SSL_CTX* getContext() = 0;

    virtual void setCertificateVerifier(
        const CertificateVerifierPtr&) = 0;

    virtual void setPasswordPrompt(
```

```
        const PasswordPromptPtr&) = 0;
};
typedef IceUtil::Handle<Plugin> PluginPtr;
}
```

The `setContext` and `getContext` methods are rarely used in practice; see Section 41.6 for more information. The `setCertificateVerifier` method installs a custom certificate verifier object that the plug-in invokes for each new connection. The `setPasswordPrompt` method provides an alternate way to supply IceSSL with passwords, as discussed in Section 41.6.

Obtaining Connection Information

You can obtain information about any SSL connection using the `Ice::Connection::getInfo` operation (see Section 36.5). It returns an `IceSSL::NativeConnectionInfo` class instance that derives from the Slice class `IceSSL::ConnectionInfo`. The Slice base class is defined as follows:

```
module Ice {
    local class ConnectionInfo {
        bool incoming;
        string adapterName;
    };

    local class IPConnectionInfo extends ConnectionInfo {
        string localAddress;
        int localPort;
        string remoteAddress;
        int remotePort;
    };
};

module IceSSL {
    local class ConnectionInfo extends Ice::IPConnectionInfo {
        string cipher;
        Ice::StringSeq certs;
    };
};
```

In turn, the C++ class `NativeConnectionInfo` is defined as follows:

```
class NativeConnectionInfo : public ConnectionInfo {
public:
    std::vector<CertificatePtr> nativeCerts;
};

typedef IceUtil::Handle<NativeConnectionInfo>
                            NativeConnectionInfoPtr;
```

Installing a Certificate Verifier

A new connection undergoes a series of verification steps before an application is allowed to use it. The low-level SSL engine executes the validation procedures described in Section 41.2.2. Assuming the certificate chain is successfully validated, IceSSL performs additional verification as directed by its configuration properties (see Section 41.4.6). Finally, if a certificate verifier is installed, IceSSL invokes it to provide the application with an opportunity to decide whether to allow the connection to proceed.

The `CertificateVerifier` interface has only one method:

```
namespace IceSSL
{
class CertificateVerifier : public IceUtil::Shared
{
public:

    virtual bool verify(const NativeConnectionInfoPtr&) = 0;
};
typedef IceUtil::Handle<CertificateVerifier>
    CertificateVerifierPtr;
}
```

IceSSL rejects the connection if `verify` returns `false`, and allows it to proceed if the method returns `true`. The `verify` method receives a `ConnectionInfo` object that describes the connection's attributes.

The `nativeCerts` member is a vector of certificates representing the peer's certificate chain. The vector is structured so that the first element is the peer's certificate, followed by its signing certificates in the order they appear in the chain, with the root CA certificate as the last element. The vector is empty if the peer did not present a certificate chain.

The `cipher` member is a description of the ciphersuite that SSL negotiated for this connection. The local and remote address information is provided in `localAddress` and `remoteAddress`[2], respectively. The `incoming` member indicates whether the connection is inbound (a server connection) or outbound (a client

connection). Finally, if `incoming` is true, the `adapterName` member supplies the name of the object adapter that hosts the endpoint.

The following class is a simple implementation of a certificate verifier:

```
class Verifier : public IceSSL::CertificateVerifier
{
public:

    bool verify(const IceSSL::NativeConnectionInfo& info)
    {
        if (!info.nativeCerts.empty())
        {
            string dn = info.nativeCerts[0].getIssuerDN();
            transform(dn.begin(), dn.end(), dn.begin(),
                      ::tolower);
            if (dn.find("zeroc") != string::npos)
            {
                return true;
            }
        }
        return false;
    }
}
```

In this example, the verifier rejects the connection unless the string `zeroc` is present in the issuer's distinguished name of the peer's certificate. In a more realistic implementation, the application is likely to perform detailed inspection of the certificate chain.

Installing the verifier is a simple matter of calling `setCertificateVerifier` on the plug-in interface:

```
IceSSL::PluginPtr sslPlugin = // ...
sslPlugin->setCertificateVerifier(new Verifier);
```

You should install the verifier before any SSL connections are established.

See Section 41.6.2 for more information on installing a certificate verifier.

Certificates

The `ConnectionInfo` class contains a vector of `Certificate` objects representing the peer's certificate chain. `Certificate` is a reference-counted

2. A bug in Windows XP prevents IceSSL from obtaining the remote address information when using IPv6.

convenience class that hides the complexity of the underlying OpenSSL API. Its methods are inspired by the Java class `X509Certificate`:

```
namespace IceSSL
{
class Certificate : public IceUtil::Shared
{
public:

    Certificate(X509*);

    static CertificatePtr load(const string&);
    static CertificatePtr decode(const string&);

    bool operator==(const Certificate&) const;
    bool operator!=(const Certificate&) const;

    PublicKeyPtr getPublicKey() const;

    bool verify(const PublicKeyPtr&) const;

    string encode() const;

    bool checkValidity() const;
    bool checkValidity(const IceUtil::Time&) const;

    IceUtil::Time getNotAfter() const;
    IceUtil::Time getNotBefore() const;

    string getSerialNumber() const;

    DistinguishedName getIssuerDN() const;
    vector<pair<int, string> > getIssuerAlternativeNames();

    DistinguishedName getSubjectDN() const;
    vector<pair<int, string> > getSubjectAlternativeNames();

    int getVersion() const;

    string toString() const;

    X509* getCert() const;
};
typedef IceUtil::Handle<Certificate> CertificatePtr;
}
```

The more commonly-used methods are described below; refer to the documentation in `IceSSL/Plugin.h` for information on the methods that are not covered.

The static method `load` creates a certificate from the contents of a PEM-encoded file. If an error occurs, the function raises `IceSSL::CertificateReadException`; the reason member provides a description of the problem.

Use `decode` to obtain a certificate from a PEM-encoded string representing a certificate. The caller must be prepared to catch `IceSSL::CertificateEncodingException` if `decode` fails; the reason member provides a description of the problem.

The `encode` method creates a PEM-encoded string that represents the certificate. The return value can later be passed to `decode` to recreate the certificate.

The `checkValidity` methods determine whether the certificate is valid. The overloading with no arguments returns true if the certificate is valid at the current time; the other overloading accepts an `IceUtil::Time` object and returns true if the certificate is valid at the given time. See Appendix F.12 for more information on `IceUtil::Time`.

The `getNotAfter` and `getNotBefore` methods return instances of `IceUtil::Time` that define the certificate's valid period.

The methods `getIssuerDN` and `getSubjectDN` supply the distinguished names of the certificate's issuer (i.e., the CA that signed the certificate) and subject (i.e., the person or entity to which the certificate was issued). The methods return instances of the class `IceSSL::DistinguishedName`, another convenience class that is described in the next section.

Finally, the `toString` method returns a human-readable string describing the certificate.

Distinguished Names

X.509 certificates use a distinguished name to identify a person or entity. The name is an ordered sequence of relative distinguished names that supply values for fields such as common name, organization, state, and country. Distinguished names are commonly displayed in stringified form according to the rules specified by RFC 2253, as shown in the following example:

```
C=US, ST=Florida, L=Palm Beach Gardens, O="ZeroC, Inc.",
OU=Servers, CN=Quote Server
```

`DistinguishedName` is a convenience class provided by IceSSL to simplify the tasks of parsing, formatting and comparing distinguished names.

```
namespace IceSSL
{
class DistinguishedName
{
public:

    DistinguishedName(const std::string&);
    DistinguishedName(
        const std::list<std::pair<std::string, std::string> >&);

    bool operator==(const DistinguishedName&) const;
    bool operator!=(const DistinguishedName&) const;
    bool operator<(const DistinguishedName&) const;

    bool match(const DistinguishedName&) const;

    operator std::string() const;
};
}
```

The first overloaded constructor accepts a string argument representing a distinguished name encoded using the rules set forth in RFC 2253. The new `DistinguishedName` instance preserves the order of the relative distinguished names in the string. The caller must be prepared to catch `IceSSL::ParseException` if an error occurs during parsing.

The second overloaded constructor requires a list of type–value pairs representing the relative distinguished names. The new `DistinguishedName` instance preserves the order of the relative distinguished names in the list.

The overloaded operator functions `operator==`, `operator!=`, and `operator<` perform an exact match of distinguished names in which the order of the relative distinguished names is important. For two distinguished names to be equal, they must have the same relative distinguished names in the same order.

The `match` function performs a partial comparison that does not consider the order of relative distinguished names. If `N1` and `N2` are instances of `DistinguishedName`, `N1.match(N2)` returns true if all of the relative distinguished names in `N2` are present in `N1`.

Finally, the string conversion operator encodes the distinguished name in the format described by RFC 2253.

41.5.2 Java

This section describes the Java API for the IceSSL plug-in.

The `Plugin` Interface

Applications can interact directly with the IceSSL plug-in using the native Java interface `IceSSL.Plugin`. A reference to a `Plugin` object must be obtained from the communicator in which the plug-in is installed:

```
Ice.Communicator comm = // ...
Ice.PluginManager pluginMgr = comm.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
```

The `Plugin` interface supports the following methods:

```
package IceSSL;

public interface Plugin extends Ice.Plugin
{
    void setContext(javax.net.ssl.SSLContext context);
    javax.net.ssl.SSLContext getContext();

    void setCertificateVerifier(CertificateVerifier verifier);
    CertificateVerifier getCertificateVerifier();

    void setPasswordCallback(PasswordCallback callback);
    PasswordCallback getPasswordCallback();

    void setKeystoreStream(java.io.InputStream stream);

    void setTruststoreStream(java.io.InputStream stream);

    void addSeedStream(java.io.InputStream stream);
}
```

The methods are summarized below:

- `setContext`
  `getContext`

  These methods are rarely used in practice; see Section 41.6 for more information.

- `setCertificateVerifier`
  `getCertificateVerifier`

  These methods install and retrieve a custom certificate verifier object that the plug-in invokes for each new connection. `getCertificateVerifier` returns null if a verifier has not been set.

- `setPasswordCallback`
  `getPasswordCallback`

  These methods install and retrieve a password callback object that supplies IceSSL with passwords, as discussed in Section 41.6. `getPasswordCallback` returns null if a callback has not been set.

- `setKeystoreStream`

  Supplies an input stream for a keystore containing the key pair. The `IceSSL.Keystore` property is ignored if this method is called with a non-null value. You may supply the same input stream object to this method and to `setTruststoreStream` if your keystore contains your key pair as well as your trusted CA certificates.

- `setTruststoreStream`

  Supplies an input stream for a truststore containing your trusted CA certificates. The `IceSSL.Truststore` property is ignored if this method is called with a non-null value. You may supply the same input stream object to this method and to `setKeystoreStream` if your keystore contains your key pair as well as your trusted CA certificates.

- `addSeedStream`

  Adds an input stream that supplies seed data for the random number generator. You may call this method multiple times if necessary.

Obtaining Connection Information

You can obtain information about any SSL connection using the `Ice::Connection::getInfo` operation (see Section 36.5). It returns an `IceSSL.NativeConnectionInfo` class instance that derives from the Slice class `IceSSL::ConnectionInfo`. The Slice base class is defined as follows:

```
module Ice {
    local class ConnectionInfo {
        bool incoming;
        string adapterName;
    };

    local class IPConnectionInfo extends ConnectionInfo {
        string localAddress;
        int localPort;
        string remoteAddress;
        int remotePort;
    };
```

```
};

module IceSSL {
    local class ConnectionInfo extends Ice::IPConnectionInfo {
        string cipher;
        Ice::StringSeq certs;
    };
};
```

In turn, the Java class `NativeConnectionInfo` is defined as follows.

```
public class NativeConnectionInfo extends ConnectionInfo
{
    public java.security.cert.Certificate[] nativeCerts;
}
```

Installing a Certificate Verifier

A new connection undergoes a series of verification steps before an application is allowed to use it. The low-level SSL engine executes the validation procedures described in Section 41.2.2. Assuming the certificate chain is successfully validated, IceSSL performs additional verification as directed by its configuration properties (see Section 41.4.6). Finally, if a certificate verifier is installed, IceSSL invokes it to provide the application with an opportunity to decide whether to allow the connection to proceed.

The `CertificateVerifier` interface has only one method:

```
package IceSSL;

public interface CertificateVerifier
{
    boolean verify(NativeConnectionInfo info);
}
```

IceSSL rejects the connection if `verify` returns `false`, and allows it to proceed if the method returns `true`. The `verify` method receives a `NativeConnectionInfo` object that describes the connection's attributes.

The `nativeCerts` member of the `NativeConnectionInfo` is an array of certificates representing the peer's certificate chain. The array is structured so that the first element is the peer's certificate, followed by its signing certificates in the order they appear in the chain, with the root CA certificate as the last element. This member is null if the peer did not present a certificate chain.

The `cipher` member is a description of the ciphersuite that SSL negotiated for this connection. The local and remote address information is provided in `loca-`

`lAddress` and `remoteAddress`, respectively. The `incoming` member indicates whether the connection is inbound (a server connection) or outbound (a client connection). Finally, if `incoming` is `true`, the `adapterName` member supplies the name of the object adapter that hosts the endpoint.

The following class is a simple implementation of a certificate verifier:

```
import java.security.cert.X509Certificate;
import javax.security.auth.x500.X500Principal;

class Verifier implements IceSSL.CertificateVerifier
{
    public boolean
    verify(IceSSL.NativeConnectionInfo info)
    {
        if (info.nativeCerts != null)
        {
            X509Certificate cert =
                (X509Certificate)info.nativeCerts[0];
            X500Principal p = cert.getIssuerX500Principal();
            if (p.getName().toLowerCase().indexOf("zeroc") != -1)
            {
                return true;
            }
        }
        return false;
    }
}
```

In this example, the verifier rejects the connection unless the string `zeroc` is present in the issuer's distinguished name of the peer's certificate. In a more realistic implementation, the application is likely to perform detailed inspection of the certificate chain.

Installing the verifier is a simple matter of calling `setCertificateVerifier` on the plug-in interface:

```
IceSSL.Plugin sslPlugin = // ...
sslPlugin.setCertificateVerifier(new Verifier());
```

You should install the verifier before any SSL connections are established. An alternate way of installing the verifier is to define the `IceSSL.CertVerifier` property with the class name of your verifier implementation. IceSSL instantiates the class using its default constructor.

See Section 41.6.2 for more information on installing a certificate verifier.

Converting Certificates

Java does not provide a simple way to create a certificate object from a PEM-encoded string, therefore IceSSL offers the following convenience method:

```
package IceSSL;

public final class Util
{
    // ...

    public static java.security.cert.X509Certificate
    createCertificate(String certPEM)
        throws java.security.cert.CertificateException;
}
```

Given a string in the PEM format, `createCertificate` returns the equivalent `X509Certificate` object.

41.5.3 .NET

This section describes the .NET API for the IceSSL plug-in.

The `Plugin` Interface

Applications can interact directly with the IceSSL plug-in using the native C# interface `IceSSL.Plugin`. A reference to a `Plugin` object must be obtained from the communicator in which the plug-in is installed:

```
Ice.Communicator comm = // ...
Ice.PluginManager pluginMgr = comm.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
```

The `Plugin` interface supports the following methods:

```
namespace IceSSL
{
    using System.Security.Cryptography.X509Certificates;

    abstract public class Plugin : Ice.Plugin
    {
        abstract public void
        setCertificates(X509Certificate2Collection certs);

        abstract public void
        setCertificateVerifier(CertificateVerifier verifier);
```

```
        abstract public CertificateVerifier
        getCertificateVerifier();

        abstract public void
        setPasswordCallback(PasswordCallback callback);

        abstract public PasswordCallback
        getPasswordCallback();
    }
}
```

The methods are summarized below:

- `setCertificates`

  This method is rarely used in practice; see Section 41.6 for more information.

- `setCertificateVerifier`
  `getCertificateVerifier`

  These methods install and retrieve a custom certificate verifier object that the plug-in invokes for each new connection. `getCertificateVerifier` returns null if a verifier has not been set.

- `setPasswordCallback`
  `getPasswordCallback`

  These methods install and retrieve a password callback object that supplies IceSSL with passwords, as discussed in Section 41.6. `getPasswordCallback` returns null if a callback has not been set.

Obtaining Connection Information

You can obtain information about any SSL connection using the `Ice::Connection::getInfo` operation (see Section 36.5). It returns an `IceSSL.NativeConnectionInfo` class instance that derives from the Slice class `IceSSL::ConnectionInfo`. The Slice base class is defined as follows:

```
module Ice {
    local class ConnectionInfo {
        bool incoming;
        string adapterName;
    };

    local class IPConnectionInfo extends ConnectionInfo {
        string localAddress;
        int localPort;
```

```
        string remoteAddress;
        int remotePort;
    };
};

module IceSSL {
    local class ConnectionInfo extends Ice::IPConnectionInfo {
        string cipher;
        Ice::StringSeq certs;
    };
};
```

In turn, the C# class `NativeConnectionInfo` is defined as follows.

```
public sealed class NativeConnectionInfo : ConnectionInfo
{
    public System.Security.Cryptography.
                X509Certificates.X509Certificate2[] nativeCerts;
}
```

Installing a Certificate Verifier

A new connection undergoes a series of verification steps before an application is allowed to use it. The low-level SSL engine executes the validation procedures described in Section 41.2.2. Assuming the certificate chain is successfully validated, IceSSL performs additional verification as directed by its configuration properties (see Section 41.4.6). Finally, if a certificate verifier is installed, IceSSL invokes it to provide the application with an opportunity to decide whether to allow the connection to proceed.

The `CertificateVerifier` interface has only one method:

```
namespace IceSSL
{
    public interface CertificateVerifier
    {
        bool verify(NativeConnectionInfo info);
    }
}
```

IceSSL rejects the connection if `verify` returns `false`, and allows it to proceed if the method returns `true`. The `verify` method receives a `NativeConnectionInfo` object that describes the connection's attributes.

The `nativeCerts` member of the `c` is an array of certificates representing the peer's certificate chain. The array is structured so that the first element is the peer's certificate, followed by its signing certificates in the order they appear in

the chain, with the root CA certificate as the last element. This member is null if the peer did not present a certificate chain.

The `cipher` member is a description of the ciphersuite that SSL negotiated for this connection. The local and remote address information is provided in `localAddress` and `remoteAddr`, respectively. The `incoming` member indicates whether the connection is inbound (a server connection) or outbound (a client connection). Finally, if `incoming` is true, the `adapterName` member supplies the name of the object adapter that hosts the endpoint.

The following class is a simple implementation of a certificate verifier:

```
using System.Security.Cryptography.X509Certificates;

class Verifier : IceSSL.CertificateVerifier
{
    public boolean
    verify(IceSSL.NativeConnectionInfo info)
    {
        if (info.nativeCerts != null)
        {
            X500DistinguishedName dn =
                    info.nativeCerts[0].IssuerName;
            if (dn.Name.ToLower().Contains("zeroc"))
            {
                return true;
            }
        }
        return false;
    }
}
```

In this example, the verifier rejects the connection unless the string `zeroc` is present in the issuer's distinguished name of the peer's certificate. In a more realistic implementation, the application is likely to perform detailed inspection of the certificate chain.

Installing the verifier is a simple matter of calling `setCertificateVerifier` on the plug-in interface:

```
IceSSL.Plugin sslPlugin = // ...
sslPlugin.setCertificateVerifier(new Verifier());
```

You should install the verifier before any SSL connections are established. An alternate way of installing the verifier is to define the `IceSSL.CertVerifier` property with the class name of your verifier implementation. IceSSL instantiates the class using its default constructor.

See Section 41.6.2 for more information on installing a certificate verifier.

Converting Certificates

IceSSL offers the following convenience method to create a certificate object from a PEM-encoded string:

```
namespace IceSSL
{
    using System.Security.Cryptography.X509Certificates;

    public sealed class Util
    {
        // ...

        public static X509Certificate2
        createCertificate(string certPEM);
    }
}
```

Given a string in the PEM format, `createCertificate` returns the equivalent `X509Certificate2` object.

41.6 Advanced Topics

This section discusses some additional capabilities of the IceSSL plug-in.

41.6.1 Passwords

IceSSL may need to obtain a password if it loads a file that contains secure data, such as an encrypted private key. Section 41.4 showed how an application can supply a plain-text password in a configuration property and mentioned that doing so is a potential security risk. For example, if you define the property on the application's command-line, it may be possible for other users on the same host to see the password simply by obtaining a list of active processes. If you define the property in a configuration file, the password is only as secure as the file in which it is defined.

In highly secure environments where access to a host is tightly restricted, a password can safely be supplied as a plain-text configuration property, or the need for the password can be eliminated altogether by using unsecured key files.

In situations where password security is a concern, the application generally needs to take additional action.

Dynamic Properties

A common technique is to prompt the user for a password and transfer the user's input to a configuration property that the application defines dynamically, as shown below:

```
// C++
string password = // ...
Ice::InitializationData initData;
initData.properties = Ice::createProperties(argc, argv);
initData.properties->setProperty("IceSSL.Password", password);
Ice::CommunicatorPtr comm = Ice::initialize(initData);
```

The password must be present in the property set before the communicator is initialized, since IceSSL needs the password during its initialization, and the communicator initializes plug-ins automatically by default.

Password Callbacks in C++

If a password is required but the application has not configured one, IceSSL prompts the user at the terminal during the plug-in's initialization. This behavior is not suitable for some types of applications, such as a program that runs automatically at system startup as a Unix daemon or Windows service (see Section 8.3.2).

A terminal prompt is equally undesirable for graphical applications, which would generally prefer to prompt the user in an application window. The dynamic property technique described in the previous section is usually appropriate in this situation.

If your application must supply a password, and you do not want to use a configuration property or a terminal prompt, your remaining option is to install a `PasswordPrompt` object in the plug-in using the `setPasswordPrompt` method shown in Section 41.5.1. The `PasswordPrompt` class has the following definition:

```
namespace IceSSL
{
class PasswordPrompt : public IceUtil::Shared
{
public:
```

```
    virtual std::string getPassword() = 0;
};
typedef IceUtil::Handle<PasswordPrompt> PasswordPromptPtr;
}
```

IceSSL invokes `getPassword` on the object when a password is required. If the object returns an incorrect password, IceSSL tries again, up to the limit defined by the `IceSSL.PasswordRetryMax` property (see Appendix D).

Note that you must delay the initialization of the IceSSL plug-in until after the `PasswordPrompt` object is installed. To illustrate this point, consider the following example:

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr =
    communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin =
    IceSSL::PluginPtr::dynamicCast(plugin);
sslPlugin->setPasswordPrompt(new Prompt); // OOPS!
```

This code is incorrect because the `PasswordPrompt` object is installed too late: the communicator is already initialized, which means IceSSL has already attempted to load the file that required a password.

The correct approach is to define the following configuration property:

```
Ice.InitPlugins=0
```

This setting causes the communicator to install, but not initialize, its configured plug-ins. The application becomes responsible for initializing the plug-ins, as shown below:

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr =
    communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin =
    IceSSL::PluginPtr::dynamicCast(plugin);
sslPlugin->setPasswordPrompt(new Prompt);
pluginMgr->initializePlugins();
```

We assume the communicator was initialized with `Ice.InitPlugins=0`. After installing the `PasswordPrompt` object, the application invokes `initializePlugins` on the plug-in manager to complete the plug-in initialization process.

Password Callbacks in Java

If you do not want to use configuration properties to define passwords, you can install a `PasswordCallback` object in the plug-in using a configuration property, or using the `setPasswordCallback` method shown in Section 41.5.2. The `PasswordCallback` interface has the following definition:

```
public interface PasswordCallback
{
    char[] getPassword(String alias);
    char[] getTruststorePassword();
    char[] getKeystorePassword();
}
```

The methods are described below:

- `getPassword`

  Supplies the password for the key with the given alias. The return value must not be null.

- `getTruststorePassword`

  Supplies the password for a truststore. The method may return null, in which case the integrity of the truststore is not verified.

- `getKeystorePassword` to obtain the password for a keystore.

  Supplies the password for a keystore. The method may return null, in which case the integrity of the keystore is not verified.

For each of these methods, IceSSL clears the contents of the returned array as soon as possible.

The simplest way to install the callback is by defining the configuration property `IceSSL.PasswordCallback`. The property's value is the name of your callback implementation class (see Appendix D). IceSSL instantiates the class using its default constructor.

To install the callback manually, you must delay the initialization of the IceSSL plug-in until after the `PasswordCallback` object is installed. To illustrate this point, consider the following example:

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI()); // OOPS!
```

This code is incorrect because the `PasswordCallback` object is installed too late: the communicator is already initialized, which means IceSSL has already attempted to retrieve the certificate that required a password.

The correct approach is to define the following configuration property:

```
Ice.InitPlugins=0
```

This setting causes the communicator to install, but not initialize, its configured plug-ins. The application becomes responsible for initializing the plug-ins, as shown below:

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI());
pluginMgr.initializePlugins();
```

We assume the communicator was initialized with `Ice.InitPlugins=0`. After installing the `PasswordCallback` object, the application invokes `initializePlugins` on the plug-in manager to complete the plug-in initialization process.

Password Callbacks in .NET

If you do not want to use configuration properties to define passwords, you can install a `PasswordCallback` object in the plug-in using a configuration property, or using the `setPasswordCallback` method shown in Section 41.5.2. The `PasswordCallback` interface has the following definition:

```
using System.Security;

public interface PasswordCallback
{
    SecureString getPassword(string file);
    SecureString getImportPassword(string file);
}
```

The methods are described below:

- `getPassword`

  Supplies the password for the given file. The method may return null if no password is required.

- `getImportPassword`

  Supplies the password for a file from which certificates are imported into the certificate store. The method may return null if no password is required.

The simplest way to install the callback is by defining the configuration property `IceSSL.PasswordCallback`. The property's value is the name of your callback implementation class (see Appendix D). IceSSL instantiates the class using its default constructor.

To install the callback manually, you must delay the initialization of the IceSSL plug-in until after the `PasswordCallback` object is installed. To illustrate this point, consider the following example:

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI()); // OOPS!
```

This code is incorrect because the `PasswordCallback` object is installed too late: the communicator is already initialized, which means IceSSL has already attempted to retrieve the certificate that required a password.

The correct approach is to define the following configuration property:

```
Ice.InitPlugins=0
```

This setting causes the communicator to install, but not initialize, its configured plug-ins. The application becomes responsible for initializing the plug-ins, as shown below:

```
Ice.Communicator communicator = // ...
Ice.PluginManager pluginMgr = communicator.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
sslPlugin.setPasswordCallback(new CallbackI());
pluginMgr.initializePlugins();
```

We assume the communicator was initialized with `Ice.InitPlugins=0`. After installing the `PasswordCallback` object, the application invokes `initializePlugins` on the plug-in manager to complete the plug-in initialization process.

Manual Configuration

The `Plugin` interface described in Section 41.5 supports a method in each of the supported language mappings that provides an application with more control over the plug-in's configuration.

In C++ and Java, an application can call the `setContext` method to supply a pre-configured "context" object used by the underlying SSL engines. In .NET, the `setCertificates` method accepts a collection of certificates that the plug-in should use. In all cases, using one of these methods causes IceSSL to ignore (at a minimum) the configuration properties related to certificates and keys. The application is responsible for accumulating its certificates and keys, and must also deal with any password requirements.

Describing the use of these plug-in methods in detail is outside the scope of this book, however it is important to understand their prerequisites. In particular, the application needs to have the communicator load the plug-in but not actually initialize it until after the application has had a chance to interact directly with it. (The previous section showed one example of this technique.) The application must define the following configuration property:

```
Ice.InitPlugins=0
```

With this setting, the application becomes responsible for completing the plug-in initialization process by invoking `initializePlugins` on the `PluginManager`. The C# example below demonstrates the proper steps:

```
// C#
Ice.Communicator comm = // ...
Ice.PluginManager pluginMgr = comm.getPluginManager();
Ice.Plugin plugin = pluginMgr.getPlugin("IceSSL");
IceSSL.Plugin sslPlugin = (IceSSL.Plugin)plugin;
X509Certificate2Collection certs = // ...
sslPlugin.setCertificates(certs);
pluginMgr.initializePlugins();
```

41.6.2 Custom Plug-Ins

The Ice plug-in facility is not restricted to protocol implementations. Ice only requires that a plug-in implement the `Ice::Plugin` interface and support the language-specific mechanism for dynamic loading.

The customization options of the IceSSL plug-in make it possible for you to install an application-specific implementation of a certificate verifier in an existing program. For example, you could install a custom certificate verifier in a

Glacier2 router without the need to modify Glacier2's source code or rebuild the executable. You would have to write a C++ plug-in to accomplish this, since Glacier2 is written in C++. In short, your plug-in must interact with the IceSSL plug-in and install a certificate verifier.

For this technique to work, it is important that the plug-ins be loaded in a particular order. Specifically, the IceSSL plug-in must be loaded first, followed by the certificate verifier plug-in. By default, Ice loads plug-ins in an undefined order, but you can use the property `Ice.PluginLoadOrder` to specify a particular order.

As an example, let's write a plug-in that installs the simple certificate verifier from Section 41.5.1. Here is the definition of our plug-in class:

```
class VerifierPlugin : public Ice::Plugin
{
public:
    VerifierPlugin(const Ice::CommunicatorPtr & comm) :
        _comm(comm)
    {
    }

    virtual void initialize()
    {
        Ice::PluginManagerPtr pluginMgr =
            _comm->getPluginManager();
        Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
        IceSSL::PluginPtr sslPlugin =
            IceSSL::PluginPtr::dynamicCast(plugin);
        sslPlugin->setCertificateVerifier(new Verifier);
    }

    virtual void destroy()
    {
    }

private:
    Ice::CommunicatorPtr _comm;
};
```

The class implements the two operations in the `Plugin` interface, `initialize` and `destroy`. The code in `initialize` installs the certificate verifier object, while nothing needs to be done in `destroy`.

The next step is to write the plug-in's factory function, which the communicator invokes to obtain an instance of the plug-in:

```
extern "C"
{

Ice::Plugin*
createVerifierPlugin(
    const Ice::CommunicatorPtr & communicator,
    const string & name,
    const Ice::StringSeq & args)
{
    return new VerifierPlugin(communicator);
}

}
```

We can give the function any name; in this example, we chose `createVerifierPlugin`.

Finally, to install the plug-in we need to define the following properties:

```
Ice.PluginLoadOrder=IceSSL,Verifier
Ice.Plugin.IceSSL=IceSSL:createIceSSL
Ice.Plugin.Verifier=Verifier:createVerifierPlugin
```

The value of `Ice.PluginLoadOrder` guarantees that IceSSL is loaded first. The plug-in specification `Verifier:createVerifierPlugin` identifies the name of the shared library or DLL and the name of the registration function; see the description of the `Ice.Plugin` property in Appendix D for more information.

There are a few more details you must attend to, such as ensuring that the factory function is exported properly and building the shared library or DLL that contains the new plug-in. Refer to Section 32.25 for more information on developing a plug-in.

41.7 Setting up a Certificate Authority

During development, it is convenient to have a simple way of creating new certificates. OpenSSL includes all of the necessary infrastructure for setting up your own certificate authority (CA), but it requires getting more familiar with OpenSSL than is really necessary. To simplify the process, Ice includes the Python script **`iceca`**, located in the `bin` subdirectory of your Ice installation, that hides the complexity of OpenSSL and allows you to quickly perform the essential tasks:

- initializing a new root CA

- generating new certificate requests
- signing certificate requests to create a valid certificate chain
- converting certificates to match platform-specific requirements.

You are not obligated to use this script; IceSSL accepts certificates from any source as long as they are provided in the appropriate formats. However, you may find this tool sufficient for your development needs, and possibly even for your deployed application as well.

Some of the script's activities use a directory that contains configuration files and a database of issued certificates. The script selects a default location for this directory that depends on your platform, or you can specify the parent directory explicitly by defining the `ICE_CA_HOME` environment variable and the script will use `$ICE_CA_HOME/ca` for its files.

41.7.1 Initializing a Certificate Authority

The script command **`iceca init`** initializes a new CA by preparing a database directory and generating the root CA certificate and private key. It accepts the following command-line arguments:

```
$ python iceca init [--no-password] [--overwrite]
```

Upon execution, the script first checks the database directory to determine whether it has already been initialized. If so, the script terminates immediately with a warning unless you specify the **`--overwrite`** option, in which case the script overwrites the previous contents of the directory.

Next, the script displays the database directory it is using and begins to prompt you for the information it needs to generate the root CA certificate and private key. It offers a default choice for the CA's distinguished name and allows you to change it:

```
The subject name for your CA will be
CN=Grid-CA ,  O=GridCA-server
Do you want to keep this as the CA subject name? (y/n) [y]
```

To specify an alternate value for the distinguished name, enter **`n`** and type the new information, otherwise hit Enter to proceed.

```
Enter the email address of the CA: ca-admin@company.com
```

The address you provide in response to this prompt is shown to users that create certificate requests. Enter the address to which such requests should be sent.

The script shows its progress as it generates the certificate and private key, then prompts you for a pass phrase. If you prefer not to secure your CA's private key with a pass phrase, use the **`--no-password`** option.

Upon completion, the script emits the following instructions:

```
The CA is initialized.

You need to distribute the following files to all machines that
can request certificates:

C:\iceca\req.cnf
C:\iceca\ca_cert.pem

These files should be placed in the user's home directory in
~/.iceca. On Windows, place these files in <ice-install>/config.
```

In this example, the `ICE_CA_HOME` environment variable was set to `C:\iceca`. As the script states, the files `req.cnf` and `ca_cert.pem` must be present on each host that can generate a certificate request. The script suggests a location for these files, which is the default directory used by the scripts if `ICE_CA_HOME` is not defined.

The `ca_cert.pem` file contains the root CA's certificate. Your IceSSL configurations must identify this certificate (in its proper form for each platform) as a trusted certificate. For example, you can use this file directly in the configuration of the C++ plug-in:

```
IceSSL.CertAuthFile=C:\iceca\ca_cert.pem
```

For .NET applications, you should import the certificate file into the proper store, as described in Section 41.4.3.

In Java, you need to add the certificate to your truststore:

```
$ keytool -import -trustcacerts -file ca_cert.pem -keystore ca.jks
Enter keystore password:
```

The keytool program requires you to enter a password, which you could use as the value of the property `IceSSL.TruststorePassword` (see Appendix D).

Now that your certificate authority is initialized, you can begin generating certificate requests.

41.7.2 Generating Certificate Requests

The script command **`iceca request`** uses the files you created in Section 41.7.1 to generate a request for a new certificate. It accepts the following command-line arguments:

```
$ python iceca request [--overwrite] [--no-password]
file common-name [email]
```

The script looks for the files `req.cnf` and `ca_cert.pem` in the directory defined by the `ICE_CA_HOME` environment variable. If that variable is not defined, the script uses a default directory that depends on your platform.

The purpose of the script is to generate two files: a private key and a file containing the certificate request. The request file must be transmitted to the certificate authority for signing, which produces a valid certificate chain.

The argument **`file`** is used as a prefix for the names of the two output files created by the script:

- ***`file`*`_key.pem`** contains the private key
- ***`file`*`_req.pem`** contains the certificate request

If the output files already exist, you must specify **`--overwrite`** to force the script to overwrite them.

The **`common-name`** argument defines the common name component of the certificate's distinguished name. If the optional **`email`** argument is provided, it is also included in the certificate request.

During execution, the script displays its progress as it generates the necessary files. It will prompt you for a pass phrase unless you used the **`--no-password`** option, and finish by showing the names of the files it created as well as instructions on how to proceed. The example below shows the output from generating a request for an IceGrid node using a filename prefix of **`node`**:

```
$ iceca request node "IceGrid Node"

Created key: node_key.pem
Created certificate request: node_req.pem

The certificate request must be signed by the CA. Send the
certificate request file to the CA at the following email
address:
ca-admin@company.com
```

```
The file node_key.pem is the new private key for the node; this file
must be kept secure. The file node_req.pem must be given to the
certificate authority. As a convenience, the script displays the CA's
email address that you entered in Section 41.7.1.
```

41.7.3 Signing Certificate Requests

As a certificate authority, you are responsible for certifying the validity of certificate requests by signing them with your private key. The product of signing a request is a valid certificate chain that a person or application can use as an identity. The **iceca sign** command performs this task for you and accepts the following command-line arguments:

```
$ python iceca sign --in <req> --out <cert> [--ip <ip> --dns <dns>]
```

The input file **req** is the certificate request, and the output file **cert** is the certificate chain. The script does not overwrite the file **cert** unless you also specify **--overwrite**. The **--ip** and **--dns** options allow you to add subject alternative names to the certificate for IP and DNS addresses, respectively.

Continuing the example from Section 41.7.2, we can sign the node's request with the following command:

```
$ python iceca sign --in node_req.pem --out node_cert.pem
```

If the CA's private key is protected by a pass phrase, we must enter that first. Next, the script displays the relevant information from the certificate request and asks you to confirm that you wish to sign the certificate:

```
The Subject's Distinguished Name is as follows
organizationName      :PRINTABLE:'Company.com'
commonName            :PRINTABLE:'IceGrid Node TestNode'
Certificate is to be certified until Jun 15 18:32:36 2011 GMT
Sign the certificate? [y/n]:
```

After reviewing the request, enter **y** to sign the certificate, and **y** again to finish the process. Upon completion, the script stores the certificate chain in the file `node_cert.pem` in your current working directory. This file, together with the node's private key we created in Section 41.7.2, establishes a secure identity for the node.

41.7.4 Converting Certificates

For Java and .NET users, the private key and certificate chain must be converted into a suitable format for your platform. The script command **iceca import** simplifies this process and accepts the following command-line arguments:

```
$ python iceca import [--overwrite] [--key-pass password]
[--store-pass password] [--java alias cert key keystore]
[--cs cert key out-file]
```

The script does not overwrite an existing file unless you specify **--overwrite**. To avoid interactive prompts for passwords, you can use the **--key-pass** option to specify the password for the private key, and the **--store-pass** option to define the password for the Java keystore. Completing our node example from prior sections, the command below imports the private key and certificate chain into a Java keystore:

```
$ python iceca import --java mycert node_cert.pem
node_key.pem cert.jks
```

The value **mycert** represents the alias associated with this entry in the keystore, and **cert.jks** is the name of the new keystore file. In an IceSSL configuration, the property `IceSSL.Keystore` refers to this file.

The equivalent command for .NET is shown below:

```
$ python iceca import --cs node_cert.pem node_key.pem cert.pfx
```

The file **cert.pfx** uses the PKCS#12 encoding and contains the certificate chain and private key. You can import this certificate into a store, or refer directly to the file using the configuration property `IceSSL.CertFile`.

41.7.5 Diagnostics

If you encounter a problem while using the **iceca** script, or simply want to learn more about the underlying OpenSSL commands used by the script, you can run the script with the **--verbose** option as shown below:

```
$ python iceca --verbose command ...
```

This option causes the script to display the commands as it executes them.

The script creates temporary files and directories that are normally deleted prior to the script's completion. If you would like to examine the contents of these files and directories, use the **--keep** option:

```
$ python iceca --keep command ...
```

41.8 Summary

The Secure Socket Layer (SSL) protocol is the de facto standard for secure network communication. Its support for authentication, non-repudiation, data integrity, and strong encryption makes it the logical choice for securing Ice applications.

Although security is an optional component of Ice, it is not an afterthought. The IceSSL plug-in integrates easily into existing Ice applications, in most cases requiring nothing more than configuration changes. Naturally, some additional effort is required to create the necessary security infrastructure for an application, but in many enterprises this work will have already been done.

Chapter 42
Glacier2

42.1 Chapter Overview

In this chapter we present the Glacier2 service, a lightweight firewall solution for Ice applications. The basic requirements for using Glacier2 are discussed in Section 42.3. Glacier2 supports callbacks from servers to clients and Section 42.4 provides details about the configuration and application requirements necessary to use callbacks. Section 42.6 covers security issues, while Section 42.7 presents an overview of Glacier2 session management. Dynamic filtering is covered in Section 42.8. Section 42.9 describes Glacier2's buffering semantics, and Section 42.10 describes the handling of request contexts. The use of a network firewall in conjunction with Glacier2 is the topic of Section 42.11. Finally, clients with special requirements are addressed by Section 42.12, and IceGrid integration is the subject of Section 42.13.

42.2 Introduction

We have presented many examples of client/server applications in this book, all of which assume that the client and server programs are running either on the same host, or on multiple hosts with no network restrictions. We can justify this assumption because this is an instructional text, but a real-world network environ-

ment is usually much more complicated: client and server hosts with access to public networks often reside behind protective router-firewalls that not only restrict incoming connections, but also allow the protected networks to run in a private address space using Network Address Translation (NAT). These features, which are practically mandatory in today's hostile network environments, also disrupt the ideal world in which our examples are running.

42.2.1 Common Scenarios

Let us assume that a client and server need to communicate over an untrusted network, and that the client and server hosts reside in private networks behind firewalls, as shown in Figure 42.1.

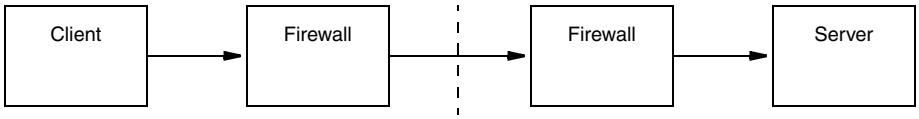


Figure 42.1. Client request in a typical network scenario.

Although the diagram looks fairly straightforward, there are several troublesome issues:

- A dedicated port on the server's firewall must be opened and configured to forward messages to the server.
- If the server uses multiple endpoints (e.g., to support both TCP and SSL), then a firewall port must be dedicated to each endpoint.
- The client's proxy must be configured to use the server's "public" endpoint, which is the host name and dedicated port of the firewall.
- If the server returns a proxy as the result of a request, the proxy must not contain the server's private endpoint because that endpoint is inaccessible to the client.

To complicate the scenario even further, Figure 42.2 adds a callback from the server to the client. Callbacks imply that the client is also a server, therefore all of the issues associated with Figure 42.1 now apply to the client as well.

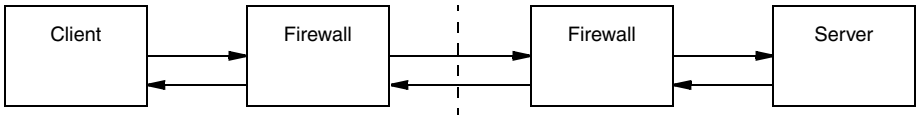


Figure 42.2. Callbacks in a typical network scenario.

As if this was not complicated enough already, Figure 42.3 adds multiple clients and servers. Each additional server (including clients requiring callbacks) adds more work for the firewall administrator as more ports are dedicated to forwarding requests.

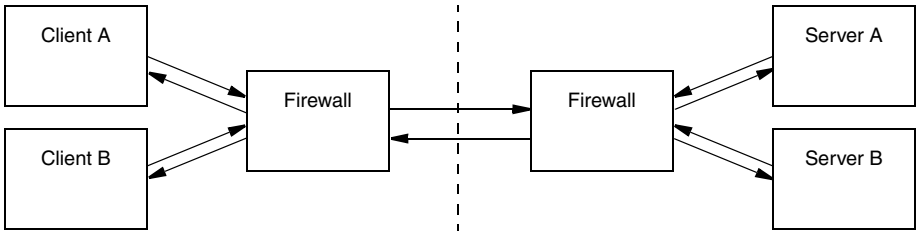


Figure 42.3. Multiple clients and servers with callbacks in a typical network scenario.

Clearly, these scenarios do not scale well, and are unnecessarily complex. Fortunately, Ice provides a solution.

42.2.2 What is Glacier2?

Glacier2, the router-firewall for Ice applications, addresses the issues raised in Section 42.2.1 with minimal impact on clients or servers (or firewall administrators). In Figure 42.4, Glacier2 becomes the server firewall for Ice applications.

What is not obvious in the diagram, however, is how Glacier2 eliminates much of the complexity of the previous scenarios.

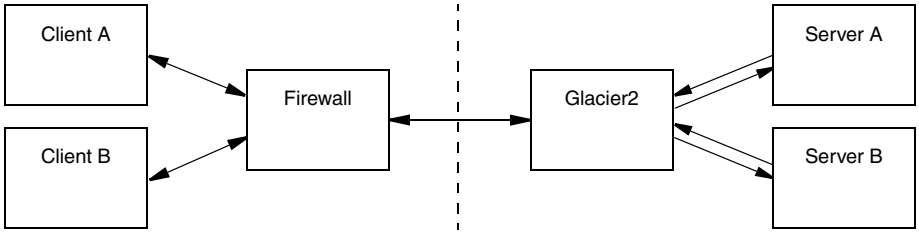


Figure 42.4. Multiple clients and servers with callbacks using Glacier.

In particular, Glacier2 provides the following advantages:

- Clients often require only minimal changes to use Glacier2.
- Only one front end port is necessary to support any number of servers, allowing a Glacier2 router to easily receive connections from a port-forwarding firewall.
- The number of connections to back end servers is reduced. Glacier2 effectively acts as a connection concentrator, establishing a single connection to each back end server to forward requests from any number of clients. Similarly, connections from back end servers to Glacier2 for the purposes of sending callbacks are also concentrated.
- Servers are unaware of Glacier2's presence, and require no modifications whatsoever to use Glacier2. From a server's perspective, Glacier2 is just another local client, therefore servers are no longer required to advertise "public" endpoints in the proxies they create. Furthermore, back-end services such as IceGrid (see Chapter 38) can continue to be used transparently via a Glacier2 router.
- Callbacks are supported without requiring new connections from servers to clients (see Section 42.4). In other words, a callback from a server to a client is sent over an existing connection from the client to the server, thereby eliminating the administrative requirements associated with supporting callbacks in the client firewall.
- Glacier2 requires no knowledge of the application's Slice definitions and therefore is very efficient: it routes request and reply messages without unmarshalling the message contents.

- In addition to its primary responsibility of forwarding Ice requests, Glacier2 offers support for user-defined session management and authentication, inactivity timeouts, and request buffering and batching.

42.2.3 How it works

The Ice core supports a generic router facility, represented by the `Ice::Router` interface, that allows a third-party service to "intercept" requests on a properly-configured proxy and deliver them to the intended server. Glacier2 is an implementation of this service, although other implementations are certainly possible.

Glacier2 normally runs on a host in the private network behind a port-forwarding firewall (see Section 42.11), but it can also operate on a host with access to both public and private networks. In this configuration it follows that Glacier2 must have endpoints on each network, as shown in Figure 42.5.

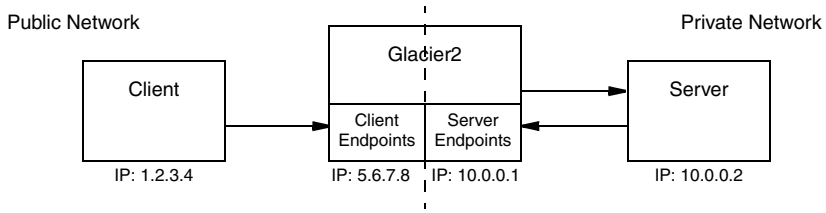


Figure 42.5. Glacier2's client and server endpoints.

In the client, proxies must be configured to use Glacier2 as a router. This configuration can be done statically for all proxies created by a communicator, or programmatically for a particular proxy. A proxy configured to use a router is called a *routed proxy.*

When a client invokes an operation on a routed proxy, the client connects to one of Glacier2's client endpoints and sends the request as if Glacier2 were the server. Glacier2 then establishes a client connection to the intended server, forwards the request, and returns the reply (if any). Glacier2 is essentially acting as a local client on behalf of the remote client.

If a server returns a proxy as the result of an operation, that proxy contains the server's endpoints in the private network, as usual. (Remember, the server is unaware of Glacier2's presence, and therefore assumes that the proxy is usable by the client that requested it.) Of course, those endpoints are not accessible to the client and, in the absence of a router, the client would receive an exception if it were to use the proxy. When that proxy is configured with a router, however, the

client ignores the server's endpoints and only sends requests to the router's client endpoints.

Glacier2's server endpoints, which reside in the private network, are only used when a server makes a callback to a client. See Section 42.4 for more information on callbacks.

42.2.4 Limitations

Glacier2 has the following limitations:

- Datagram protocols, such as UDP, are not supported.
- Callback objects in a client must use a Glacier2-supplied category in their identities (see Section 42.4).

42.3 Using Glacier2

Getting started with Glacier2 in a minimal configuration involves the following tasks:

1. Write a configuration file for the router.
2. Write a password file for the router. (Section 42.6 discusses alternative ways to authenticate users.)
3. Decide whether to use the router's internal session manager, or supply your own (see Section 42.7).
4. Start the router on a host with access to the public and private networks.
5. Modify the client configuration to use the router.
6. Modify the client to create a router session.
7. Ensure that the router session remains active for as long as the client requires it.

For the sake of example, let us assume that the router's public address is `5.6.7.8` and its private address is `10.0.0.1`.

42.3.1 Configuring the Router

The following router configuration properties establish the necessary endpoint and define when a session expires due to inactivity:

```
Glacier2.Client.Endpoints=tcp -h 5.6.7.8 -p 4063
Glacier2.SessionTimeout=60
```

The endpoint defined by `Glacier2.Client.Endpoints` is used by the Ice run time in a client to interact directly with the router. It is also the endpoint where requests from routed proxies are sent. This endpoint is defined on the public network interface because it must be accessible to clients.[1] Furthermore, the endpoint uses a fixed port because clients may be statically configured with a proxy for this endpoint.

The port numbers 4063 (for TCP) and 4064 (for SSL) are reserved for Glacier2 by the Internet Assigned Numbers Authority (IANA).

A client's session is destroyed when explicitly requested, or when the session is inactive for a configurable number of seconds. For this example, we have specified a timeout of 60 seconds. It is not mandatory to define a timeout, but it is recommended, otherwise session state might accumulate in the router. See Section 42.7 for more information on sessions.

Note that this configuration enables the router to forward requests from clients to servers, but not from servers to clients (that is, it cannot forward callbacks). We discuss callbacks in Section 42.4.

You must also decide which authentication scheme (or schemes) to use. Section 42.3.2 describes a file-based mechanism, and Section 42.6 covers the router's more sophisticated facilities.

If clients access a location service via the router, additional router configuration is typically necessary (see Section 38.15).

42.3.2 Writing a Password File

The router's simplest authentication mechanism uses an access control list in a file consisting of user name–password pairs. The password is a 13-character string encoded using the `crypt` algorithm, similar to a `passwd` file on a typical Unix system. The property `Glacier2.CryptPasswords` specifies the name of the password file:

```
Glacier2.CryptPasswords=passwords
```

1. This sample configuration uses TCP as the endpoint protocol, although in most cases SSL is preferable (see Section 42.6).

The format of the password file is very simple. Each user name–password pair must reside on a separate line, with whitespace separating the user name from the password. For example, the following password file contains an entry for the user name `test`:

```
test xxMqsnnDcK8tw
```

You can use the **`openssl`** utility (included in the OpenSSL toolkit) to generate `crypt` passwords:

```
$ openssl
OpenSSL> passwd
Password:
Verifying - Password:
xxMqsnnDcK8tw
```

At the prompt, issue the **`passwd`** command. You are asked for a password, and then asked to confirm the password, at which point the utility displays the `crypt`-encoded version of your password that you can paste into the router's password file.

42.3.3 Starting the Router

The router supports the following command-line options:

```
$ glacier2router -h
Usage: glacier2router [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
--nowarn             Suppress warnings.
```

The `--nowarn` option prevents the router from displaying warning messages at startup when it is unable to contact a permissions verifier object or a session manager object specified by its configuration.

Additional command line options are supported, including those that allow the router to run as a Windows service or Unix daemon. See Appendix H for more information.

Assuming the configuration properties shown in Section 42.3.1 and Section 42.3.2 are stored in a file named `config`, you can start the router with the following command:

```
$ glacier2router --Ice.Config=config
```

42.3.4 Configuring the Client

The following properties configure a client to use a Glacier2 router:

```
Ice.Default.Router=Glacier2/router:tcp -h 5.6.7.8 -p 4063
Ice.ACM.Client=0
Ice.RetryIntervals=-1
```

The value of the `Ice.Default.Router` property is a proxy whose endpoints must match those in `Glacier2.Client.Endpoints`.

The property `Ice.ACM.Client` governs the behavior of *active connection management* (ACM, see Section 36.4), which conserves resources by periodically closing idle outgoing connections. This feature must be disabled in a client that uses a Glacier2 router, otherwise ACM might transparently close a client's connection to a router and thereby terminate the router session prematurely. ACM is enabled by default, and therefore must be disabled by setting this property to zero.

Finally, setting `Ice.RetryIntervals` to `-1` disables automatic retries, which are not useful for proxies configured to use a Glacier2 router. For more information about automatic retries, see Section 32.22.

42.3.5 Object Identities

A Glacier2 router hosts two well-known objects. The default identities of these objects are `Glacier2/router` and `Glacier2/admin`, corresponding to the `Glacier2::Router` and `Glacier2::Admin` interfaces, respectively. If an application requires the use of multiple different (that is, not replicated) routers, it is a good idea to assign unique identities to these objects by configuring the routers with different values of the `Glacier2.InstanceName` property, as shown in the following example:

```
Glacier2.InstanceName=PublicRouter
```

This property changes the category of the object identities, which become `PublicRouter/router` and `PublicRouter/admin`. The client's configuration must also be changed to reflect the new identity:

```
Ice.Default.Router=PublicRouter/router:tcp -h 5.6.7.8 -p 4063
```

One exception to this rule is if you deploy multiple Glacier2 routers as replicas, for example, to gain redundancy or to distribute the message forwarding load over a number of machines. In that case, all the routers must use the same instance name; the clients in that case use proxies with multiple endpoints, such as:

```
Ice.Default.Router=PublicRouter/router:tcp -h 5.6.7.8 -p 4063:
tcp -h 6.10.7.8 -p 4063
```

42.3.6 Creating a Session

Session management is provided by the `Glacier2::Router` interface:[2]

```
module Glacier2 {
    exception PermissionDeniedException {
        string reason;
    };

    interface Router extends Ice::Router {
        Session* createSession(string userId, string password)
            throws PermissionDeniedException,
                   CannotCreateSessionException;

        Session* createSessionFromSecureConnection()
            throws PermissionDeniedException,
                   CannotCreateSessionException;

        idempotent string getCategoryForClient();

        void refreshSession()
            throws SessionNotExistException;

        void destroySession()
            throws SessionNotExistException;

        idempotent long getSessionTimeout();
    };
};
```

The interface defines two operations for creating sessions: `createSession` and `createSessionFromSecureConnection`. The router requires each client to create a session using one of these operations; only after the session is created will the router forward requests on behalf of the client.

The `createSession` operation expects a user name and password, and returns a `Session` proxy or nil, depending on the router's configuration (see Section 42.7). When using the default authentication scheme, the given user name

2. The `getCategoryForClient` operation is used for bidirectional connections (see page 1719).

and password must match an entry in the router's password file in order to successfully create a session.

The `createSessionFromSecureConnection` operation does not require a user name and password because it uses the credentials supplied by an SSL connection to authenticate the client (see Section 42.6).

To create a session, the client typically obtains the router proxy from the communicator, downcasts the proxy to the `Glacier2::Router` interface, and invokes one of the operations. The sample code below demonstrates how to do it in C++; the code will look very similar in the other language mappings.

```
Ice::RouterPrx defaultRouter =
    communicator->getDefaultRouter();
Glacier2::RouterPrx router =
    Glacier2::RouterPrx::checkedCast(defaultRouter);
string username = ...;
string password = ...;
Glacier2::SessionPrx session;
try
{
    session = router->createSession(username, password);
}
catch(const Glacier2::PermissionDeniedException& ex)
{
    cout << "permission denied:\n" << ex.reason << endl;
}
catch(const Glacier2::CannotCreateSessionException& ex)
{
    cout << "cannot create session:\n" << ex.reason << endl;
}
```

If the router is configured with a session manager, the `createSession` and `createSessionFromSecureConnection` operations may return a proxy for an object implementing the `Glacier2::Session` interface (or an application-specific derived interface). The client receives a null proxy if no session manager is configured.

In order to successfully use a session proxy, it must be configured with the router that created it; that is, the session object is only accessible via the router. If the router is configured as the client's default router at the time `createSession` or `createSessionFromSecureConnection` is invoked, as is the case in the example above, then the session proxy is already properly configured and nothing else is required. Otherwise, the client must explicitly configure the session proxy with a router using the `ice_router` proxy method (see Section 32.11.2).

If the client wishes to destroy the session explicitly, it must invoke `destroy-Session` on the router proxy. If a client does not destroy its session, the router destroys it automatically when it expires due to inactivity. A client can obtain the inactivity timeout value by calling `getSessionTimeout` and keep the session alive by periodically calling `refreshSession` if necessary (see Section 42.3.7).

An example of a Glacier2 client is provided in the directory `demo/Glacier2/callback`.

42.3.7 Session Expiration

A Glacier2 router may be configured to destroy sessions after a period of inactivity. This feature allows the router, as well as a custom session manager, to reclaim resources acquired during the session (see Section 42.7), but it requires some coordination between the router and its clients.

Ideally you would select a session timeout that is long enough to accommodate the usage patterns of your clients. For example, a session timeout of thirty seconds is a reasonable choice for a client that invokes an operation on a back-end server once every five seconds. However, that timeout could disrupt a different client that has long periods of inactivity, such as when its invocations are prompted by human interaction.

If you cannot predict with certainty the usage patterns of your clients, we recommend modifying the clients so that they actively prevent their sessions from expiring. A client simply needs to make an invocation at regular intervals, where the period is less than the router's timeout by a comfortable margin. Typically a client creates a dedicated thread whose only purpose is keeping the session alive. It can accomplish this by invoking `refreshSession` on the router proxy, or by invoking the `ice_ping` operation on the proxy of any back-end object that is accessed via the router (including the `Session` proxy, if one was returned by `createSession`). Ice includes helper classes that you can use to simplify the task of create a session and keeping it alive (see XREF).

Note that if a session times out, the next client invocation raises `Connection-LostException`. To re-establish the session, the client must explicitly re-create it. If the client uses callbacks (see Section 42.4), it must re-create the callback adapter and re-add the callback servants to the ASM.

See Section 42.7.2 for more information on configuring session timeouts.

42.4 Callbacks

Callbacks from servers to clients are commonly used in distributed applications, often for notification purposes (such as the completion of a long-running calculation or a change to a database record). Unfortunately, supporting callbacks in a complicated network environment presents its own set of problems, as described in Section 42.2.1. Ice overcomes these obstacles using a Glacier2 router and bidirectional connections.

42.4.1 Bidirectional Connections

While a regular unrouted connection allows requests to flow in only one direction (from client to server), a bidirectional connection enables requests to flow in both directions. This capability is necessary to circumvent the network restrictions discussed in Section 42.2.1, namely, client-side firewalls that prevent a server from establishing an independent connection directly to the client. By sending callback requests over the existing connection from the client to the server (more accurately, from the client to the router), we have created a virtual connection back to the client. Figure 42.6 illustrates the steps involved in making a callback using Glacier2.

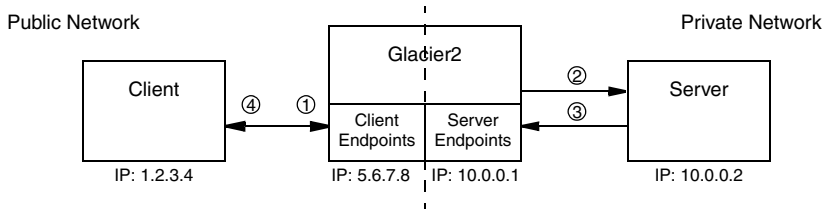


Figure 42.6. A callback via Glacier2.

1. The client has a routed proxy for the server and makes an invocation. A connection is established to the router's client endpoint and the request is sent to the router.
2. The router, using information from the client's proxy, establishes a connection to the server and forwards the request. In this example, one of the arguments in the request is a proxy for a callback object in the client.
3. The server makes a callback to the client. For this to succeed, the proxy for the callback object must contain endpoints that are accessible to the server. The

only path back to the client is through the router, therefore the proxy contains the router's server endpoints (see Section 42.4.3). The server connects to the router and sends the request.

4. The router forwards the callback request to the client using the bidirectional connection established in step 1.

The arrows in Figure 42.6 indicate the flow of requests; notice that two connections are used between the router and the server. Since the server is unaware of the router, it does not use routed proxies, and therefore does not use bidirectional connections.

It is also possible for applications to manually configure bidirectional connections without the use of a router. See Section 36.7 for more information on bidirectional connections.

42.4.2 Lifetime of a Bidirectional Connection

When a client terminates, it closes its connection to the router. If a server later attempts to make a callback to the client, the attempt fails because the router has no connection to the client over which to forward the request. This situation is no worse than if the server attempted to contact the client directly, which would be prevented by the client firewall. However, this illustrates the inherent limitation of bidirectional connections: the lifetime of a client's callback proxy is bounded by the lifetime of the client's router session.

42.4.3 Configuring the Router

In order for the router to support callbacks from servers, it needs to have endpoints in the private network. The configuration file shown below adds the property `Glacier2.Server.Endpoints`:

```
Glacier2.Client.Endpoints=tcp -h 5.6.7.8 -p 4063
Glacier2.Server.Endpoints=tcp -h 10.0.0.1
```

As the example shows, the server endpoint does not require a fixed port.

42.4.4 Configuring the Client's Object Adapter

A client that receives callbacks is also a server, and therefore must have an object adapter. Typically, an object adapter has endpoints in the local network, but those endpoints are of no use to a server in our restricted network environment. We

really want the client's callback proxy to contain the router's server endpoints, and we accomplish that by configuring the client's object adapter with a proxy for the router[3]. We supply the router's proxy by creating the object adapter with `createObjectAdapterWithRouter`, or by defining an object adapter property as shown below:

```
CallbackAdapter.Router=Glacier2/router:tcp -h 5.6.7.8 -p 4063
```

For each object adapter, the Ice run time maintains a list of endpoints that are embedded in proxies created by that adapter (see Section 32.4.6). Normally, this list simply contains the local endpoints defined for the object adapter but, when the adapter is configured with a router, the list only contains the router's server endpoints. When using a router, this object adapter allows the client to service callback requests via the router. Because the adapter only contains the router's server endpoints, this means that, if the client also wants to service requests via local (non-routed) endpoints, the client must create a separate adapter for these requests.

42.4.5 Callback Object Identities

Glacier2 assigns a unique category to each client for use in the identities of the client's callback objects. The client creates proxies that contain this identity category for back-end servers to use when making callback requests to the client. This category serves two purposes:

1. Upon receipt of a callback request from a back-end server, the router uses the request's category to identify the intended client.
2. The category is sufficiently random that, without knowing the category in advance, it is practically impossible for a misbehaving or malicious back-end server to send callback requests to an arbitrary client.

A client can obtain its assigned category by calling `getCategoryForClient` on the `Router` interface as shown in the C++ example below:

```
Glacier2::RouterPrx router = // ...
string category = router->getCategoryForClient();
```

3. Note that multiple object adapters created by the same communicator cannot use the same router.

42.4.6 Nested Invocations

If a router client intends to receive callbacks and make nested twoway invocations, it is important that the client be configured correctly. When using the thread pool concurrency model, you must increase the size of the client thread pool to at least two threads. See Section 36.7.5 for more information.

42.4.7 Session Timeout

If the client's session times out, the next invocation raises `ConnectionLostException`. The client can recover from this situation by re-creating the session and re-creating the callback adapter (adding all the callback servants to the ASM of the re-created adapter).

42.4.8 Example

The `demo/Glacier2/callback` example illustrates the use of callbacks with Glacier2. The `README` file in the directory provides instructions on running the example, and comments in the configuration file describe the properties in detail.

42.5 Helper Classes

Ice includes a number of helper classes to help you build robust Glacier2 clients.

42.5.1 The `Glacier2::Application` class

You may already be familiar with the `Ice::Application` class, which encapsulates some basic Ice functionality such as communicator initialization, communicator destruction, and proper handling of signals and exceptions. The `Glacier2::Application` extends `Ice::Application` to add functionality that is commonly needed by Glacier2 clients:

- Keeps a session alive by periodically sending "ping" requests from a background thread
- Automatically restarts a session if a failure occurs
- Optionally creates an object adapter for callbacks
- Destroys the session when the application completes

The C++ definition of `Glacier2::Application` is shown below. (The Java and C# versions offer identical functionality so we do not show them here.)

```
namespace Glacier2 {

class Application : public Ice::Application {
public:
    Application();
    Application(Ice::SignalPolicy policy);

    virtual int runWithSession(int argc, char* argv[]) = 0;

    virtual Glacier2::SessionPrx createSession() = 0;

    virtual void sessionDestroyed();

    static Glacier2::RouterPrx router();

    static Glacier2::SessionPrx session();

    void restart();

    std::string categoryForClient();

    Ice::Identity createCallbackIdentity(const std::string& name);

    Ice::ObjectPrx addWithUUID(const Ice::ObjectPtr& servant);

    Ice::ObjectAdapterPtr objectAdapter();
};

}
```

The following methods are supported:

- `Application()`

  Instantiating the class using its default constructor has the same semantics as calling `Application(Ice::HandleSignals)`.

- `Application(Ice::SignalPolicy policy)`

  This constructor allows you to indicate whether the class should handle signals. The `SignalPolicy` enumeration contains two enumerators: `HandleSignals` and `NoSignalHandling`. If you specify `HandleSignals`, the class automatically shuts down or destroys its communicator upon receipt of certain signals. Refer to the `Ice::Application` documen-

tation in the relevant language mapping chapter for more information on signal handling.

- `int runWithSession(int argc, char* argv[])`

  This method must be overridden by a subclass and represents the "main loop" of the application. It is called after the communicator has been initialized and the Glacier2 session has been established. The argument vector passed to this method contains the arguments passed to `Application::main` with all Ice-related options removed. The implementation of `runWithSession` must return zero to indicate success and non-zero to indicate failure; the value returned by `runWithSession` becomes the return value of `Application::main`.

- `Glacier2::SessionPrx createSession()`

  This method must be overridden by a subclass and implements the logic necessary to create the application's Glacier2 session. A call to `createSession` always precedes a call to `runWithSession`. The application terminates if `createSession` raises (or allows to be raised) an `Ice::LocalException`.

- `void sessionDestroyed()`

  A subclass can optionally override this method to take action when connectivity with the Glacier2 router is lost.

- `Glacier2::RouterPrx router()`

  Returns the proxy for the Glacier2 router.

- `Glacier2::SessionPrx session()`

  Returns the proxy for the current session.

- `void restart()`

  Causes the `Application` class to destroy the current session, create a new session (by calling `createSession`), and start a new main loop (in `runWithSession`). This method does not return but rather raises a `RestartSessionException` that is trapped by the `Application` class.

- `std::string categoryForClient()`

  Returns the category to be used in the identities of all of the client's callback objects. Clients must use this category for the router to forward callback requests to the intended client. The method raises `SessionNotExistException` if no session is currently active.

- `Ice::Identity createCallbackIdentity(`
  `const std::string& name)`

  Creates a new Ice identity for a callback object with the given identity name.

- `Ice::ObjectPrx addWithUUID(`
  `const Ice::ObjectPtr& servant)`

  Adds a servant to the callback object adapter's Active Servant Map using a UUID for the identity name.

- `Ice::ObjectAdapterPtr objectAdapter()`

  Returns the object adapter used for callbacks, creating the object adapter if necessary.

The Ice distribution includes an example in `demo/Glacier2/callback` that shows how to use the `Glacier2::Application` class.

42.5.2 GUI Helper Classes

The "main loop" design imposed by the `Glacier2::Application` class is not suitable for graphical applications, therefore Ice also includes a collection of Java and C# classes that better accommodate the needs of GUI programs:

- `Glacier2.SessionFactoryHelper`

  This class simplifies the task of creating a Glacier2 session. It provides overloaded `connect` methods that support the two authentication styles (username/password and SSL credentials) accepted by the Glacier2 router, and returns an instance of `Glacier2.SessionHelper` for each new session.

- `Glacier2.SessionHelper`

  This class encapsulates a Glacier2 session and provides much of the same functionality as `Glacier2::Application`. If the application supplies an instance of `Glacier2.SessionCallback` when creating the session, `SessionHelper` invokes the callback object about important events in the session's lifecycle.

- `Glacier2.SessionCallback`

  An application can implement this interface to receive notification about session lifecycle events.

The classes are discussed further in the subsections below. You can find sample applications that make use of these classes in the `demo/Glacier2/chat` directory of your Ice distribution.

The `SessionFactoryHelper` Class

The `SessionFactoryHelper` class provides convenience methods for configuring the settings that are commonly used to create a Glacier2 session, such as the router's host and port number. Once the application has completed its configuration, it calls one of the `connect` methods to initialize a communicator, establish a Glacier2 session, and receive a `SessionHelper` object with which it can manage new session. An application should create a new `SessionFactoryHelper` object for each router instance that it uses.

`SessionFactoryHelper` creates an `Ice.InitializationData` object if the application does not pass one to the `SessionFactoryHelper` constructor. `SessionFactoryHelper` also creates a new property set if necessary, and then sets some configuration properties required by Glacier2 clients. The resulting `InitializationData` object is eventually used in `connect` to initialize a new communicator.

The Java definition of `SessionFactoryHelper` is shown below (the C# version is nearly identical and is not shown here):

```
package Glacier2;

public class SessionFactoryHelper {

    public SessionFactoryHelper(SessionCallback callback)
        throws Ice.InitializationException;

    public SessionFactoryHelper(Ice.InitializationData initData,
                                SessionCallback callback)
        throws Ice.InitializationException;

    public SessionFactoryHelper(Ice.Properties properties,
                                SessionCallback callback)
        throws Ice.InitializationException;

    public void setRouterIdentity(Ice.Identity identity);
    public Ice.Identity getRouterIdentity();

    public void setRouterHost(String hostname);
    public String getRouterHost();

    public void setSecure(boolean secure);
    public boolean getSecure();

    public void setTimeout(int timeoutMillisecs);
    public int getTimeout();
```

```
    public void setPort(int port);
    public int getPort();

    public Ice.InitializationData getInitializationData();

    public void setConnectContext(
        java.util.Map<String, String> ctx);

    public SessionHelper connect();
    public SessionHelper connect(String username,
                                 String password);
}
```

The following methods are supported:

- `SessionFactoryHelper(SessionCallback callback)`

  This constructor is useful when your application has no other configuration requirements. The constructor allocates an `InitializationData` object and a new property set. The `callback` argument must not be null.

- `SessionFactoryHelper(`
  `Ice.InitializationData initData,`
  `SessionCallback callback)`

  Use this constructor when you want to prepare your own instance of `InitializationData`. The `callback` argument must not be null.

- `SessionFactoryHelper(Ice.Properties properties,`
  `SessionCallback callback)`

  This constructor is convenient when you wish to supply an initial set of properties. The `callback` argument must not be null.

- `void setRouterIdentity(Ice.Identity identity)`
  `Ice.Identity getRouterIdentity()`

  Determines the object identity of the Glacier2 router. Note that this setting is only used if `Ice.Default.Router` is undefined.

- `void setRouterHost(String hostname)`
  `String getRouterHost()`

  Determines the host name of the Glacier2 router. Note that this setting is only used if `Ice.Default.Router` is undefined.

- `void setSecure(boolean secure)`
  `boolean getSecure()`

  Determines whether the connection to the Glacier2 router must be secure. Note that this setting is only used if `Ice.Default.Router` is undefined.

- `void setTimeout(int timeoutMillisecs)`
  `int getTimeout()`

  Determines the timeout setting (in milliseconds) for the connection to the Glacier2 router. No timeout is used if the argument is less than or equal to zero. Note that this setting is only used if `Ice.Default.Router` is undefined.

- `void setPort(int port)`
  `int getPort()`

  Determines the port on which the Glacier2 router is listening. Note that this setting is only used if `Ice.Default.Router` is undefined.

- `Ice.InitializationData getInitializationData()`

  Returns a reference to the `InitializationData` object that will be used during communicator initialization. If necessary, an application can make modifications to this object prior to calling `connect`.

- `void setConnectContext(`
  `  java.util.Map<String, String> ctx)`

  Sets the request context to be used when creating a session. This method must be invoked prior to `connect`.

- `SessionHelper connect()`

  Initializes a communicator, creates a Glacier2 session using SSL credentials, and returns a new `SessionHelper` object. The `connected` method is invoked on the session callback if the session was created successfully, otherwise the `connectFailed` method is invoked.

- `SessionHelper connect(String username,`
  `                       String password)`

  Initializes a communicator, creates a Glacier2 session using the given user name and password, and returns a new `SessionHelper` object. The `connected` method is invoked on the session callback if the session was created successfully, otherwise the `connectFailed` method is invoked.

The `SessionHelper` Class

The `SessionHelper` class encapsulates a Glacier2 session and keeps the session alive by periodically "pinging" the router. `SessionHelper` also provides several convenience methods for common session-related actions (shown below in Java):

```
package Glacier2;

public class SessionHelper {

    public void destroy();

    public Ice.Communicator communicator();

    public String categoryForClient()
        throws SessionNotExistException;

    public Ice.ObjectPrx addWithUUID(Ice.Object servant)
        throws SessionNotExistException;

    public Glacier2.SessionPrx session()
        throws SessionNotExistException;

    public boolean isConnected();

    public Ice.ObjectAdapter objectAdapter()
        throws SessionNotExistException;
}
```

The following methods are supported:

- `void destroy()`

  Destroys the Glacier2 session.

- `Ice.Communicator communicator()`

  Returns the communicator created by the `SessionHelper`.

- `String categoryForClient()`

  Returns the category that must be used in the identities of all callback objects. Raises `SessionNotExistException` if no session is currently active.

- `Ice.ObjectPrx addWithUUID(Ice.Object servant)`

  Adds a servant to the callback object adapter using a UUID for the identity name. Raises `SessionNotExistException` if no session is currently active.

- `Glacier2.SessionPrx session()`

  Returns a proxy for the Glacier2 session. Raises `SessionNotExistException` if no session is currently active.

- `boolean isConnected()`

  Returns true if the session is currently active, false otherwise.

- `Ice.ObjectAdapter objectAdapter()`

  Returns the callback object adapter, creating it if necessary. Raises `SessionNotExistException` if no session is currently active.

The `SessionCallback` Interface

An application must supply an instance of `SessionCallback` when instantiating a `SessionFactoryHelper` object. The callback methods allow the application to receive notification about events in the lifecycle of the session (shown below in Java):

```
package Glacier2;

public interface SessionCallback {

    void createdCommunicator(SessionHelper session);

    void connected(SessionHelper session)
        throws SessionNotExistException;

    void disconnected(SessionHelper session);

    void connectFailed(SessionHelper session, Throwable ex);
}
```

The following callback methods are supported:

- `void createdCommunicator(SessionHelper session)`

  Called after successfully initializing a communicator.

- `void connected(SessionHelper session)`

  Called after successfully establishing the Glacier2 session. The method can raise `SessionNotExistException` to force the new session to be destroyed.

- `void disconnected(SessionHelper session)`

  Called after the Glacier2 session is destroyed.

- `void connectFailed(SessionHelper session, Throwable ex)`

  Called if a failure occurred while attempting to establish a Glacier2 session.

42.6 Router Security

As a firewall, a Glacier2 router represents a doorway into a private network, and in most cases that doorway should have a good lock. The obvious first step is to use SSL for the router's client endpoints. This allows you to secure the message traffic and restrict access to clients having the proper credentials (see Chapter 41). However, the router takes security even further by providing access control and filtering capabilities.

42.6.1 Access Control

The authentication capabilities of SSL may not be sufficient for all applications: the certificate validation phase of the SSL handshake verifies that the user is who he says he is, but how do we know that he should be allowed to use the router? Glacier2 addresses this issue through the use of an access control facility that supports two forms of authentication: passwords and certificates. You can configure the router to use whichever authentication method is most appropriate for your application, or you can configure both methods in the same router.

Password Authentication

The router verifies the user name and password arguments to its `createSession` operation before it forwards any requests on behalf of the client. Given that the password is sent "in the clear," it is important to protect these values by using an SSL connection with the router. Section 42.3.6 demonstrates how to use the `createSession` operation.

There are two ways for the router to verify a user name and password. By default, the router uses a file-based access control list, but you can override this behavior by installing a proxy for an application-defined verifier object. Configuration properties define the password file name or the verifier proxy; if you install a verifier proxy, the password file is ignored. Since we have already discussed the password file in Section 42.3.2, we will focus on the custom verifier interface in the remainder of this section.

An application that has special requirements can implement the interface `Glacier2::PermissionsVerifier` to gain programmatic control over access to a router. This can be especially useful in situations where a repository of account information already exists (such as an LDAP directory), in which case duplicating that information in another file would be tedious and error-prone.

The Slice definition for the interface contains just one operation:

```
module Glacier2 {
    interface PermissionsVerifier {
        idempotent
        bool checkPermissions(string userId, string password,
                              out string reason);
    };
};
```

The router invokes `checkPermissions` on the verifier object, passing it the user name and password arguments that were given to `createSession`. The operation must return true if the arguments are valid, and false otherwise. If the operation returns false, a reason can be provided in the output parameter.

To configure a router with a custom verifier, set the configuration property `Glacier2.PermissionsVerifier` with the proxy for the object.

In situations where authentication is not necessary, such as during development or when running in a trusted environment, you can use Glacier2's built-in "null" permissions verifier. This object accepts any combination of username and password, and you can enable it with the following property definition:

```
Glacier2.PermissionsVerifier=Glacier2/NullPermissionsVerifier
```

Note that the category of the object's identity (`Glacier2` in this example) must match the value of the property `Glacier2.InstanceName`.

A sample implementation of the `PermissionsVerifier` interface is provided in the `demo/Glacier2/callback` directory.

Certificate Authentication

As shown in Section 42.3.6, the `createSessionFromSecureConnection` operation does not require a user name or password because the client's SSL connection to the router already supplies the credentials necessary to sufficiently identify the client, in the form of X.509 certificates. (See Chapter 41 for details on IceSSL configuration.)

It is up to you to decide what constitutes sufficient identification. For example, a single certificate could be shared by all clients if there is no need to distinguish between them, or you could generate a unique certificate for each client or a group

of clients. Glacier2 does not enforce any particular policy, but simply delegates the decision of whether to accept the client's credentials to an application-defined object that implements the `Glacier2::SSLPermissionsVerifier` interface:

```
module Glacier2 {
    interface SSLPermissionsVerifier {
        idempotent bool authorize(SSLInfo info,
                                  out string reason);
    };
};
```

Router clients may only use `createSessionFromSecureConnection` if the router is configured with a proxy for an `SSLPermissionsVerifier` object. The implementation of `authorize` must return true to allow the client to establish a session. To reject the session, `authorize` must return false and may optionally provide a value for `reason`, which is returned to the client as a member of `PermissionDeniedException`.

The verifier examines the members of `SSLInfo` to authenticate a client:

```
module Glacier2 {
    struct SSLInfo {
        string remoteHost;
        int remotePort;
        string localHost;
        int localPort;
        string cipher;
        Ice::StringSeq certs;
    };
};
```

The structure includes address information about the remote and local hosts, and a string that describes the ciphersuite negotiated for the SSL connection between the client and the router. These values are generally of interest for logging purposes, whereas the `certs` member supplies the information the verifier needs to make its decision. The client's certificate chain is represented as a sequence of strings that use the Privacy Enhanced Mail (PEM) encoding. The first element of the sequence corresponds to the client's certificate, followed by its signing certificates. The certificate of the root Certificate Authority (CA) is the last element of the sequence. An empty sequence indicates that the client did not supply a certificate chain.

Although the certificate chain has already been validated by the SSL implementation, a verifier implementation typically needs to examine it in detail before making its decision. As a result, the verifier will need to convert the contents of

`certs` into a more usable form. Some Ice platforms, such as Java and .NET 2, already provide certificate abstractions, and IceSSL supplies its own for C++ users. IceSSL for Java and .NET 2 defines the method `IceSSL.Util.createCertificate`, which accepts a PEM-encoded string and returns an instance of the platform's certificate class. In C++, the class `IceSSL::Certificate` has a constructor that accepts a PEM-encoded string. Chapter 41 provides the relevant details.

In addition to examining certificate attributes such as the distinguished name of the subject and issuer, it is also important that a verifier consider the length of the certificate chain. Refer to Section 41.4.6 for a discussion of this issue.

To install your verifier, set the `Glacier2.SSLPermissionsVerifier` property with the proxy of your verifier object.

In situations where authentication is not necessary, such as during development or when running in a trusted environment, you can use Glacier2's built-in "null" permissions verifier. This object accepts the credentials of any client, and you can enable it with the following property definition:

```
Glacier2.SSLPermissionsVerifier=\
    Glacier2/NullSSLPermissionsVerifier
```

Note that the category of the object's identity (`Glacier2` in this example) must match the value of the property `Glacier2.InstanceName`.

Interaction with a Permissions Verifier

The router attempts to contact the configured permissions verifiers at startup. If an object is unreachable, the router logs a warning message but continues its normal operation (you can suppress the warning using the `--nowarn` option – see Section 42.3.3). The router does not contact a verifier again until it needs to invoke an operation on the object. For example, when a client asks the router to create a new session, the router makes another attempt to contact the verifier; if the object is still unavailable, the router logs a message and returns `PermissionDeniedException` to the client.

Obtaining SSL Credentials

Servers that wish to receive information about a client's SSL connection to the router can define the `Glacier2.AddConnectionContext` property (see Section D.18). When enabled, the router adds several entries to the request context of each invocation it forwards to a server, providing information such as the client's encoded certificate (if supplied) and addressing details.

If the client's connection uses SSL, the router defines the `_con.peerCert` entry in the context. A server can check for the presence of this entry and also extract additional context entries as shown below in this C++ example:

```
void unlockDoor(string id, const Ice::Current& curr)
{
    Ice::Context::const_iterator i = curr.ctx.find("_con.peerCert"
);
    if (i != curr.ctx.end()) {
        string certPEM;
        certPEM = i->second;
        cout << "Client address = "
             << curr.ctx["_con.remoteAddress"]
             << ":" << curr.ctx["_con.remotePort"] << endl;
        ...
    }
    ...
}
```

If the client supplied a certificate, the server can decode and examine it using the techniques discussed in Chapter 41.

42.6.2 Filtering

The Glacier2 router is capable of filtering requests based on a variety of criteria, which helps to ensure that clients do not gain access to unintended objects.

Address Filters

To prevent a client from accessing arbitrary back-end hosts or ports, you can configure a Glacier2 router to validate the address information in each proxy the client attempts to use. Two properties determine the router's filtering behavior:

- `Glacier2.Filter.Address.Accept`

  An address is accepted if it matches an entry in this property and does not match an entry in `Glacier2.Filter.Address.Reject`.

- `Glacier2.Filter.Address.Reject`

  An address is rejected if it matches an entry in this property.

The value of each property is a list of *address*:*port* pairs separated by spaces, as shown in the example below:

```
Glacier2.Filter.Address.Accept=192.168.1.5:4063 192.168.1.6:4063
```

This configuration allows clients to use only two hosts in the back-end network, and only one port on each host. A client that attempts to use a proxy containing any other host or port receives an `ObjectNotExistException` on its initial request.

You can also use ranges, groups and wildcards when defining your address filters. For example, the following property value shows how to use an address range:

```
Glacier2.Filter.Address.Accept=192.168.1.[5-6]:4063
```

This property is equivalent to the first example, but the range notation allows us to define the filter more concisely. Similarly, we can restate the property using the group notation by separating values with a comma:

```
Glacier2.Filter.Address.Accept=192.168.1.[5,6]:4063
```

The wildcard notation uses the `*` character to substitute for a value:

```
Glacier2.Filter.Address.Accept=10.0.*.1:4063
```

The range, group, and wildcard notation is also supported when specifying ports, as shown below:

```
Glacier2.Filter.Address.Accept=192.168.1.[5,6]:[10000-11000]
Glacier2.Filter.Address.Reject=192.168.1.[5,6]:[10500,10501]
```

In this configuration, the router allows clients to access all of the ports in the range `10000` to `11000`, except for the two ports `10500` and `10501`.

At first glance, you might think that the following property definition is pointless because it would prevent clients from accessing any back-end server:

```
Glacier2.Filter.Address.Reject=*
```

In reality, this configuration only prevents clients from accessing servers using direct proxies, that is, proxies that contain endpoints. As a result, the property causes Glacier2 to accept only indirect proxies (see Section 2.2.2).

NOTE: By default, a Glacier2 router forwards requests for any address, which is equivalent to defining the property `Glacier2.Filter.Address.Accept=*`.

Category Filters

As described in Section 32.5, the `Ice::Identity` type contains two string members: `category` and `name`. You can configure a router with a list of valid identity categories, in which case it only routes requests for objects in those categories.

The configuration property `Glacier2.Filter.Category.Accept` supplies the category list:

```
Glacier2.Filter.Category.Accept=cat1 cat2
```

This property does not affect the routing of callback requests from back-end servers to router clients. See Section 42.4 for more information on callbacks.

NOTE: By default a Glacier2 router forwards requests for any category.

If a category contains spaces, you can enclose the value in single or double quotes. If a category contains a quote character, it must be escaped with a leading backslash.

Glacier2 can optionally manipulate the category filter automatically. When you set `Glacier2.Filter.Category.AcceptUser` to a value of `1`, the router adds the session's username (for password authentication) or distinguished name (for SSL authentication) to the list of accepted categories. To ensure the uniqueness of your categories, you may prefer setting the property to a value of `2`, which causes the router to prepend an underscore to the username or distinguished name before adding it to the list.

A session manager can also configure category filters dynamically; see Section 42.8 for details.

Identity Filters

The ability to filter on identity categories, as described in the previous section, is a convenient way to limit clients to particular groups of objects. For even stricter control over the identities that clients are allowed to access, you can use the `Glacier2.Filter.Identity.Accept` property. The value of this property is a list of identities, separated by whitespace, representing the *only* objects the router's clients may use.

If an identity contains spaces, you can enclose the value in single or double quotes. If an identity contains a quote character, it must be escaped with a leading backslash.

Clearly, specifying a static list of identities is only practical for a small set of objects. Furthermore, in many applications, the complete set of identities cannot be known in advance, such as when objects are created on a per-session basis and use UUIDs in their identities. For these situations, category-based filtering is generally sufficient. However, a session manager can also use Glacier2's dynamic

filtering interface, `SessionControl`, to manage the set of valid identities at run time. See Section 42.8 for more information.

Adapter Filters

Applications often use IceGrid in their back-end network to simplify server administration and take advantage of the benefits offered by indirect proxies. Once you have configured Glacier2 with an appropriate locator (see Section 42.13), clients can use indirect proxies to refer to objects in IceGrid-managed servers. Recall from Section 2.2.2 that an indirect proxy comes in two forms: one that contains only an identity, and one that contains an identity and an object adapter identifier. You can use the category and identity filters described in previous sections to control identity-only proxies, and you can use the property `Glacier2.Filter.AdapterId.Accept` to enforce restrictions on indirect proxies that use an object adapter identifier.

For example, the following property definition allows a client to use the proxy `factory@WidgetAdapter` but not the proxy `factory@SecretAdapter`:

```
Glacier2.Filter.AdapterId.Accept=WidgetAdapter
```

If an adapter identifier contains spaces, you can enclose the value in single or double quotes. If an adapter identifier contains a quote character, it must be escaped with a leading backslash.

A session manager can also configure this filter dynamically, as described in Section 42.8.

Proxy Filters

The Glacier2 router maintains an internal routing table that contains an entry for each proxy used by a router client; the size of the routing table grows in proportion to the number of clients and their proxy usage. Furthermore, the amount of memory that the routing table consumes is affected by the number of endpoints in each proxy. Glacier2 provides two properties that you can use to limit the size of the routing table and defend against malicious router clients.

The property `Glacier2.RoutingTable.MaxSize` specifies the maximum number of entries allowed in the routing table. If the size of the table exceeds the value of this property, the router evicts older entries on a least-recently-used basis. (Eviction of proxies from the routing table is transparent to router clients.) The default size of the routing table is 1000, but you may need to define a different value depending on the needs of your application. While experimenting with different values, you may find it useful to define the property

`Glacier2.Trace.RoutingTable` to see a log of the router's activities with respect to the routing table.

The property `Glacier2.Filter.ProxySizeMax` sets a limit on the size of a stringified proxy. The Ice run time places no limits on the size of proxy components such as identities and host names, but a malicious client could manufacture very large proxies in a denial-of-service attack on a Glacier2 router. By setting this property to a reasonably small value, you can prevent proxies from consuming excessive memory in the router process.

Client Impact

The Glacier2 router immediately terminates a client's session if it attempts to use a proxy that is rejected by an address filter or exceeds the size limit defined by the property `Glacier2.Filter.ProxySizeMax`. The Ice run time in the client responds by raising `ConnectionLostException` to the application.

For category, identity, and adapter identifier filters, the router raises `ObjectNotExistException` if any of the filters rejects a proxy and none of the filters accepts it.

To obtain more information on the router's reasons for terminating a session or rejecting a request, set the following property and examine the router's log output:

```
Glacier2.Client.Trace.Reject=1
```

42.6.3 Administration

Glacier2 supports an administrative interface that allows you to shut down a router programmatically:

```
module Glacier2 {
    interface Admin {
        idempotent void shutdown();
    };
};
```

To prevent unwanted clients from using the `Admin` interface, the object is only accessible on the endpoints defined by the `Glacier2.Admin.Endpoints` property. This property has no default value, meaning the `Admin` interface is inaccessible unless you explicitly define it.

If you decide to define `Glacier2.Admin.Endpoints`, choose your endpoints carefully. We generally recommend the use of endpoints that are accessible only from behind a firewall.

42.7 Session Management

A Glacier2 router requires a client to create a session (see Section 42.3.6) and forwards requests on behalf of the client until its session expires. A session expires when it is explicitly destroyed, or when it times out due to inactivity.

If your application needs to track the session activities of a router, you can configure the router to use a custom session manager. For example, your application may need to acquire resources and initialize the state of back-end services for each new session, and later reclaim those resources when the session expires.

As with the authentication facility described in Section 42.6, Glacier2 provides two session manager interfaces that an application can implement. The `SessionManager` interface receives notifications about sessions that use password authentication, while the `SSLSessionManager` interface is for sessions authenticated using SSL certificates.

42.7.1 The Session Manager Interfaces

The relevant Slice definitions are shown below:

```
module Glacier2 {
    exception CannotCreateSessionException {
        string reason;
    };

    interface Session {
        void destroy();
    };

    interface SessionManager {
        Session* create(string userId, SessionControl* control)
            throws CannotCreateSessionException;
    };

    interface SSLSessionManager {
        Session* create(SSLInfo info, SessionControl* control)
            throws CannotCreateSessionException;
    };
};
```

When a client invokes `createSession` on the `Router` interface (see Section 42.3.6), the router validates the client's user name and password and then calls `SessionManager::create`. Similarly, a call to `createSessionFromSecure-`

`Connection` causes the router to invoke `SSLSessionManager::create`. The `SSLInfo` structure is described in Section 42.6.1. The second argument to the `create` operations is a proxy for a `SessionControl` object, which a session can use to perform dynamic filtering (see Section 42.8).

The `create` operations must return the proxy of a new `Session` object, or raise `CannotCreateSessionException` and provide an appropriate reason. The `Session` proxy returned by `create` is ultimately returned to the client as the result of `createSession` or `createSessionFromSecureConnection`.

Glacier2 invokes the `destroy` operation on a `Session` proxy when the session expires. This provides a custom session manager with the opportunity to reclaim resources that were acquired for the session during `create`.

NOTE: The `create` operations may be called with information that identifies an existing session. For example, this can occur if a client has lost its connection to the router and therefore must create a new session but its previous session has not expired yet, and the router therefore has not yet invoked `destroy` on its `Session` proxy. A session manager implementation must be prepared to handle this situation.

To configure the router with a custom session manager, define the properties `Glacier2.SessionManager` or `Glacier2.SSLSessionManager` with the proxies of the session manager objects. If necessary, you can configure a router with proxies for both types of session managers. If a session manager proxy is not supplied, the call to `createSession` or `createSessionFromSecureConnection` always returns a null proxy.

The router attempts to contact the configured session manager at startup. If the object is unreachable, the router logs a warning message but continues its normal operation (you can suppress the warning using the `--nowarn` option – see Section 42.3.3). The router does not contact the session manager again until it needs to invoke an operation on the object. For example, when a client asks the router to create a new session, the router makes another attempt to contact the session manager; if the session manager is still unavailable, the router logs a message and returns `CannotCreateSessionException` to the client.

A sample implementation of the `SessionManager` interface is provided in the `demo/Glacier2/callback` directory.

42.7.2 Session Timeouts

The value of the `Glacier2.SessionTimeout` property specifies the number of seconds a session must be inactive before it expires. If the property is not defined, then sessions never expire due to inactivity. If a non-zero value is specified, it is very important that the application chooses a value that does not result in premature session expiration. For example, if it is normal for a client to create a session and then have long periods of inactivity, then a suitably long timeout must be chosen, or the client must actively keep its session alive, or timeouts must be disabled altogether.

Once a session has expired (or been destroyed for some other reason), the client will no longer be able to send requests via the router, but instead receives a `ConnectionLostException`. The client must explicitly create a new session in order to continue using the router. If necessary, clients can use a dedicated thread to keep their sessions alive (see Section 42.3.6).

In general, we recommend the use of an appropriate session timeout, otherwise resources created for each session will accumulate in the router. However, you can safely disable the session timeout if the server regularly calls back to the client. In that case, Glacier2 will automatically destroy the session if a failure occurs while forwarding a server callback to the client.

42.7.3 Invocation Timeouts

If you require invocation timeouts in the client for routed proxies, you must set the timeout on the router that you use to establish the session. This is because the Ice run time forwards the invocation to Glacier2, and the timeout applies to that invocation.

In other words, whatever timeout you set on the router you use to create the session is the timeout that applies to all routed proxies. Do not attempt to override the timeout on a per-proxy basis; if you do, any setting other than the timeout used to establish the session results in a `ConnectionLostException`. This is because proxies with different timeout values establish separate connections, but there can be only one connection to Glacier2.[4]

For invocations made by Glacier2 to the server, whatever timeout value is set on the *first* proxy that is used to make an invocation applies to all proxies for the same object. This is because Glacier2 adds the proxy to its routing table during the

4. A future version of Ice may make it illegal to set a timeout on a routed proxy.

first invocation and, thereafter, reuses that cached proxy for all invocations to the same object identity.

Here is an example to illustrate this:

```
// 10-second session timeout for router.
ObjectPrx router = communicator->stringToProxy(
    "Glacier2/router:tcp -h host1 -p 4063 -t 10000");
communicator->setDefaultRouter(
    RouterPrx::uncheckedCast(router));

// Ping with 20-second timeout
communicator->stringToProxy("id:tcp -h host2 -p 12345 -t 20000")
    ->ice_ping();

// Ping with 30-second timeout
communicator->stringToProxy("id:tcp -h host2 -p 12345 -t 30000")
    ->ice_ping();
```

In this case, all invocations made by the client use a 10-second timeout to forward the invocations to Glacier2. The first call to `ice_ping`, when forwarded by Glacier2 to the server, uses a 20-second timeout. The second call to `ice_ping` also uses a 20-second timeout, even though the proxy specifies a 30-second timeout.

If you have a timeout on both the client–Glacier2 and the Glacier2–server connections, the timeout on the client–Glacier2 connection should be slightly longer; otherwise, invocation timeouts that are encountered by Glacier2 when it forwards an operation to the server cannot be propagated back to the client.

42.7.4 Connection Caching

Glacier2 disables connection caching on session manager proxies, therefore if you configure the router with a session manager proxy that contains multiple endpoints, the router attempts to use a different endpoint for each invocation on a session manager. The purpose of this behavior is to distribute the load among multiple active session manager objects without using the replication features provided by IceGrid. Be aware that including an invalid endpoint in your session manager proxy, such as the endpoint of a session manager server that is not currently running, can cause router clients to experience delays during session creation.

If your session managers are in a replica group, Section 42.13.2 provides more information on the router's caching behavior.

42.8 Dynamic Filtering

Section 42.6.2 described various ways of statically configuring a router to filter requests. Glacier2 also allows a session manager to customize filters for each session at run time via its `SessionControl` interface:

```
module Glacier2 {
    interface SessionControl {
        StringSet* categories();
        StringSet* adapterIds();
        IdentitySet* identities();
        void destroy();
    };
};
```

The router creates a `SessionControl` object for each client session and supplies a proxy for the object to the session manager `create` operations (see Section 42.7). Note that the `SessionControl` proxy is null unless the router is configured with server endpoints; refer to Section 42.4.3 for an example of configuring these endpoints.

Invoking the `destroy` operation causes the router to destroy the client's session, which eventually results in an invocation of `destroy` on the application-defined `Session` object, if one was provided.

The interface operations `categories`, `adapterIds` and `identities` return proxies to objects representing the modifiable filters for the session. The router initializes these filters using their respective static configuration properties.

The `SessionControl` object uses a `StringSet` to manage the category and adapter identifier filters:

```
module Glacier2 {
    interface StringSet {
        idempotent void add(Ice::StringSeq additions);
        idempotent void remove(Ice::StringSeq deletions);
        idempotent Ice::StringSeq get();
    };
};
```

Similarly, the `IdentitySet` interface manages the identity filters:

```
module Glacier2 {
    interface IdentitySet {
        idempotent void add(Ice::IdentitySeq additions);
        idempotent void remove(Ice::IdentitySeq deletions);
        idempotent Ice::IdentitySeq get();
    };
};
```

In both interfaces, the `add` operation silently ignores duplicates, and the `remove` operation silently ignores non-existent entries.

Dynamic filtering is often necessary when each session must be restricted to a particular group of objects. Upon session creation, a session manager typically allocates a number of objects in back-end servers for that session to use. To prevent other sessions from accessing these objects (intentionally or not), the session manager can configure the session's filters so that it is only permitted to use the objects that were created for it.

For example, a session manager can retain the `SessionControl` proxy and add a new identity to the `IdentitySet` as each new object is created for the session. A simpler solution is to create a unique identifier for the session, add it to the session's category filter, and use that category in the identities of all of the objects accessible by that session. Using a category filter in this way reserves an identity namespace for each session and avoids the need to update the filter for each new object.

To aid in logging and debugging, you can select a category that identifies the client, such as the user name that was supplied during session creation, or an attribute of the client's certificate such as the common name, as long as the selected category is sufficiently unique that it will not conflict with another client's session. You must also ensure that the categories you assign to sessions never match the categories of back-end objects that are not meant to be accessed by router clients. As an example, consider the following session manager implementation:

```
class SessionManagerI : public Glacier2::SessionManager
{
public:

    virtual Glacier2::SessionPrx
    create(const string& username,
           const Glacier2::SessionControlPrx& ctrl,
           const Ice::Current& curr)
    {
        string category = "_" + username;
```

```
        ctrl->categories()->add(category);
        // ...
    }
};
```

This session manager derives a category for the session by prepending an underscore to the user name and then adds this category to the session's filter. As long as our back-end objects do not use a leading underscore in their identity categories, this guarantees that a session's category can never match the category of a back-end object.

For your convenience, Glacier2 already includes support for automatic category filtering. See the discussion of category filters on page 1734 for more information.

42.9 Request Buffering

A Glacier2 router can forward requests in buffered or unbuffered mode. In addition, the buffering mode can be set independently for each direction (client-to-server and server-to-client).

The configuration properties `Glacier2.Client.Buffered` and `Glacier2.Server.Buffered` govern the buffering behavior. The former affects buffering of requests from clients to servers, and the latter affects buffering of requests from servers to clients. If a property is not specified, the default value is `1`, which enables buffering. A property value of `0` selects the unbuffered mode.

The primary difference between the two modes is in the way requests are forwarded:

- Buffered

  The router queues incoming requests and outgoing replies for delivery in a separate thread.

- Unbuffered

  The router forwards requests in the same thread that received the request.

Although unbuffered mode consumes fewer resources than buffered mode, certain features such as request overriding (see Section 42.10.2) and request batching (see Section 42.10.3) are available only in buffered mode.

42.10 Request Contexts

The Glacier2 router examines the context of an incoming request (see Section 32.12) for special keys that affect how the router forwards the request. These contexts have the same semantics regardless of whether the request is sent from client to server or from server to client.

42.10.1 `_fwd`

The `_fwd` context determines the proxy mode that the router uses when forwarding the request. The value associated with the `_fwd` key must be a string containing one or more of the characters shown in Table 42.1.

Table 42.1. Legal values for `_fwd` context key.

Value	Mode
`d`	Datagram
`D`	Batch datagram
`o`	Oneway
`O`	Batch oneway
`s`	Secure
`t`	Twoway
`z`	Compress

These characters match the stringified proxy options described in Appendix E.

For requests whose `_fwd` context specify a batch mode, the forwarding behavior of the router depends on whether it is buffering requests (see Section 42.10.3).

If the `_fwd` key is not present in a request context, the mode used by the router to forward that request depends on the mode used by the client's proxy and the router's own configuration. If the client used twoway mode, the router also uses twoway mode. If the client sent the request as a oneway or batch oneway, the

router's behavior is determined by the configuration properties described in Section 42.10.3.

42.10.2 _ovrd

In buffered mode, the router allows a new incoming request to override any pending requests that are still in the router's queue, effectively replacing any pending requests with the new request. For a new request to override a pending request, both requests must meet the following criteria:

- they specify the `_ovrd` key in the request context with the same value
- they are oneway requests
- they are requests on the same object.

This feature is intended to be used by clients that are sending frequent oneway requests in which the most recent request takes precedence. This feature minimizes the number of requests that are forwarded to the server when requests are sent frequently enough that they accumulate in the router's queue before the router has a chance to process them.

Note that the properties `Glacier2.Client.SleepTime` and `Glacier2.Server.SleepTime` can be used to add a delay to the router once it has sent all pending requests (see page 1747). Setting a delay increases the likelihood of overrides actually taking effect.

42.10.3 Batch Requests

Clients can direct the router to forward oneway requests in batches by including the `D` or `O` characters in the `_fwd` context, as described in Section 42.10.1. If the router is configured for buffered mode and several such requests accumulate in its queue, the router forwards them together in a batch rather than as individual requests. See Section 32.16 for more information on batched invocations.

In addition, the properties `Glacier2.Client.AlwaysBatch` and `Glacier2.Server.AlwaysBatch` determine whether oneway requests are always batched regardless of the `_fwd` context. The former property affects requests from clients to servers, while the latter affects requests from servers to clients. If a property is defined with a non-zero value, then all requests whose `_fwd` context includes the `o` character or were sent as oneway invocations are treated as if `O` were specified instead, and are batched when possible. Likewise, requests whose `_fwd` context includes the `d` character or were sent as datagram invocations are treated as if `D` were specified.

If a property is not defined, the router does not batch requests unless specifically directed to do so by the `_fwd` context.

The configuration properties `Glacier2.Client.SleepTime` and `Glacier2.Server.SleepTime` can be used to force the router's delivery threads to sleep for the specified number of milliseconds after the router has sent all of its pending requests. (Incoming requests are queued during this period.) The delay is useful to increase the effectiveness of batching because it makes it more likely for additional requests to accumulate in a batch before the batch is sent.

If these properties are not defined, or their value is zero, the corresponding thread does not sleep after sending queued requests.

42.10.4 Context Forwarding

The configuration properties `Glacier2.Client.ForwardContext` and `Glacier2.Server.ForwardContext` determine whether the router includes the context when forwarding a request. The former property affects requests from clients to servers, while the latter affects requests from servers to clients. If a property is not defined or has the value zero, the router does not include the context when forwarding requests.

42.11 Firewalls

The Glacier2 router requires only one external port to receive connections from clients and therefore can easily coexist with a network firewall device. For example, consider the network shown in Figure 42.7.

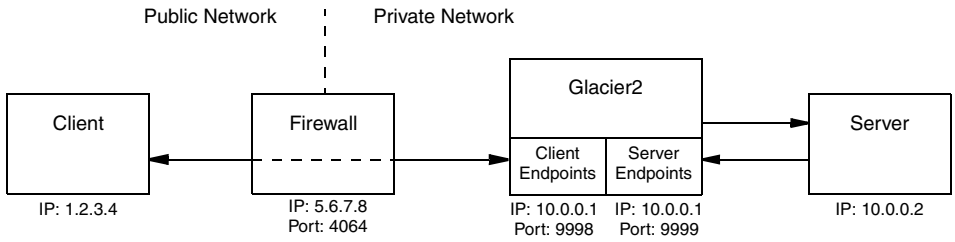


Figure 42.7. Using Glacier2 with a network firewall.

In contrast to Figure 42.6, the Glacier2 router in this example has both of its endpoints in the private network and its host requires only one IP address. We

assume that the firewall has been configured to forward connections from port `4064` to the router's client endpoint at port `9998`. Meanwhile, the client must be configured to use the firewall's address information in its router proxy, as shown below:

```
Ice.Default.Router=Glacier2/router:ssl -h 5.6.7.8 -p 4064
```

The Glacier2 router configuration for this example requires the following properties:

```
Glacier2.Client.Endpoints=ssl -h 10.0.0.1 -p 9998
Glacier2.Server.Endpoints=tcp -h 10.0.0.1 -p 9999
```

Note that the server endpoint specifies a fixed port (`9999`), but the router does not require a fixed port in this endpoint to operate properly.

42.12 Advanced Client Configurations

This section discusses strategies that Glacier2 clients can use to address more advanced requirements.

42.12.1 Object Adapter Strategies

An application that needs to support callback requests from a router as well as requests from local clients should use multiple object adapters. This strategy ensures that proxies created by these object adapters contain the appropriate endpoints. For example, suppose we have the network configuration shown in

Figure 42.8. Notice that the two local area networks use the same private network addresses, which is not an unrealistic scenario.

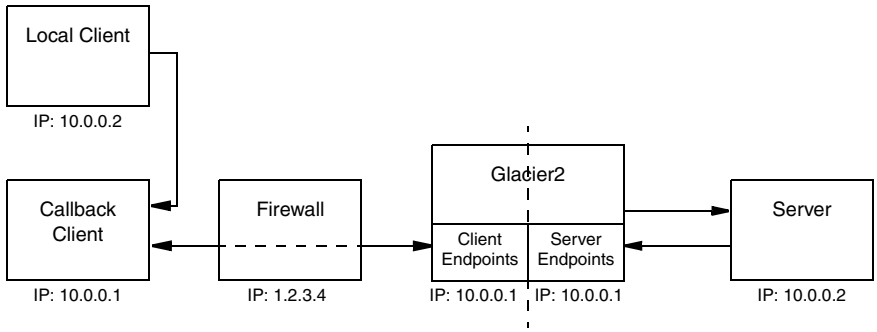


Figure 42.8. Supporting callback and local requests.

Now, if the callback client were to use a single object adapter for handling both callback requests and local requests, then any proxies created by that object adapter would contain the application's local endpoints as well as the router's server endpoints. As you might imagine, this could cause some subtle problems.

1. When the local client attempts to establish a connection to the callback client via one of these proxies, it might arbitrarily select one of the router's server endpoints to try first. Since the router's server endpoints use addresses in the same network, the local client attempts to make a connection over the local network, with two possible outcomes: the connection attempts to those endpoints fail, in which case they are skipped and the real local endpoints are attempted; or, even worse, one of the endpoints might accidentally be valid in the local network, in which case the local client has just connected to some unknown server.

2. The server may encounter similar problems when attempting to establish a local connection to the router in order to make a callback request.

The solution is to dedicate an object adapter solely to handling callback requests, and another one for servicing local clients. The object adapter dedicated to callback requests must be configured with the router proxy as described in Section 42.4.4.

42.12.2 Using Multiple Routers

A client is not limited to using only one router at a time: the proxy operation `ice_router` allows a client to configure its routed proxies as necessary. With respect to callbacks, a client must create a new callback object adapter for each router that can forward callback requests to the client.

For information on configuring multiple routers, see Section 42.3.5.

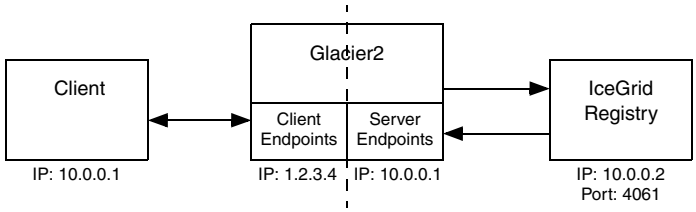
42.13 IceGrid Integration

IceGrid is a server activation and location service, as described in Chapter 38. This section describes the ways in which you can integrate Glacier2 and IceGrid.

42.13.1 Configuring Clients

It is not uncommon for a Glacier2 client to require access to a locator service such as IceGrid. A locator client would typically define the property `Ice.Default.Locator` with a stringified proxy for the locator service, as described in Section 38.4.3. However, when that locator service is accessed via a Glacier2 router, the configuration requirements are slightly different. It is no longer necessary for the client to define `Ice.Default.Locator`; this property must be defined in the Glacier2 router's configuration instead.

For example, consider the following network architecture:



In this case the Glacier2 router's configuration must include the property shown below:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h 10.0.0.2 -p 4061
```

42.13.2 Locator Timeouts

An IceGrid application might want to use replication (see Section 38.9) to increase the availability of Glacier2 session managers. When you configure an indirect proxy for a session manager (and configure Glacier2 with a locator proxy, as described in Section 42.13.1), the Ice run time in the router queries the locator to obtain a proxy for a session manager replica. By default, this proxy is cached for 10 minutes, meaning the router uses the same session manager proxy to create sessions for a 10-minute period, after which it queries the locator again. If you want to distribute the session-creation load among the session manager replicas more evenly, you can decrease the locator cache timeout using configuration properties. For example, the following settings use a timeout of 30 seconds:

```
Glacier2.SessionManager.LocatorCacheTimeout=30
Glacier2.SSLSessionManager.LocatorCacheTimeout=30
```

As you can see, timeouts are specified individually for the `SessionManager` and `SSLSessionManager` proxies. You can also disable caching completely by using a value of 0, in which case the router queries the locator before every invocation on a session manager.

See Section 42.7.4 for more information on connection caching.

42.14 Summary

Complex network environments are a fact of life. Unfortunately, the cost of securing an enterprise's network is increased application complexity and administrative overhead. Glacier2 helps to minimize these costs by providing a low-impact, efficient and secure router for Ice applications.

Chapter 43
IceBox

43.1 Chapter Overview

In this chapter we present IceBox, an easy-to-use framework for Ice application services. Section 43.2 provides an overview of IceBox and the advantages of using it. A tutorial on writing and configuring an IceBox service is presented in Section 43.3, and Section 43.4 describes how to start the server. IceBox administration is the subject of Section 43.5.

43.2 Introduction

The Service Configurator pattern [7] is a useful technique for configuring services and centralizing their administration. In practical terms, this means services are developed as dynamically-loadable components that can be configured into a general purpose "super server" in whatever combinations are necessary. IceBox is an implementation of the Service Configurator pattern for Ice services.

A generic IceBox server replaces the typical monolithic Ice server you normally write. The IceBox server is configured via properties with the application-specific services it is responsible for loading and managing, and it can be administered remotely. There are several advantages in using this architecture:

- Services loaded by the same IceBox server can be configured to take advantage of Ice's collocation optimizations. For example, if one service is a client of another service, and those services reside in the same IceBox server, then invocations between them can be optimized.
- Composing an application consisting of various services is done by configuration, not by compiling and linking. This decouples the service from the server, allowing services to be combined or separated as needed.
- Multiple Java services can be active in a single instance of a Java Virtual Machine (JVM). This conserves operating system resources when compared to running several monolithic servers, each in its own JVM.
- Services implement an IceBox service interface, providing a common framework for developers and a centralized administrative facility.
- IceBox support is integrated into IceGrid, the server activation and deployment service (see Section 38.8).

43.3 Developing a Service

Writing an IceBox service requires implementing one of the IceBox service interfaces. The sample implementations we present in this section implement `IceBox::Service`, shown below:

```
module IceBox {
local interface Service {
    void start(string name,
               Ice::Communicator communicator,
               Ice::StringSeq args);
    void stop();
};
};
```

As you can see, a service needs to implement only two operations, `start` and `stop`. These operations are invoked by the server; `start` is called after the service is loaded, and `stop` is called when the IceBox server is shutting down.

The `start` operation is the service's opportunity to initialize itself; this typically includes creating an object adapter and servants. The `name` and `args` parameters supply information from the service's configuration (see Section 43.3.4), and the `communicator` parameter is an `Ice::Communicator` object created by the server for use by the service. Depending on the service configuration, this

communicator instance may be shared by other services in the same IceBox server, therefore care should be taken to ensure that items such as object adapters are given unique names.

The `stop` operation must reclaim any resources used by the service. Generally, a service deactivates its object adapter, and may also need to invoke `waitForDeactivate` on the object adapter in order to ensure that all pending requests have been completed before the clean up process can proceed. The server is responsible for destroying the communicator instance that was passed to `start`.

Whether the service's implementation of `stop` should explicitly destroy its object adapter depends on other factors. For example, the adapter should be destroyed if the service uses a shared communicator, especially if the service could eventually be restarted. In other circumstances, the service can allow its adapter to be destroyed as part of the communicator's destruction.

These interfaces are declared as `local` for a reason: they represent a contract between the server and the service, and are not intended to be used by remote clients. Any interaction the service has with remote clients is done via servants created by the service.

43.3.1 C++ Service Example

The example we present here is taken from the `IceBox/hello` sample program provided in the Ice distribution.

The class definition for our service is quite straightforward, but there are a few aspects worth mentioning:

```
#include <IceBox/IceBox.h>

#if defined(_WIN32)
#   define HELLO_API __declspec(dllexport)
#else
#   define HELLO_API /**/
#endif

class HELLO_API HelloServiceI : public IceBox::Service {
public:
    virtual void start(const std::string&,
                       const Ice::CommunicatorPtr&,
                       const Ice::StringSeq&);
    virtual void stop();
```

```
private:
    Ice::ObjectAdapterPtr _adapter;
};
```

First, we include the IceBox header file so that we can derive our implementation from `IceBox::Service`. Second, the preprocessor definitions are necessary because, on Windows, this service resides in a Dynamic Link Library (DLL), therefore we need to export the class so that the server can load it properly.

The member definitions are equally straightforward:

```
#include <Ice/Ice.h>
#include <HelloServiceI.h>
#include <HelloI.h>

using namespace std;

void
HelloServiceI::start(
    const string& name,
    const Ice::CommunicatorPtr& communicator,
    const Ice::StringSeq& args)
{
    _adapter = communicator->createObjectAdapter(name);
    Ice::ObjectPtr object = new HelloI(communicator);
    _adapter->add(object,
                  communicator->stringToIdentity("hello"));
    _adapter->activate();
}

void
HelloServiceI::stop()
{
    _adapter->deactivate();
}
```

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

Entry Point

The last piece of the puzzle is the *entry point* function, which the IceBox server calls to obtain an instance of the service:

```
extern "C" {
    HELLO_API IceBox::Service*
    create(Ice::CommunicatorPtr communicator)
    {
        return new HelloServiceI;
    }
}
```

In this example, the `create` function returns a new instance of the `Hello` service. The name of the function is not important, but it must have the signature shown above. In particular, the function must have C linkage, accept a single parameter of type `Ice::CommunicatorPtr`, and return a native pointer to `IceBox::Service`.

C linkage is required so that the IceBox server can locate this function in a dynamically-loaded library. The restrictions imposed on functions with C linkage prevent us from using the normal Ice calling conventions, which never return native pointers and always pass smart pointers by const reference. For example, such a function cannot return an object type (such as a smart pointer), which forces us to return a native pointer instead.

Section 43.3.4 provides more information on entry points and describes how to configure your service into an IceBox server.

43.3.2 Java Service Example

As with the C++ example presented in the previous section, the complete source for the Java example can be found in the `IceBox/hello` directory of the Ice distribution. The class definition for our service looks as follows:

```
public class HelloServiceI implements IceBox.Service
{
    public void
    start(String name,
          Ice.Communicator communicator,
          String[] args)
    {
        _adapter = communicator.createObjectAdapter(name);
        Ice.Object object = new HelloI(communicator);
        _adapter.add(object, Ice.Util.stringToIdentity("hello"));
        _adapter.activate();
    }

    public void
    stop()
```

```
    {
        _adapter.deactivate();
    }

    private Ice.ObjectAdapter _adapter;
}
```

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

The server requires a service implementation to have a default constructor. This is the *entry point* for a Java IceBox service; that is, the server dynamically loads the service implementation class and invokes the default constructor to obtain an instance of the service.

This example is a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the service implementation class, it can be configured into an IceBox server as described in Section 43.3.4.

43.3.3 C# Service Example

The complete source for the C# example can be found in the `IceBox/hello` directory of the Ice distribution. The class definition for our service looks as follows:

```
class HelloServiceI : IceBox.Service
{
    public void
    start(string name,
          Ice.Communicator communicator,
          string[] args)
    {
        _adapter = communicator.createObjectAdapter(name);
        _adapter.add(new HelloI(),
                     Ice.Util.stringToIdentity("hello"));
        _adapter.activate();
    }

    public void
    stop()
    {
        _adapter.deactivate();
```

```
    }

    private Ice.ObjectAdapter _adapter;
}
```

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

The server requires a service implementation to have a default constructor. This is the *entry point* for a C# IceBox service; that is, the server dynamically loads the service implementation class from an assembly and invokes the default constructor to obtain an instance of the service.

This example is a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the service implementation class, it can be configured into an IceBox server as described in Section 43.3.4.

43.3.4 Configuring a Service

A service is configured into an IceBox server using a single property. This property serves several purposes: it defines the name of the service, it provides the server with the service entry point, and it defines properties and arguments for the service.

The format of the property is shown below:

```
IceBox.Service.name=entry_point [args]
```

The *`name`* component of the property key is the service name (IceStorm, in this example). This name is passed to the service's `start` operation, and must be unique among all services configured in the same IceBox server. It is possible, though rarely necessary, to load two or more instances of the same service under different names.

The first argument in the property value is the entry point specification. Any arguments following the entry point specification are examined. If an argument has the form `--name=value`, then it is interpreted as a property definition that appears in the property set of the communicator passed to the service `start` operation. These arguments are removed, and any remaining arguments are passed to the `start` operation in the `args` parameter.

C++ Service

For a C++ service, the entry point must have the form *library*[,*version*]:*symbol*, where *library* is the simple name of the service's shared library or DLL, and *symbol* is the name of the entry point function. A "simple name" is one without any platform-specific prefixes or extensions; the server adds appropriate decorations depending on the platform. The version is optional. If specified, the version is embedded in the library name.

As an example, here is how we could configure IceStorm (see Chapter 44), which is implemented as an IceBox service in C++:

```
IceBox.Service.IceStorm=IceStormService,34:createIceStorm
```

IceBox uses the information provided in the entry point specification to compose a library name. For the IceStorm example shown above, IceBox on Windows would compose the library name `IceStormService34.dll`. If IceBox is compiled with debug information, it appends a `d` to the library name, so the name becomes `IceStormService34d.dll` instead.[1] The shared library or DLL must reside in a directory that appears in `PATH` on Windows or the shared library search path (such as `LD_LIBRARY_PATH`) on POSIX systems.

The entry point function, *symbol*, must have the signature that we originally presented in Section 43.3.1:

```
extern "C" IceBox::Service* function(Ice::CommunicatorPtr);
```

The communicator instance passed to this function is the IceBox server's communicator and should only be used for administrative purposes. For example, the entry point function could use this communicator's logger to display log messages. For a service's normal operations, it must use the communicator that it receives as an argument to its `start` method.

Here is sample configuration for our C++ service from Section 43.3.1:

```
IceBox.Service.Hello=HelloService:create \
    --Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in `HelloService.dll` on Windows or `libHelloService.so` on Linux, and the entry point function `create` is invoked to create an

1. The exact name of the library that is loaded depends on the naming conventions of the platform IceBox executes on. For example, on Apple machines, the library name is `libIceStormService34.dylib`.

instance of the service. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

Java Service

For a Java service, the entry point is simply the complete class name (including any package) of the service implementation class. The class must reside in the class path of the server, and must define a public default constructor.

Here is a sample configuration for our Java example from Section 43.3.2:

```
IceBox.Service.Hello=HelloServiceI \
    --Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in the class `HelloServiceI`. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

.NET Service

The entry point of a .NET service has the form *assembly*:*class*. The *assembly* component can be specified as the name of a DLL present in `PATH`, or as the full name of an assembly residing in the Global Assembly Cache (GAC), such as `hello,Version=0.0.0.0,Culture=neutral`. The *class* component is the complete class name of the service implementation class, which must define a public default constructor.

Here is a sample configuration for our C# example from Section 43.3.3:

```
IceBox.Service.Hello=helloservice.dll:HelloServiceI \
    --Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in the assembly named `helloservice.dll`, implemented by the class `HelloServiceI`. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

Sharing a Communicator

A service can be configured to use a shared communicator using the following property:

```
IceBox.UseSharedCommunicator.name=1
```

The default behavior if this property is not specified is to create a new communicator instance for the service. However, if collocation optimizations between services are desired, each of those services must be configured to use the shared communicator.

Inherited Properties

By default, a service does not inherit the server's configuration properties. For example, consider the following server configuration:

```
IceBox.Service.Weather=... --Ice.Config=svc.cfg
Ice.Trace.Network=1
```

The `Weather` service only receives the properties that are defined in its `IceBox.Service` property. In the example above, the service's communicator is initialized with the properties from the file `svc.cfg`.

If services need to inherit the server's configuration properties, define the following property in the IceBox server's configuration:

```
IceBox.InheritProperties=1
```

The properties of the shared communicator (see page 1761) are also affected by this setting.

Loading Services

By default, the server loads the configured services in an undefined order, meaning services in the same IceBox server should not depend on one another. If services must be loaded in a particular order, the `IceBox.LoadOrder` property can be used:

```
IceBox.LoadOrder=Service1,Service2
```

In this example, `Service1` is loaded first, followed by `Service2`. Any remaining services are loaded after `Service2`, in an undefined order. Each service mentioned in `IceBox.LoadOrder` must have a matching `IceBox.Service` property.

During shutdown, services are stopped in the reverse of the order in which they were loaded.

43.4 Starting IceBox

Incorporating everything we discussed in the previous sections, we can now configure and start IceBox servers.

43.4.1 Starting the C++ Server

The configuration file for our example C++ service is shown below:

```
IceBox.Service.Hello=HelloService:create
Hello.Endpoints=tcp -p 10001
```

Notice that we define an endpoint for the object adapter created by the `Hello` service.

Assuming these properties reside in a configuration file named `config`, we can start the C++ IceBox server as follows:

```
$ icebox --Ice.Config=config
```

Additional command line options are supported, including those that allow the server to run as a Windows service or Unix daemon. See Section 8.3.2 for more information.

43.4.2 Starting the Java Server

Our Java configuration is nearly identical to the C++ version, except for the entry point specification:

```
IceBox.Service.Hello=HelloServiceI
Hello.Endpoints=tcp -p 10001
```

Notice that we define an endpoint for the object adapter created by the `Hello` service.

Assuming these properties reside in a configuration file named `config`, we can start the Java IceBox server as follows:

```
$ java IceBox.Server --Ice.Config=config
```

43.4.3 Starting the C# Server

The configuration file for our example C# service is shown below:

```
IceBox.Service.Hello=helloservice.dll:HelloService
Hello.Endpoints=tcp -p 10001
```

Notice that we define an endpoint for the object adapter created by the `Hello` service.

Assuming these properties reside in a configuration file named `config`, we can start the C# IceBox server as follows:

```
$ iceboxnet --Ice.Config=config
```

43.4.4 Initialization Failure

At startup, an IceBox server inspects its configuration for all properties having the prefix `IceBox.Service.` and initializes each service. If initialization fails for a service, the IceBox server invokes the `stop` operation on any initialized services, reports an error, and terminates.

43.5 IceBox Administration

An IceBox server internally creates an object called the *service manager* that is responsible for loading and initializing the configured services. You can optionally expose this object to remote clients, such as the IceBox and IceGrid administrative utilities, so that they can execute certain administrative tasks.

43.5.1 Slice Interfaces

The Slice definitions shown below comprise the IceBox administrative interface:

```
module IceBox {
exception AlreadyStartedException {};
exception AlreadyStoppedException {};
exception NoSuchServiceException {};

interface ServiceObserver {
    ["ami"] void servicesStarted(Ice::StringSeq services);
    ["ami"] void servicesStopped(Ice::StringSeq services);
};

interface ServiceManager {
    idempotent Ice::SliceChecksumDict getSliceChecksums();
    ["ami"] void startService(string service)
        throws AlreadyStartedException, NoSuchServiceException;
    ["ami"] void stopService(string service)
        throws AlreadyStoppedException, NoSuchServiceException;
```

```
    ["ami"] void addObserver(ServiceObserver* observer)
    void shutdown();
};
};
```

ServiceManager

The `ServiceManager` interface provides access to the service manager object of an IceBox server. It defines the following operations:

- `getSliceChecksums`

  Returns a dictionary of checksums that allows a client to verify that it is using the same Slice definitions as the server (see Section 4.21).

- `startService`

  Starts a pre-configured service that is currently inactive. This operation cannot be used to add new services at run time, nor will it cause an inactive service's implementation to be reloaded. If no matching service is found, the operation raises `NoSuchServiceException`. If the service is already active, the operation raises `AlreadyStartedException`.

- `stopService`

  Stops an active service but does not unload its implementation. The operation raises `NoSuchServiceException` if no matching service is found, and `AlreadyStoppedException` if the service is stopped at the time `stopService` is invoked.

- `addObserver`

  Adds an observer that is called when IceBox services are started or stopped. The service manager ignores operations that supply a null proxy, or a proxy that has already been registered.

- `shutdown`

  Terminates the services and shuts down the IceBox server.

ServiceObserver

An administrative client that is interested in receiving callbacks when IceBox services are started or stopped must implement the `ServiceObserver` interface and register the callback object's proxy with the service manager using its `addObserver` operation. The `ServiceObserver` interface defines two operations:

- `servicesStarted`

  Invoked immediately upon registration to supply the current list of active services, and thereafter each time a service is started.

- `servicesStopped`

  Invoked whenever a service is stopped, and when the IceBox server is shutting down.

The IceBox server unregisters an observer if the invocation of either operation causes an exception.

Section 38.22.3 demonstrates how to register a `ServiceObserver` callback with an IceBox server deployed with IceGrid.

43.5.2 Enabling the Service Manager

IceBox's administrative functionality is disabled by default. You can enable it in two ways:

1. Define endpoints for the `IceBox.ServiceManager` object adapter.
2. Satisfy the prerequisites for enabling the Ice administrative facility described in Section 32.18.

For example, the following configuration property enables the `IceBox.ServiceManager` object adapter:

```
IceBox.ServiceManager.Endpoints=tcp -h 127.0.0.1 -p 10000
```

Similarly, the Ice administrative facility requires that endpoints be defined for the `Ice.Admin` object adapter with the property `Ice.Admin.Endpoints`. Note that the `Ice.Admin` object adapter is enabled automatically in an IceBox server that is deployed by IceGrid (see Section 38.22).

Regardless of which object adapter(s) you choose to enable, exposing the service manager makes an IceBox server vulnerable to denial-of-service attacks from malicious clients. Consequently, you should choose the endpoints and transports carefully; Section 32.18.8 explores these issues in greater depth.

43.5.3 Object Identities

Although an IceBox server has only one service manager object, the object is accessible via two different identities depending on how the administrative functionality was enabled (see Section 43.5.2).

The `IceBox.ServiceManager` Object Adapter

When this object adapter is enabled, the service manager object has the default identity `IceBox/ServiceManager`. If an application requires the use of multiple IceBox servers, it is a good idea to assign unique identities to their

service manager objects by configuring the servers with different values for the `IceBox.InstanceName` property, as shown in the following example:

```
IceBox.InstanceName=IceBox1
```

This property changes the category of the object's identity, which becomes `IceBox1/ServiceManager`. A corresponding change must be made in the configuration of administrative clients.

The Ice Administrative Facility

When this facility is enabled, the service manager is added as a facet of the server's `admin` object. As a result, the identity of the service manager is the same as that of the `admin` object, and the name of its facet is `IceBox.ServiceManager`. Section 32.18.1 explains that the identity of the `admin` object uses either a UUID or a statically-configured value for its category, and the value `admin` for its name. For example, consider the following property definitions:

```
Ice.Admin.Endpoints=tcp -h 127.0.0.1 -p 10001
Ice.Admin.InstanceName=IceBox
```

In this case, the identity of the `admin` object is `IceBox/admin`.

IceBox also registers a `Properties` facet (see Section 32.18.5) for each of its services so that the configuration properties of a service can be inspected remotely. The facet name is constructed as follows:

```
IceBox.Service.name.Properties
```

The value *`name`* represents the service name.

43.5.4 Client Configuration

A client requiring administrative access to the service manager can create a proxy using the endpoints configured in Section 43.5.2.

Using the `IceBox.ServiceManager` Object Adapter

To access the service manager via the `IceBox.ServiceManager` object adapter, the proxy should use the default identity `IceBox/ServiceManager` unless the server has changed the category using the `IceBox.InstanceName` property (see Section 43.5.3).

Using the Ice Administrative Facility

To access the service manager via the administrative facility, the client must first obtain (or be able to construct) a proxy for the `admin` object. As explained in Section 32.18.1, the default identity of the `admin` object uses a UUID for its category, which means the client cannot predict the identity and therefore will be unable to construct the proxy itself. If the IceBox server is deployed with IceGrid, the client can use the technique described in Section 38.22.3 to access its `admin` object.

In the absence of IceGrid, the IceBox server should set the `Ice.Admin.InstanceName` property if remote administration is required. In so doing, the identity of the `admin` object becomes well-known, and a client can construct the proxy on its own. For example, let us assume that the IceBox server defines the following property:

```
Ice.Admin.InstanceName=IceBox
```

A client can define the proxy for the `admin` object in a configuration property as follows:

```
ServiceManager.Proxy=IceBox/admin -f IceBox.ServiceManager
-h 127.0.0.1 -p 10001
```

The proxy option `-f IceBox.ServiceManager` specifies the name of the service manager's administrative facet.

43.5.5 Administrative Utility

IceBox includes C++ and Java implementations of an administrative utility. The utilities have the same usage:

```
Usage: iceboxadmin [options] [command...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.

Commands:
start SERVICE        Start a service.
stop SERVICE         Stop a service.
shutdown             Shutdown the server.
```

The C++ utility is named `iceboxadmin`, while the Java utility is represented by the class `IceBox.Admin`.

The `start` command is equivalent to invoking `startService` on the service manager interface. Its purpose is to start a pre-configured service; it cannot be

used to add new services at run time. Note that this command does not cause the service's implementation to be reloaded.

Similarly, the `stop` command stops the requested service but does not cause the IceBox server to unload the service's implementation.

The `shutdown` command stops all active services and shuts down the IceBox server.

The C++ and Java utilities obtain the service manager's proxy from the property `IceBoxAdmin.ServiceManager.Proxy`, therefore this proxy must be defined in the program's configuration file or on the command line, and the proxy's contents of depend on the server's configuration. If the IceBox server is deployed with IceGrid, we recommend using the IceGrid administrative utilities instead (see Section 38.24), which provide equivalent commands for administering an IceBox server. Otherwise, the proxy should have the endpoints configured for the server as described in Section 43.5.2 and the identity as described in Section 43.5.3.

43.6 Summary

IceBox offers a refreshing change of perspective: developers focus on writing services, not applications. The definition of an application changes as well; using IceBox, an application becomes a collection of discrete services whose composition is determined dynamically by configuration, rather than statically by the linker.

Chapter 44
IceStorm

44.1 Chapter Overview

In this chapter we present IceStorm, an efficient publish/subscribe service for Ice applications. Section 44.2 provides an introduction to IceStorm, while Section 44.3 discusses some basic IceStorm concepts. An overview of the IceStorm Slice interfaces is provided in Section 44.4, and Section 44.5 presents an example IceStorm application. An IceStorm publisher can optionally publish events to specific subscribers; this mechanism is detailed in Section 44.6. Information about IceStorm's replication facilities can be found in Section 44.7. The IceStorm administration tool is described in Section 44.8, and the subject of federation is discussed in Section 44.9. IceStorm's quality of service parameters are defined in Section 44.10, and Section 44.11 reviews the various modes of event delivery. Finally, IceStorm configuration is addressed in Section 44.12.

44.2 Introduction

Applications often need to disseminate information to multiple recipients. For example, suppose we are developing a weather monitoring application in which we collect measurements such as wind speed and temperature from a meteorolog-

ical tower and periodically distribute them to weather monitoring stations. We initially consider using the architecture shown in Figure 44.1.

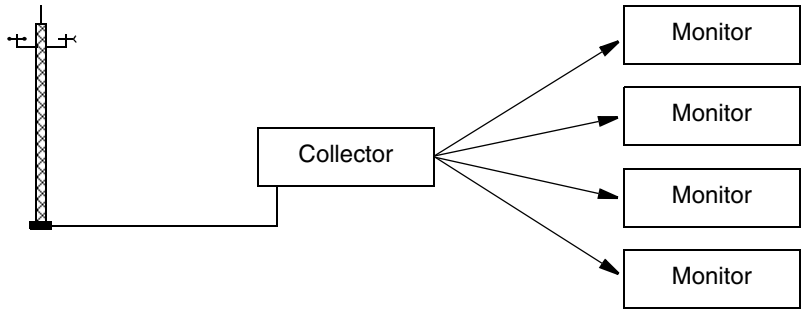


Figure 44.1. Initial design for a weather monitoring application.

However, the primary disadvantage of this architecture is that it tightly couples the collector to its monitors, needlessly complicating the collector implementation by requiring it to manage the details of monitor registration, measurement delivery, and error recovery. We can rid ourselves of these mundane duties by incorporating IceStorm into our architecture, as shown in Figure 44.2.

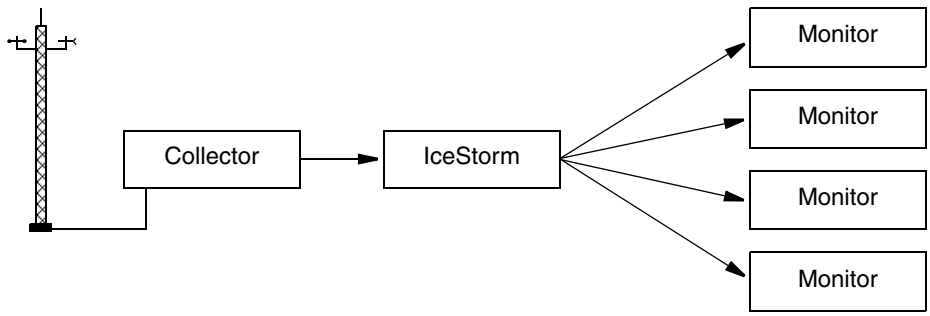


Figure 44.2. A weather monitoring application using IceStorm.

IceStorm simplifies the collector implementation significantly by decoupling it from the monitors. As a publish/subscribe service, IceStorm acts as a mediator between the collector (the publisher) and the monitors (the subscribers), and offers several advantages:

- When the collector is ready to distribute a new set of measurements, it makes a single request to the IceStorm server. The IceStorm server takes responsibility for delivering the request to the monitors, including handling any exceptions caused by ill-behaved or missing subscribers. The collector no longer needs to be aware of its monitors, or whether it even has any monitors at that moment.
- Similarly, monitors interact with the IceStorm server to perform tasks such as subscribing and unsubscribing, thereby allowing the collector to focus on its application-specific responsibilities and not on administrative trivia.
- The collector and monitor applications require very few changes to incorporate IceStorm.

44.3 Concepts

This section discusses several concepts that are important for understanding IceStorm's capabilities.

44.3.1 Message

An IceStorm *message* is strongly typed and is represented by an invocation of a Slice operation: the operation name identifies the type of the message, and the operation parameters define the message contents. A message is published by invoking the operation on an IceStorm proxy in the normal fashion. Similarly, subscribers receive the message as a regular servant upcall. As a result, IceStorm uses the "push" model for message delivery; polling is not supported.

44.3.2 Topic

An application indicates its interest in receiving messages by subscribing to a *topic*. An IceStorm server supports any number of topics, which are created dynamically and distinguished by unique names. Each topic can have multiple publishers and subscribers.

A topic is essentially equivalent to an application-defined Slice interface: the operations of the interface define the types of messages supported by the topic. A publisher uses a proxy for the topic interface to send its messages, and a subscriber implements the topic interface (or an interface derived from the topic interface) in order to receive the messages. This is no different than if the

publisher and subscriber were communicating directly in the traditional client-server style; the interface represents the contract between the client (the publisher) and the server (the subscriber), except IceStorm transparently forwards each message to multiple recipients.

IceStorm does not verify that publishers and subscribers are using compatible interfaces, therefore applications must ensure that topics are used correctly.

44.3.3 Unidirectional Messages

IceStorm messages are unidirectional, that is, they must have `void` return type, cannot have out-parameters, and cannot raise user exceptions. It follows that a publisher cannot receive replies from its subscribers. Any of the Ice transports (TCP, SSL, and UDP) can be used to publish and receive messages.

44.3.4 Federation

IceStorm supports the formation of topic graphs, also known as federation. A topic graph is formed by creating links between topics, where a *link* is a unidirectional association from one topic to another. Each link has a *cost* that may restrict message delivery on that link (see Section 44.9.2). A message published on a topic is also published on all of the topic's links for which the message cost does not exceed the link cost.

Once a message has been published on a link, the receiving topic publishes the message to its subscribers, but does not publish it on any of its links. In other words, IceStorm messages propagate at most one hop from the originating topic in a federation (see Section 44.9.1).

Figure 44.3 presents an example of topic federation. Topic T_1 has links to T_2 and T_3, as indicated by the arrows. The subscribers S_1 and S_2 receive all messages

published on T_2, as well as those published on T_1. Subscriber S_3 receives messages only from T_1, and S_4 receives messages from both T_3 and T_1.

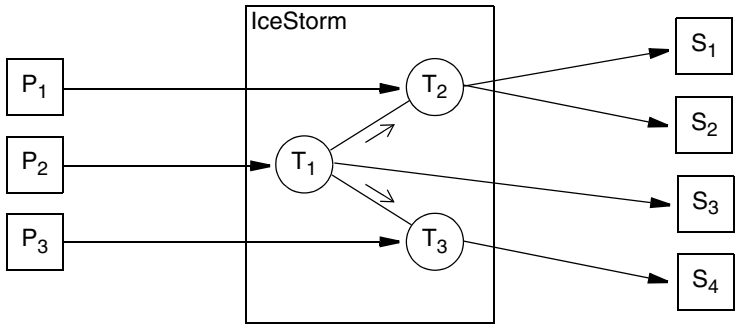


Figure 44.3. Topic federation.

IceStorm makes no attempt to prevent a subscriber from receiving duplicate messages. For example, if a subscriber is subscribed to both T_2 and T_3, then it would receive two requests for each message published on T_1.

44.3.5 Quality of Service

IceStorm allows each subscriber to specify its own *quality of service* (QoS) parameters that affect the delivery of its messages. Quality of service parameters are represented as a dictionary of name–value pairs. The supported QoS parameters are described in Section 44.10.

44.3.6 Replication

IceStorm supports replication to provide higher availability for publishers and subscribers. Refer to Section 44.7 for more information on this subject.

44.3.7 Persistent Mode

IceStorm's default behavior maintains information about topics, links, and subscribers in a database. However, a message sent via IceStorm is not stored persistently, but rather is discarded as soon as it is delivered to the topic's current set of subscribers. If an error occurs during delivery to a subscriber, IceStorm does not queue messages for that subscriber.

By default, IceStorm stores its persistent state in a Freeze database (see Chapter 39). However, you can configure IceStorm to use a different dabase, such as MySQL, among others (see Section 44.12.6).

44.3.8 Transient Mode

IceStorm can optionally run in a fully transient mode in which no database is required. Replication is not supported in this mode.

44.3.9 Subscriber Errors

If IceStorm encounters a failure while attempting to deliver a message to a subscriber, the subscriber is immediately unsubscribed from the topic on which the message was published.

Note that this is important if you make changes to a Slice data type or operation signature: if you do, you must ensure that both publishers and subscribers use the same Slice definitions; if you do not, IceStorm is likely to encounter marshaling errors when forwarding an event to a subscriber with a mismatched Slice definition and remove the subscription.

44.4 IceStorm Interface Overview

This section provides a brief introduction to the Slice interfaces comprising the IceStorm service. See the online Slice API Reference for the Slice documentation.

44.4.1 TopicManager

The `TopicManager` is a singleton object that acts as a factory and repository of `Topic` objects. Its interface and related types are shown below:

```
module IceStorm {
dictionary<string, Topic*> TopicDict;

exception TopicExists {
    string name;
};

exception NoSuchTopic {
    string name;
```

```
};

interface TopicManager {
    Topic* create(string name) throws TopicExists;
    idempotent Topic* retrieve(string name) throws NoSuchTopic;
    idempotent TopicDict retrieveAll();
    idempotent Ice::SliceChecksumDict getSliceChecksums();
};
};
```

The `create` operation is used to create a new topic, which must have a unique name. The `retrieve` operation allows a client to obtain a proxy for an existing topic, and `retrieveAll` supplies a dictionary of all existing topics. The `getSliceChecksums` operation returns Slice checksums for the IceStorm definitions (see Section 4.21 for more information).

44.4.2 Topic

The `Topic` interface represents a topic and provides several administrative operations for configuring links and managing subscribers.

```
module IceStorm {
struct LinkInfo {
    Topic* theTopic;
    string name;
    int cost;
};
sequence<LinkInfo> LinkInfoSeq;

dictionary<string, string> QoS;

exception LinkExists {
    string name;
};

exception NoSuchLink {
    string name;
};

exception AlreadySubscribed {};

exception BadQoS {
    string reason;
};
```

```
interface Topic {
    idempotent string getName();
    idempotent Object* getPublisher();
    idempotent Object* getNonReplicatedPublisher();
    Object* subscribeAndGetPublisher(
                QoS theQoS, Object* subscriber)
        throws AlreadySubscribed, BadQoS;
    idempotent void unsubscribe(Object* subscriber);
    idempotent void link(Topic* linkTo, int cost)
        throws LinkExists;
    idempotent void unlink(Topic* linkTo) throws NoSuchLink;
    idempotent LinkInfoSeq getLinkInfoSeq();
    void destroy();
};
};
```

The `getName` operation returns the name assigned to the topic, while the `getPublisher` and `getNonReplicatedPublisher` operations return proxies for the topic's publisher object (see Section 44.5.2).

The `subscribeAndGetPublisher` operation adds a subscriber's proxy to the topic; if another subscriber proxy already exists with the same object identity, the operation throws `AlreadySubscribed`. The operation returns the publisher for the topic (see Section 44.6).

The `unsubscribe` operation removes the subscriber from the topic.

A link to another topic is created using the `link` operation; if a link already exists to the given topic, the `LinkExists` exception is raised. Links are destroyed using the `unlink` operation.

Finally, the `destroy` operation permanently destroys the topic.

44.5 Using IceStorm

In this section we expand on the weather monitoring example from Section 44.2, demonstrating how to create, subscribe to and publish messages on a topic. We use the following Slice definitions in our example:

```
struct Measurement {
    string tower; // tower id
    float windSpeed; // knots
    short windDirection; // degrees
    float temperature; // degrees Celsius
```

```
};

interface Monitor {
    void report(Measurement m);
};
```

`Monitor` is our topic interface. For the sake of simplicity, it defines just one operation, `report`, taking a `Measurement` struct as its only parameter.

44.5.1 Implementing a Publisher

The implementation of our collector application can be summarized easily:

1. Obtain a proxy for the `TopicManager`. This is the primary IceStorm object, used by both publishers and subscribers.
2. Obtain a proxy for the `Weather` topic, either by creating the topic if it does not exist, or retrieving the proxy for the existing topic.
3. Obtain a proxy for the `Weather` topic's "publisher object." This proxy is provided for the purpose of publishing messages, and therefore is narrowed to the topic interface (`Monitor`).
4. Collect and report measurements.

In the sections below, we present collector implementations in C++ and Java.

C++ Example

As usual, our C++ example begins by including the necessary header files. The interesting ones are `IceStorm/IceStorm.h`, which is generated from the IceStorm Slice definitions, and `Monitor.h`, containing the generated code for our monitor definitions shown above.

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

int main(int argc, char* argv[])
{
    ...
    Ice::ObjectPrx obj = communicator->stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm::TopicManagerPrx topicManager =
        IceStorm::TopicManagerPrx::checkedCast(obj);
    IceStorm::TopicPrx topic;
    while (!topic) {
```

```
        try {
            topic = topicManager->retrieve("Weather");
        } catch (const IceStorm::NoSuchTopic&) {
            try {
                topic = topicManager->create("Weather");
            } catch (const IceStorm::TopicExists&) {
                // Another client created the topic.
            }
        }
    }

    Ice::ObjectPrx pub = topic->getPublisher()->ice_oneway();
    MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
    while (true) {
        Measurement m = getMeasurement();
        monitor->report(m);
    }
    ...
}
```

Note that this example assumes that IceStorm uses the instance name IceStorm. The actual instance name may differ, and you need to use it as the category when calling `stringToProxy` (see page 1928).

After obtaining a proxy for the topic manager, the collector attempts to retrieve the topic. If the topic does not exist yet, the collector receives a `NoSuchTopic` exception and then creates the topic.

```
IceStorm::TopicPrx topic;
while(!topic) {
    try {
        topic = topicManager->retrieve("Weather");
    } catch (const IceStorm::NoSuchTopic&) {
        try {
            topic = topicManager->create("Weather");
        } catch (const IceStorm::TopicExists&) {
            // Another client created the topic.
        }
    }
}
```

The next step is obtaining a proxy for the publisher object, which the collector narrows to the `Monitor` interface. (We create a oneway proxy for the publisher purely for efficiency reasons.)

```
Ice::ObjectPrx pub = topic->getPublisher()->ice_oneway();
MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
```

Finally, the collector enters its main loop, collecting measurements and publishing them via the IceStorm publisher object.

```
while (true) {
    Measurement m = getMeasurement();
    monitor->report(m);
}
```

Java Example

The equivalent Java version is shown below.

```
public static void main(String[] args)
{
    ...
    Ice.ObjectPrx obj = communicator.stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm.TopicManagerPrx topicManager =
        IceStorm.TopicManagerPrxHelper.checkedCast(obj);
    IceStorm.TopicPrx topic = null;
    while (topic == null) {
        try {
            topic = topicManager.retrieve("Weather");
        } catch (IceStorm.NoSuchTopic ex) {
            try {
                topic = topicManager.create("Weather");
            } catch (IceStorm.TopicExists ex) {
                // Another client created the topic.
            }
        }
    }

    Ice.ObjectPrx pub = topic.getPublisher().ice_oneway();
    MonitorPrx monitor = MonitorPrxHelper.uncheckedCast(pub);
    while (true) {
        Measurement m = getMeasurement();
        monitor.report(m);
    }
    ...
}
```

Note that this example assumes that IceStorm uses the instance name IceStorm. The actual instance name may differ, and you need to use it as the category when calling `stringToProxy` (see page 1928).

After obtaining a proxy for the topic manager, the collector attempts to retrieve the topic. If the topic does not exist yet, the collector receives a `NoSuchTopic` exception and then creates the topic.

```
IceStorm.TopicPrx topic = null;
while (topic == null) {
    try {
        topic = topicManager.retrieve("Weather");
    } catch (IceStorm.NoSuchTopic ex) {
        try {
            topic = topicManager.create("Weather");
        } catch (IceStorm.TopicExists ex) {
            // Another client created the topic.
        }
    }
}
```

The next step is obtaining a proxy for the publisher object, which the collector narrows to the `Monitor` interface.

```
Ice.ObjectPrx pub = topic.getPublisher().ice_oneway();
MonitorPrx monitor = MonitorPrxHelper.uncheckedCast(pub);
```

Finally, the collector enters its main loop, collecting measurements and publishing them via the IceStorm publisher object.

```
while (true) {
    Measurement m = getMeasurement();
    monitor.report(m);
}
```

44.5.2 Using a Publisher Object

Each topic creates a publisher object for the express purpose of publishing messages. It is a special object in that it implements an Ice interface that allows the object to receive and forward requests (i.e., IceStorm messages) without requiring knowledge of the operation types.

Type Safety

From the publisher's perspective, the publisher object appears to be an application-specific type. In reality, the publisher object can forward requests for any type, and that introduces a degree of risk: a misbehaving publisher can use `uncheckedCast` to narrow the publisher object to any type and invoke any

operation; the publisher object unknowingly forwards those requests to the subscribers.

If a publisher sends a request using an incorrect type, the Ice run time in a subscriber typically responds by raising `OperationNotExistException`. However, since the subscriber receives its messages as oneway invocations, no response can be sent to the publisher object to indicate this failure, and therefore neither the publisher nor the subscriber is aware of the type-mismatch problem. In short, IceStorm places the burden on the developer to ensure that publishers and subscribers are using it correctly.

Oneway or Twoway?

IceStorm messages are unidirectional, but publishers may use either oneway or twoway invocations when sending messages to the publisher object. Each invocation style has advantages and disadvantages that you should consider when deciding which one to use. The differences between the invocation styles affect a publisher in four ways:

- Efficiency

  Oneway invocations have the advantage in efficiency because the Ice run time in the publisher does not await a reply to each message (and, of course, no reply is sent by IceStorm on the wire).

- Ordering

  The use of oneway invocations by a publisher may affect the order in which subscribers receive messages. If ordering is important, use twoway invocations with a reliability QoS of `ordered`, or use a single thread in the subscriber (see also Section 44.10.1).

- Reliability

  Oneway invocations can be lost under certain circumstances, even when they are sent over a reliable transport such as TCP (see Section 32.14). If the loss of messages is unacceptable, or you are unable to address the potential causes of lost oneway messages, then twoway invocations are recommended.

- Delays

  A publisher may experience network-related delays when sending messages to IceStorm if subscribers are slow in processing messages. Twoway invocations are more susceptible to these delays than oneway invocations.

Transports

Each publisher can select its own transport for message delivery, therefore the transport used by a publisher to communicate with IceStorm has no effect on how IceStorm delivers messages to its subscribers.

For example, a publisher can use a UDP transport if the possibility of lost messages is acceptable (and if IceStorm provides a UDP endpoint to publishers). However, the TCP or SSL transports are generally recommended for IceStorm's publisher endpoint in order to ensure that published messages are delivered reliably to IceStorm, even if they may not be delivered reliably to some subscribers.

Request Contexts

A request context is an optional argument of all remote invocations (see Section 32.12). If a publisher supplies a request context when publishing a message, IceStorm will forward it intact to subscribers.

Services such as Glacier2 employ request contexts to provide applications with more control over the service's behavior. For example, if a publisher knows that IceStorm is delivering messages to subscribers via a Glacier2 router, the publisher can influence Glacier2's behavior by including a request context, as shown in the following C++ example:

```
Ice::ObjectPrx pub = topic->getPublisher();
Ice::Context ctx;
ctx["_fwd"] = "Oz";
MonitorPrx monitor =
    MonitorPrx::uncheckedCast(pub->ice_context(ctx));
```

The `_fwd` context key, when encountered by Glacier2, causes the router to forward the request using compressed batch oneway messages. The `ice_context` method is used to obtain a proxy that includes the Glacier2 request context in every invocation, eliminating the need for the publisher to specify it explicitly. See Section 42.10 for more information on Glacier2's use of request contexts.

44.5.3 Implementing a Subscriber

Our subscriber implementation takes the following steps:

1. Obtain a proxy for the `TopicManager`. This is the primary IceStorm object, used by both publishers and subscribers.
2. Create an object adapter to host our `Monitor` servant.

3. Instantiate the `Monitor` servant and activate it with the object adapter.
4. Subscribe to the `Weather` topic.
5. Process `report` messages until shutdown.
6. Unsubscribe from the `Weather` topic.

In the sections below, we present monitor implementations in C++ and Java.

C++ Example

Our C++ monitor implementation begins by including the necessary header files. The interesting ones are `IceStorm/IceStorm.h`, which is generated from the IceStorm Slice definitions, and `Monitor.h`, containing the generated code for our monitor definitions shown at the beginning of Section 44.2.

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

using namespace std;

class MonitorI : virtual public Monitor {
public:
    virtual void report(const Measurement& m,
                        const Ice::Current&) {
        cout << "Measurement report:" << endl
             << "  Tower: " << m.tower << endl
             << "  W Spd: " << m.windSpeed << endl
             << "  W Dir: " << m.windDirection << endl
             << "   Temp: " << m.temperature << endl
             << endl;
    }
};

int main(int argc, char* argv[])
{
    ...
    Ice::ObjectPrx obj = communicator->stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm::TopicManagerPrx topicManager =
        IceStorm::TopicManagerPrx::checkedCast(obj);

    Ice::ObjectAdapterPtr adapter =
        communicator->createObjectAdapter("MonitorAdapter");

    MonitorPtr monitor = new MonitorI;
```

```
    Ice::ObjectPrx proxy = adapter->
                addWithUUID(monitor)->ice_oneway();

    IceStorm::TopicPrx topic;
    try {
        topic = topicManager->retrieve("Weather");
        IceStorm::QoS qos;
        topic->subscribeAndGetPublisher(qos, proxy);
    }
    catch (const IceStorm::NoSuchTopic&) {
        // Error! No topic found!
        ...
    }

    adapter->activate();
    communicator->waitForShutdown();

    topic->unsubscribe(proxy);
    ...
}
```

Our implementation of the `Monitor` servant is currently quite simple. A real implementation might update a graphical display, or incorporate the measurements into an ongoing calculation.

```
class MonitorI : virtual public Monitor {
public:
    virtual void report(const Measurement& m,
                        const Ice::Current&) {
        cout << "Measurement report:" << endl
             << "  Tower: " << m.tower << endl
             << "  W Spd: " << m.windSpeed << endl
             << "  W Dir: " << m.windDirection << endl
             << "   Temp: " << m.temperature << endl
             << endl;
    }
};
```

After obtaining a proxy for the topic manager, the program creates an object adapter, instantiates the `Monitor` servant and activates it.

```
    Ice::ObjectAdapterPtr adapter =
        communicator->createObjectAdapter("MonitorAdapter");

    MonitorPtr monitor = new MonitorI;
    Ice::ObjectPrx proxy =
        adapter->addWithUUID(monitor)->ice_oneway();
```

Note that the code creates a oneway proxy for the `Monitor` servant. This is for efficiency reasons: by subscribing with a oneway proxy, IceStorm will deliver events to the subscriber via oneway messages, instead of via twoway messages.

Next, the monitor subscribes to the topic.

```
IceStorm::TopicPrx topic;
try {
    topic = topicManager->retrieve("Weather");
    IceStorm::QoS qos;
    topic->subscribeAndGetPublisher(qos, proxy);
}
catch (const IceStorm::NoSuchTopic&) {
    // Error! No topic found!
    ...
}
```

Finally, the monitor activates its object adapter and waits to be shutdown. After `waitForShutdown` returns, the monitor cleans up by unsubscribing from the topic.

```
adapter->activate();
communicator->waitForShutdown();

topic->unsubscribe(proxy);
```

Java Example

The Java implementation of the monitor is shown below.

```
class MonitorI extends _MonitorDisp {
    public void report(Measurement m, Ice.Current curr) {
        System.out.println(
            "Measurement report:\n" +
            "  Tower: " + m.tower + "\n" +
            "  W Spd: " + m.windSpeed + "\n" +
            "  W Dir: " + m.windDirection + "\n" +
            "   Temp: " + m.temperature + "\n");
    }
}

public static void main(String[] args)
{
    ...
    Ice.ObjectPrx obj = communicator.stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm.TopicManagerPrx topicManager =
        IceStorm.TopicManagerPrxHelper.checkedCast(obj);
```

```
    Ice.ObjectAdapterPtr adapter =
        communicator.createObjectAdapter("MonitorAdapter");

    Monitor monitor = new MonitorI();
    Ice.ObjectPrx proxy =
        adapter.addWithUUID(monitor).ice_oneway();

    IceStorm.TopicPrx topic = null;
    try {
        topic = topicManager.retrieve("Weather");
        java.util.Map qos = null;
        topic.subscribeAndGetPublisher(qos, proxy);
    }
    catch (IceStorm.NoSuchTopic ex) {
        // Error! No topic found!
        ...
    }

    adapter.activate();
    communicator.waitForShutdown();

    topic.unsubscribe(proxy);
    ...
}
```

Our implementation of the `Monitor` servant is currently quite simple. A real implementation might update a graphical display, or incorporate the measurements into an ongoing calculation.

```
class MonitorI extends _MonitorDisp {
    public void report(Measurement m, Ice.Current curr) {
        System.out.println(
            "Measurement report:\n" +
            "  Tower: " + m.tower + "\n" +
            "  W Spd: " + m.windSpeed + "\n" +
            "  W Dir: " + m.windDirection + "\n" +
            "   Temp: " + m.temperature + "\n");
    }
}
```

After obtaining a proxy for the topic manager, the program creates an object adapter, instantiates the `Monitor` servant and activates it.

```
    Monitor monitor = new MonitorI();
    Ice.ObjectPrx proxy =
        adapter.addWithUUID(monitor).ice_oneway();
```

Note that the code creates a oneway proxy for the `Monitor` servant. This is for efficiency reasons: by subscribing with a oneway proxy, IceStorm will deliver events to the subscriber via oneway messages, instead of via twoway messages.

Next, the monitor subscribes to the topic.

```
IceStorm.TopicPrx topic = null;
try {
    topic = topicManager.retrieve("Weather");
    java.util.Map qos = null;
    topic.subscribeAndGetPublisher(qos, proxy);
}
catch (IceStorm.NoSuchTopic ex) {
    // Error! No topic found!
    ...
}
```

Finally, the monitor activates its object adapter and waits to be shutdown. After `waitForShutdown` returns, the monitor cleans up by unsubscribing from the topic.

```
adapter.activate();
communicator.waitForShutdown();

topic.unsubscribe(proxy);
```

44.6 Publishing to a Specific Subscriber

If you send events to the publisher object you obtain by calling `Topic::getPublisher`, the event is forwarded to all subscribers for that topic:

```
IceStorm::TopicPrx topic = ...;
Ice::ObjectPrx pub = topic->getPublisher()->ice_oneway();

MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
Measurement m = ...;

monitor->report(m); // Sent to all subscribers
```

You can also publish an event to a single specific subscriber, by using the return value of `subscribeAndGetPublisher`. For example:

```
MonitorPtr monitor = new MonitorI;
Ice::ObjectPrx proxy = adapter->
                    addWithUUID(monitor)->ice_oneway();
```

```
IceStorm::topicPrx topic = ...;

Icestorm::QoS qos;
Ice::ObjectPrx pub = topic->subscribeAndGetPublisher(qos, proxy);
MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);

Measurement m = ...;
monitor->report(m); // Sent to only to this subscriber
```

Note that, here, we save the return value of `subscribeAndGetPublisher`. The return value is a proxy that connects specifically to the `MonitorI` instance denoted by `proxy`. However, when the code calls `report` on that proxy, instead of directly invoking on the `MonitorI` instance, the request is forwarded via IceStorm.

As it stands, this code is not very interesting. After all, the call to `monitor->report` is just a round-about way for the subscriber to publish a message to itself. However, the subscriber can pass this subscriber-specific publisher proxy to another process. When that process publishes an event via the proxy, the event is sent only to the specific subscriber, instead of to all subscribers for the topic. In turn, this is useful if you are using the observer pattern, with all observers attached to an IceStorm topic.

As an example, we might have a list whose state is to be monitored by a number of observers. Updates to the list are published to an IceStorm topic, say, `ListUpdates`. The observers of the list subscribe with an interface such as:

```
interface ListObserver {
    void init(/* The entire state of the list */);
    void itemChange(/* The added or deleted item */);
};
```

The idea is that, when an observer first starts observing the list, the `init` operation is called on the observer and passed the entire list. This initializes the observer with the current state of the list. Thereafter, whenever the list changes, it calls `itemChange` on the observer to inform it of the addition or deletion of an item. (The details of how this happens are secondary; the important point is that the observer is informed of the current state of the list initially and, thereafter, receives incremental updates about modifications to the list, rather than the entire list whenever it changes.)

The list itself might look something like this:

```
interface List {
    void add(Item i);
    void remove(Item i);

    void addObserver(ListObserver* lo);
    void removeObserver(ListObserver* lo);
};
```

The list provides operations to add and remove an item, as well as operations to add and remove an observer. Every time `add` or `remove` are called on the list, the list publishes an `itemChange` event to the `ListUpdates` topic; this informs all the subscribed observers of the change to the list. However, when an observer is first added, the observer's `init` operation must be called. Moreover, we want to call that method only once for each observer, so we cannot just publish the initial state of the list on a topic that all observers subscribe to.

The subscriber-specific proxy that is returned by `subscribeAndGetPublisher` solves this nicely: the implementation of `addObserver` calls `subscribeAndGetPublisher`, and then invokes `init` on the observer. This both subscribes the observer to the topic, and IceStorm forwards the call to `init` to the observer. This is preferable to the list invoking `init` on the observer directly: if the observer is misbehaved (for example, if its `init` implementation blocks for some time), the list is unaffected because IceStorm shields the list from such behavior.

44.7 Highly Available IceStorm

IceStorm offers a highly available (HA) mode that employs master-slave replication with automatic failover in case the master fails.

44.7.1 Algorithm

HA IceStorm uses the Garcia-Molina "Invitation Election Algorithm" as described in [28], in which each replica has a priority and belongs to a replica group. The replica with the highest priority in the group becomes the coordinator, and the remaining replicas are slaves of the coordinator.

All replicas are statically configured with information about all other replicas, including their priority. The group combining works as follows:

- When recovering from an error, or during startup, replicas form a single self-coordinated group.

- Coordinators periodically attempt to combine their groups with other groups in order to form larger groups.

At regular intervals, slave replicas contact their coordinator to ensure that the coordinator is still the master of the slave's group. If a failure occurs, the replica considers itself in error and performs error recovery as described above.

Replication commences once a group contains a majority of replicas. A majority is necessary to avoid the possibility of network partitioning, in which two groups of replicas form that cannot communicate and whose database contents diverge. With respect to IceStorm, a consequence of requiring a majority is that a minimum of three replicas are necessary.

An exception to the majority rule is made during full system startup (i.e., when no replica is currently running). In this situation, replication can only commence with the participation of every replica in the group. This requirement guarantees that the databases of all replicas are synchronized, and avoids the risk that the database of an offline replica might contain more recent information.

Once a majority group has been formed, all database states are compared. The most recent database state (as determined by comparing a time stamp recorded upon each database change) is transferred to all replicas and replication commences. IceStorm is now available for use.

44.7.2 Replica State

IceStorm replicas can have one of four states:

- Inactive

  The node is inactive and awaiting an election.

- Election

  The node is electing a coordinator.

- Reorganization

  The replica group is reorganizing.

- Normal

  The replica group is active and replicating.

For debugging purposes, you can obtain the state of the replicas using the **`replica`** command, as shown below:

```
$ icestormadmin --Ice.Config=config
>>> replica
replica count: 3
1: id:         1
1: coord:      3
1: group name: 3:191131CC-703A-41D6-8B80-D19F0D5F0410
1: state:      normal
1: group:
1: max:        3
2: id:         2
2: coord:      3
2: group name: 3:191131CC-703A-41D6-8B80-D19F0D5F0410
2: state:      normal
2: group:
2: max:        3
3: id:         3
3: coord:      3
3: group name: 3:191131CC-703A-41D6-8B80-D19F0D5F0410
3: state:      normal
3: group:      1,2
3: max:        3
```

Each line begins with the identifier of the replica. The command displays the following information:

- `id`

  The identifier of the replica.

- `coord`

  The identifier of the group's coordinator.

- `group name`

  The name of the group to which this replica belongs.

- `state`

  The replica's current state.

- `group`

  The identifiers of the other replicas in the group. Note that only the coordinator knows, or cares about, this information.

- `max`

  The maximum number of replicas seen by this replica. This value is used during startup to determine whether full participation is necessary. If the value is less than the total number of replicas, full participation is required.

See Section 44.8 for more information on the **icestormadmin** utility.

44.7.3 IceStorm Clients

As previously noted, an individual IceStorm replica can be in one of several states. However, IceStorm clients have a different perspective in which the replication group as a whole is in one of the states shown below:

- Down

  All requests to IceStorm fail.

- Inactive

  All requests to IceStorm block until the replica is either down (in which case the request fails), or becomes Active.

- Active

  Requests are processed.

It is also possible, but highly unlikely, for a request to result in an `Ice::UnknownException`. This can happen, for example, if a replica loses the majority and thus progresses to the inactive state during request processing. In this case, the result of the request is indeterminate (the request may or may not have succeeded) and therefore the IceStorm client can draw no conclusion. The client should retry the request and be prepared for the request to fail. Consider this example:

```
// C++
TopicPrx topic = ...;
Ice::ObjectPrx sub = ...;
IceStorm::QoS qos;
topic->subscribeAndGetPublisher(qos, sub);
```

The call to `subscribeAndGetPublisher` may fail in very rare cases with an `UnknownException`, indicating that the subscription may or may not have succeeded. Here is the proper way to deal with the possibility of an `UnknownException`:

```
// C++
TopicPrx topic = ...;
Ice::ObjectPrx sub = ...;
IceStorm::QoS qos;
while(true) {
    try {
        topic->subscriberAndGetPublisher(qos, sub);
    } catch(const Ice::UnknownException&) {
        continue;
```

```
        } catch(const IceStorm::AlreadySubscribed&) {
            // Expected.
        }
        break;
    }
```

44.7.4 Subscribers

Subscribers can receive events from any replica. The subscriber will stop receiving events under two circumstances:

- The subscriber is unsubscribed by calling `Topic::unsubscribe`.
- The subscriber is removed as a result of a failure to deliver events. See Section 44.10.2 for more details.

44.7.5 Publishers

A publisher for HA IceStorm typically receives a proxy containing multiple endpoints. With this proxy, the publisher normally binds to a single replica and continues using that replica unless there is a failure, or until active connection management (ACM) closes the connection.

As with non-HA IceStorm, event delivery ordering can be guaranteed if the subscriber and publisher are suitably configured (see Section 44.11) and the publisher continues to use the same replica when publishing events.

Ordering guarantees are lost as soon as a publisher changes to a different replica. Furthermore, a publisher may receive no notification that a change has occurred, which is possible under two circumstances:

- ACM has closed the connection (see Section 36.4).
- Publishing to a replica fails and the Ice invocation can be retried, in which case the Ice run time in the publisher automatically and transparently attempts to send the request to another replica. The publisher receives an exception if the invocation cannot be retried.

A publisher has two ways of ensuring that it is notified about a change in replicas:

- The simplest method is to use the `Topic::getNonReplicatedPublisher` operation. The proxy returned by this operation points directly at the current replica and no transparent failover to a different replica can occur.
- If you never want transparent failover to occur during publishing, you can configure your publisher proxy so that it contains only one endpoint (see

Section 44.12.3). In this configuration, the `Topic::getPublisher` operation behaves exactly like `getNonReplicatedPublisher`.

Of the two strategies, using `getNonReplicatedPublisher` is preferable for two reasons:

- It does not involve changes to IceStorm's configuration.
- It is still possible to obtain a replicated publisher proxy by calling `getPublisher`, whereas if you had used the second strategy you would have eliminated that possibility.

The second strategy may be necessary in certain circumstances, such as when an existing IceStorm application is deployed and cannot be changed.

Regardless of the strategy you choose, a publisher can recover from the failure of a replica by requesting another proxy from the replicated topic using `getPublisher` or `getNonReplicatedPublisher`.

44.8 IceStorm Administration

The IceStorm administration tool is a command-line program that provides administrative control of an IceStorm server. The tool requires that the `IceStormAdmin.TopicManager.Default` property be specified as described in Section 44.12.4.

The following command-line options are supported:

```
$ icestormadmin -h
Usage: icestormadmin [options] [file...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
-e COMMANDS          Execute COMMANDS.
-d, --debug          Print debug messages.
```

If one or more `-e` options are specified, the tool executes the given commands and exits, otherwise the tool enters an interactive session. The **`help`** command displays the following usage information:

`help`

Print this message.

`exit`, **`quit`**

Exit this program.

create *TOPICS*

Add ***TOPICS***.

destroy *TOPICS*

Remove ***TOPICS***.

link *FROM TO* [*COST*]

Link ***FROM*** to ***TO*** with the optional ***COST***.

unlink *FROM TO*

Unlink ***TO*** from ***FROM***.

links [*INSTANCE-NAME*]

Without an argument, ***links*** displays the links of all topics in the current topic manager. You can specify a different topic manager by providing its instance name.

topics [*INSTANCE-NAME*]

Without an argument, ***topics*** displays the names of all topics in the current topic manager. You can specify a different topic manager by providing its instance name.

current [*INSTANCE-NAME*]

Set the current topic manager to the topic manager with instance name ***INSTANCE-NAME***. The proxy of the corresponding topic manager must be specified by setting an `IceStormAdmin.TopicManager.`*`name`* property. Without an argument, the command shows the current topic manager.

replica [*INSTANCE-NAME*]

Display replication information for the given ***INSTANCE-NAME***. See Section 44.7.2 for more details on this command.

Some of the commands accept one or more topic names (***TOPICS***) as arguments. Topic names containing white space or matching a command keyword must be enclosed in single or double quotes.

By default, **icestormadmin** uses the topic manager specified by the setting of the `IceStormAdmin.TopicManager.Default` property, which specifies the proxy for the topic manager. For example, without additional arguments, the **create** command operates on that topic manager.

If you are using multiple topic managers, you can specify the proxies by setting the property `IceStormAdmin.TopicManager.`*`name`* for each topic manager. For example:

```
IceStormAdmin.TopicManager.A=A/TopicManager:tcp -h x -p 9995
IceStormAdmin.TopicManager.B=Foo/TopicManager:tcp -h x -p 9996
IceStormAdmin.TopicManager.C=Bar/TopicManager:tcp -h z -p 9995
```

This sets the proxies for three topic managers. Note that *`name`* need not match the instance name of the corresponding topic manager—*`name`* simply serves as a tag. With these property settings, the **`icestormadmin`** commands that accept a topic can now specify a topic manager other than the default topic manager that is configured with `IceStormAdmin.TopicManager.Default`. For example:

```
current Foo
create myTopic
create Bar/myOtherTopic
```

This sets the current topic manager to the one with instance name `Foo`; the first **`create`** command then creates the topic within that topic manager, whereas the second **`create`** command uses the topic manager with instance name `Bar`.

44.9 Topic Federation

The ability to link topics together into a federation provides IceStorm applications with a lot of flexibility, while the notion of a "cost" associated with links allows applications to restrict the flow of messages in creative ways. IceStorm applications have complete control of topic federation using the `TopicManager` interface described in the online Slice Reference, allowing links to be created and removed dynamically as necessary. For many applications, however, the topic graph is static and therefore can be configured using the administrative tool discussed in Section 44.8.

44.9.1 Message Propagation

IceStorm messages are never propagated over more than one link. For example, consider the topic graph shown in Figure 44.4.

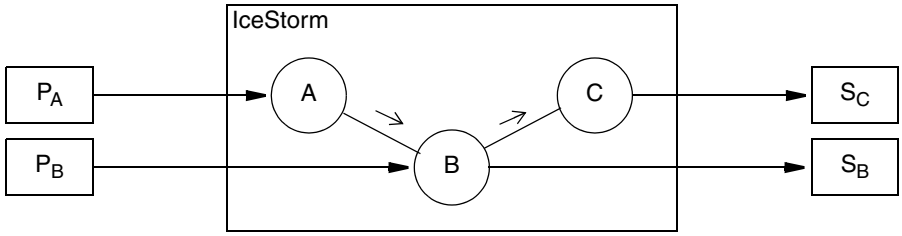


Figure 44.4. Message propagation.

In this case, messages published on `A` are propagated to `B`, but `B` does not propagate `A`'s messages to `C`. Therefore, subscriber S_B receives messages published on topics `A` and `B`, but subscriber S_C only receives messages published on topics `B` and `C`. If the application needs messages to propagate from `A` to `C`, then a link must be established directly between `A` and `C`.

44.9.2 Cost

As described in Section 44.9.1, IceStorm messages are only propagated on the originating topic's immediate links. In addition, applications can use the notion of cost to further restrict message propagation.

A cost is associated with messages and links. When a message is published on a topic, the topic compares the cost associated with each of its links against the message cost, and only propagates the message on those links whose cost equals or exceeds the message cost. A cost value of zero (`0`) has the following implications:

- messages with a cost value of zero (`0`) are published on all of the topic's links regardless of the link cost;
- links with a cost value of zero (`0`) accept all messages regardless of the message cost.

For example, consider the topic graph shown in Figure 44.5.

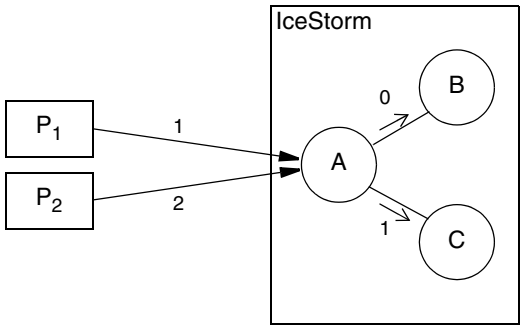


Figure 44.5. Cost semantics.

Publisher P_1 publishes a message on topic `A` with a cost of `1`. This message is propagated on the link to topic `B` because the link has a cost of `0` and therefore accepts all messages. The message is also propagated on the link to topic `C`, because the message cost does not exceed the link cost (`1`). On the other hand, the message published by P_2 with a cost of `2` is only propagated on the link to `B`.

Request Context

The cost of a message is specified in an Ice request context. Each Ice proxy operation has an implicit argument of type `Ice::Context` representing the request context (see Section 32.12). This argument is rarely used, but it is the ideal location for specifying the cost of an IceStorm message because an application only needs to supply a request context if it actually uses IceStorm's cost feature. If the request context does not contain a cost value, the message is assigned the default cost value of zero (`0`).

Publishing a Message with a Cost

The code examples below demonstrate how a collector can publish a measurement with a cost value of `5`. First, the C++ version:

```
Measurement m = getMeasurement();
Ice::Context ctx;
ctx["cost"] = "5";
monitor->report(m, ctx);
```

And here is the equivalent version in Java:

```
Measurement m = getMeasurement();
java.util.HashMap ctx = new java.util.HashMap();
ctx.put("cost", "5");
monitor.report(m, ctx);
```

Receiving a Message with a Cost

A subscriber can discover the cost of a message by examining the request context supplied in the `Ice::Current` argument. For example, here is a C++ implementation of `Monitor::report` that displays the cost value if it is present:

```
virtual void report(const Measurement& m,
                    const Ice::Current& curr) {
    Ice::Context::const_iterator p = curr.ctx.find("cost");
    cout << "Measurement report:" << endl
         << "  Tower: " << m.tower << endl
         << "  W Spd: " << m.windSpeed << endl
         << "  W Dir: " << m.windDirection << endl
         << "   Temp: " << m.temperature << endl
         << "   Temp: " << m.temperature << endl;
    if (p != curr.ctx.end())
        cout << "   Cost: " << p->second << endl;
    cout << endl;
}
```

And here is the equivalent Java implementation:

```
public void report(Measurement m, Ice.Current curr) {
    String cost = null;
    if (curr.ctx != null)
        cost = curr.ctx.get("cost");
    System.out.println(
        "Measurement report:\n" +
        "  Tower: " + m.tower + "\n" +
        "  W Spd: " + m.windSpeed + "\n" +
        "  W Dir: " + m.windDirection + "\n" +
        "   Temp: " + m.temperature);
    if (cost != null)
        System.out.println("   Cost: " + cost);
    System.out.println();
}
```

For the sake of efficiency, the Ice for Java run time may supply a null value for the request context in `Ice.Current`, therefore an application is required to check for null before using the request context.

44.9.3 Automating Federation

Given the restrictions on message propagation described in the previous sections, creating a complex topic graph can be a tedious endeavor. Of course, creating a topic graph is not typically a common occurrence, since IceStorm keeps a persistent record of the graph. However, there are situations where an automated procedure for creating a topic graph can be valuable, such as during development when the graph might change significantly and often, or when graphs need to be recomputed based on changing costs.

Administration Tool Script

A simple way to automate the creation of a topic graph is to create a text file containing commands to be executed by the IceStorm administration tool. For example, the commands to create the topic graph shown in Figure 44.5 are shown below:

```
create A B C
link A B 0
link A C 1
```

If we store these commands in the file `graph.txt`, we can execute them using the following command:

```
$ icestormadmin --Ice.Config=config graph.txt
```

We assume that the configuration file `config` contains the definition for the property `IceStormAdmin.TopicManager.Default`.

44.9.4 Proxies for Federation

Note that, if you federate IceStorm servers, you must ensure that the proxies for the linked topics always use the same host and port (or, alternatively, can be indirectly bound via IceGrid), otherwise the federation cannot be re-established if one of the servers in the federation shuts down and is restarted later.

44.10 Quality of Service

An IceStorm subscriber specifies Quality of Service (QoS) parameters at the time of subscription. The supported QoS parameters are described in the sections below.

44.10.1 Reliability

The QoS parameter `reliability` affects message delivery. The only legal values at this point are `ordered` and the empty string. If not specified, the default value is the empty string (meaning not ordered).

If you specify `ordered` as the reliability QoS, IceStorm forwards events to subscribers in the order in which they are received. Without this setting, events are forwarded immediately, as soon as they are received; because events can arrive from different publishers publishing to the same topic, this means that they can be forwarded to subscribers in an order that differs from the order in which they were received.

Whether the subscriber receives events in the same order in which they are sent by IceStorm also depends on the subscriber's threading model—see Section 44.11.

44.10.2 Retry Count

IceStorm automatically removes a subscriber if `ObjectNotExistException` or `NotRegisteredException` is raised while attempting to deliver an event. IceStorm considers these exceptions as indicators of a hard failure, after which it is unnecessary to continue event delivery.

For other kinds of failures, IceStorm uses the QoS parameter `retryCount` to determine when to remove a subscriber. A value of `-1` means IceStorm retries forever and never automatically removes a subscriber unless a hard failure occurs. A value of zero means IceStorm never retries and immediately removes the subscriber. For positive values, IceStorm decrements the subscriber's retry count for each failure and removes the subscriber once it reaches zero. Linked topics always have a configured retry count of `-1`. The default value of the `retryCount` parameter is zero.

A retry count of `-1` adds some resiliency to your IceStorm application by ignoring intermittent network failures such as `ConnectionRefusedException`. However, there is also some risk inherent in using a retry count of `-1` because an improperly configured subscriber may never be removed. For example, consider what happens when a subscriber registers using a transient endpoint: if that subscriber happens to terminate and resubscribe with a different endpoint, IceStorm will continue trying to deliver events to the subscriber at its old endpoint. IceStorm can only remove the subscriber if it receives a hard error, and that is only possible when the subscriber is reachable.

To use a retry count of `-1` successfully, the subscriber should either register with a fixed endpoint, or use IceGrid to take advantage of indirect proxies and automatic activation (see Chapter 38). Furthermore, if the subscriber is expected to function correctly after a restart of its process, the subscriber must use the same identity. The application can rely on the `subscribeAndGetPublisher` operation to raise `AlreadySubscribed` when the subscriber is already subscribed.

44.10.3 Example

The Slice type `IceStorm::QoS` is defined as a `dictionary` whose key and value types are both `string`, therefore the QoS parameter name and value are both represented as strings. The example code presented in Section 44.5.3 used an empty dictionary for the QoS argument, meaning default values are used. The C++ and Java examples shown below illustrate how to set the `reliability` parameter to `ordered`.

C++ Example

```
IceStorm::QoS qos;
qos["reliability"] = "ordered";
topic->subscribeAndGetPublisher(qos, proxy->ice_twoway());
```

Java Example

```
java.util.Map qos = new java.util.HashMap();
qos.put("reliability", "ordered");
topic.subscribeAndGetPublisher(qos, proxy.ice_twoway());
```

44.11 Delivery Mode

The delivery mode for events sent to subscribers is controlled by the proxy that the subscriber passes to IceStorm. For example, if the subscriber subscribes with a oneway proxy, events will be forwarded by IceStorm as oneway messages. Subscribers can use the following proxies:

- Twoway

  Each event is sent to the subscriber as a separate twoway message. Using this delivery mode allows the subscriber to enable server-side active connection management without risking lost messages (see Section 36.4) because IceS-

torm will re-send an event if the subscriber happens to close its connection at the wrong moment.

If you combine a twoway proxy with a `reliability` QoS of `ordered`, messages will be forwarded to the subscriber in the order in which they are received. This is guaranteed because IceStorm will wait for a reply from the subscriber for each event before sending the next event.

Without ordered delivery, events may be delivered out-of-order to the subscriber because IceStorm will send an event as soon as possible (without waiting for a reply for the preceding event). If the subscriber uses a thread pool with more than one thread, this can result in out-of-order dispatch of messages in the subscriber.

For single-threaded subscribers and subscribers using a serialized thread pool (see Section 32.10), twoway delivery always results in in-order dispatch of events in the subscriber.

With twoway delivery, IceStorm is informed of any failure to deliver an event by the Ice run time. For example, IceStorm may not be able to establish a connection to a subscriber, or may receive an `ObjectNotExistException` when it forwards an event. Any failure to deliver an event to a subscriber (possibly after a transparent retry by the Ice run time) results in the cancellation of the corresponding subscription.

- Oneway

  Each event is sent to the subscriber as a oneway message. If more than one event is ready to be delivered, the events are sent in a single batch. This delivery mode is more efficient than using twoway delivery. However, the subscriber cannot use active connection management without the risk of events being lost. In addition, if something goes wrong with the subscriber, such as the subscriber having destroyed its callback object without unsubscribing, or having subscribed an object with the wrong interface, IceStorm does not notice the failure and will continue to send events to the non-existent subscriber object for as long as it can maintain a connection to the subscriber's endpoint.

  For multi-threaded subscribers, oneway delivery can result in out-of-order delivery of events. For single-threaded subscribers and subscribers using a serialized thread pool, events are delivered in order.

- Batch Oneway

  With this delivery mode, IceStorm buffers events from publishers and sends them in batches to the subscriber (see Section 32.16). This reduces network overhead and is more efficient than oneway delivery. However, as for oneway delivery, the subscriber cannot use active connection management without the risk of losing events. In addition, events can be delivered out of order if the subscriber is multi-threaded. Batch oneway delivery, while providing better throughput, increases latency because events arrive in "bursts". You can control the interval at which batched events are flushed by setting the `IceStorm.Flush.Timeout` Property (see page 1927).

- Datagram

  With this delivery mode, events are forwarded as UDP messages, optionally with multicast semantics. Obviously, this means that events can be delivered out of order, can be lost, and can even be duplicated. In addition, IceStorm cannot detect anything about the delivery status of events. This means that if a subscriber disappears without unsubscribing, IceStorm will attempt to forward events to the subscriber indefinitely. If you use datagram delivery, you need to be careful that subscribers unsubscribe before they disappear; otherwise, stale subscriptions can accumulate in IceStorm over time, bogging down the service as it delivers more and more events to no-longer-existent subscribers.

- Batch Datagram

  With this delivery mode, events are forwarded as batches within a datagram. The same considerations as for datagram delivery and oneway batched delivery apply here. In addition, keep in mind that, due to the size limit for datagrams, batched datagram delivery makes sense only if events are small. (You should also consider enabling compression with this delivery mode.)

44.12 Configuring IceStorm

IceStorm is a relatively lightweight service in that it requires very little configuration and is implemented as an IceBox service (see Chapter 43). The configuration properties supported by IceStorm are described in Appendix D; some of them control diagnostic output and are not discussed in this chapter.

44.12.1 Property Prefix

As you will see in the description of the IceStorm properties in Appendix D, IceStorm uses its IceBox service name as the prefix for all of its properties. For example, the property *service*.`TopicManager.Endpoints` becomes `DemoIceStorm.TopicManager.Endpoints` when IceStorm is configured as the IceBox service `DemoIceStorm`.

44.12.2 Server Configuration

The first step is configuring IceBox to run the IceStorm service:

```
IceBox.Service.DemoIceStorm=IceStormService,34:createIceStorm --
Ice.Config=config.service
```

In this example, the IceStorm service itself is configured by the properties in the `config.service` file, which might look as follows for a non-replicated service:

```
Freeze.DbEnv.DemoIceStorm.DbHome=db
DemoIceStorm.TopicManager.Endpoints=tcp -p 9999
DemoIceStorm.Publish.Endpoints=tcp -p 10000
```

In this example, IceStorm uses Freeze to manage the service's persistent state, therefore the first property specifies the pathname of the Freeze database environment directory (see Chapter 39) for the service. Here the directory `db` is used, which must already exist in the current working directory. This property can be omitted when the service is running in transient mode; see the description of the *service*.`Transient` property in Appendix D for more information.

The final two properties specify the endpoints used by the IceStorm object adapters; notice that their property names begin with `DemoIceStorm`, matching the service name. The `TopicManager` property specifies the endpoints on which the `TopicManager` and `Topic` objects reside; these endpoints must use a connection-oriented protocol such as TCP or SSL. The `Publish` property specifies the endpoints used by topic publisher objects; using datagram endpoints in this property is possible but carries additional risk (see Section 44.5.2 for more information on publisher objects).

Threads

IceStorm's default thread pool configuration is sufficient when the service is running on a single CPU machine. On a host with multiple CPUs, you may be able to improve IceStorm's performance by increasing the size of its client-side thread

pool using the `Ice.ThreadPool.Client.*` properties, but the optimal number of threads can only be determined with careful benchmarking. Refer to Section 32.10 for more information on the Ice threading model.

44.12.3 Deploying IceStorm Replicas

There are two ways of deploying IceStorm in its highly available (replicated) mode. In both cases, adding another replica requires that all active replicas be stopped while their configurations are updated; it is not possible to add a replica while replication is running.

To remove a replica, stop all replicas and alter the configuration as necessary. You must be careful not to remove a replica if it has the latest database state. This situation will never occur during normal operation since the database state of all replicas is identical. However, in the event of a crash it is possible for a coordinator to have later database state than all replicas. The safest approach is to verify that all replicas are active prior to stopping them. You can do this using the **`icestormadmin`** utility by checking that all replicas are in the `Normal` state (see Section 44.8).

IceGrid Deployment

IceGrid (see Chapter 38) is a convenient way of deploying IceStorm replicas. The term *replica* is also used in the context of IceGrid, specifically when referring to groups of object adapters that participate in replication, as described in Section 38.9. It is important to be aware of the distinction between IceStorm replication and object adapter replication; IceStorm replication *uses* object adapter replication when deployed with IceGrid, but IceStorm does not *require* object adapter replication as you will see in the next section.

An IceGrid deployment typically uses two adapter replica groups: one for the publisher proxies, and another for the topics, as shown below:

```
<replica-group id="DemoIceStorm-PublishReplicaGroup">
</replica-group>

<replica-group id="DemoIceStorm-TopicManagerReplicaGroup">
    <object identity="DemoIceStorm/TopicManager"
        type="::IceStorm::TopicManager"/>
</replica-group>
```

The object adapters are then configured to use these replica groups:

```
<adapter name="${service}.Publish"
    endpoints="tcp"
    replica-group="${instance-name}-PublishReplicaGroup"/>

<adapter name="${service}.TopicManager"
    endpoints="tcp"
    replica-group="${instance-name}-TopicManagerReplicaGroup"/>
```

As discussed in Section 44.7.5, an application may not want publisher proxies to contain multiple endpoints. In this case you should remove `PublishReplicaGroup` from the above deployment.

The next step is defining the endpoints for the adapter `Node`, which is used internally for communication with other IceStorm replicas and is not part of an adapter replica group:

```
<adapter name="${service}.Node" endpoints="tcp"/>
```

Finally, you must define the node id for each IceStorm replica using the `NodeId` property. The node id must be a non-negative integer:

```
<property name="${service}.NodeId" value="${index}"/>
```

You can find a complete example of an IceGrid deployment in the directory `demo/IceStorm/replicated` in the C++ distribution.

Manual Deployment

You can also deploy IceStorm replicas without IceGrid, although it requires more manual configuration; an IceGrid deployment is simpler to maintain.

The first step is defining the set of node proxies using properties of the form `Nodes.`*`id`*. These proxies allow replicas to contact each other; their object identities are composed using *`instance-name`*`/node`*`id`*.

For example, assuming we are using the IceBox service name `IceStorm` and have three replicas with the identifiers 0, 1, 2 and an instance name of `DemoIceStorm`, we can configure the proxies as shown below:

```
IceStorm.InstanceName=DemoIceStorm
IceStorm.Nodes.0=DemoIceStorm/node0:tcp -p 13000
IceStorm.Nodes.1=DemoIceStorm/node1:tcp -p 13010
IceStorm.Nodes.2=DemoIceStorm/node2:tcp -p 13020
```

These properties must be defined in each replica. Additionally, each replica must define its node id, as well as the node's endpoints. For example, we can configure node 0 as follows:

```
IceStorm.NodeId=0
IceStorm.Node.Endpoints=tcp -p 13000
```

The endpoints for each replica and id must match the proxies configured in the `Nodes.`*`id`* properties.

Two additional properties allow you to configure replicated endpoints:

- *`service-name`*`.ReplicatedTopicManagerEndpoints`

  Defines the endpoints contained in proxies returned by the topic manager.

- *`service-name`*`.ReplicatedPublishEndpoints`

  Defines the endpoints contained in the publisher proxy returned by the topic.

For example, suppose we configure three replicas:

```
IceStorm.NodeId=0
IceStorm.TopicManager.Endpoints=tcp -p 10000
IceStorm.Publish.Endpoints=tcp -p 10001:udp -p 10001

IceStorm.NodeId=1
IceStorm.TopicManager.Endpoints=tcp -p 10010
IceStorm.Publish.Endpoints=tcp -p 10011:udp -p 10011

IceStorm.NodeId=2
IceStorm.TopicManager.Endpoints=tcp -p 10020
IceStorm.Publish.Endpoints=tcp -p 10021:udp -p 10021
```

Each replica should also define these properties:

```
IceStorm.ReplicatedPublishEndpoints=tcp -p 10001:tcp -p 10011:tcp
-p 10021:udp -p 10001:udp -p 10011:udp -p 10021
IceStorm.ReplicatedTopicManagerEndpoints=tcp -p 10000:tcp -p 10010
:tcp -p 10020
```

As discussed in Section 44.7.5, an application may not want publisher proxies to contain multiple endpoints. In this case you should remove the definition of the `ReplicatedPublishEndpoints` property from the above deployment.

You can find a complete example of a manual deployment in the directory `demo/IceStorm/replicated2` in the C++ distribution.

44.12.4 Client Configuration

Clients of the service can define a proxy for the `TopicManager` object as follows:

```
TopicManager.Proxy=IceStorm/TopicManager:tcp -p 9999
```

The name of the property is not relevant, but the endpoint must match that of the *service*.TopicManager.Endpoints property, and the object identity must use the IceStorm instance name as the category and TopicManager as the name.

44.12.5 Object Identities

IceStorm hosts one well-known object, which implements the IceStorm::TopicManager interface. The default identity of this object is IceStorm/TopicManager, as seen in the stringified proxy example from Section 44.12.4. If an application requires the use of multiple IceStorm services, it is a good idea to assign unique identities to the well-known objects by configuring the services with different values for the *service*.InstanceName property, as shown in the following example:

```
DemoIceStorm.InstanceName=Measurement
```

This property changes the category of the object's identity, which becomes Measurement/TopicManager. The client's configuration must also be changed to reflect the new identity:

```
TopicManager.Proxy=Measurement/TopicManager:tcp -p 9999
```

44.12.6 Using a Different Database

By default, IceStorm uses a Freeze database (see Chapter 39) to store its persistent state. You can configure IceStorm to use an SQL database by setting a number of properties.

If you use IceStorm with an SQL database, its persistent state is stored in two tables:

- *<service-name>[_<node-id>]*_SubscriberMap
- *<service-name>[_<node-id>]*_LLU

A number of properties control how IceStorm accesses an SQL database:

- Ice.Plugin.DB

  To use IceStorm with an SQL database, you must set this property to the value IceStormSqlDB:createSqlDB.

- Ice.Plugin.SQLThreadHook

  This property is necessary to ensure that SQL resources are cleaned up properly and must be set to IceStormService:createThreadHook.

- *service*.SQL.DatabaseType

- *service*.SQL.DatabaseName
- *service*.SQL.HostName
- *service*.SQL.Port
- *service*.SQL.UserName
- *service*.SQl.Password

  These properties are described in detail in Appendix D.

44.13 Summary

IceStorm is a publish/subscribe service that offers Ice applications a flexible and efficient means of publishing unidirectional requests to a group of subscribers. IceStorm simplifies development by relieving the application from the burden of managing subscribers and handling delivery errors, allowing the application to focus on publishing its data and not on the minutiae of distribution. Its support for replication and integration with IceGrid provides greater reliability in the face of network and system failures without complex administrative burdens. Finally, IceStorm leverages Ice request-forwarding facilities in order to provide a typed interface to publishers and subscribers, minimizing the impact on applications.

Chapter 45
IcePatch2

45.1 Chapter Overview

This chapter presents IcePatch2,[1] the Ice solution for secure replication of a directory tree. Section 45.2 provides an overview of IcePatch2 concepts and operation, and Sections 45.3 to 45.5 discuss how to prepare a file set and how to run the IcePatch2 client and server. Section 45.6 shows how to configure multiple IcePatch2 servers, and Section 45.7 describes a C++ utility library for use in custom IcePatch2 clients.

45.2 Introduction

IcePatch2 is an efficient file patching service that is easy to configure and use. It includes the following components:

- the IcePatch server (**`icepatch2server`**)
- a text-based IcePatch client (**`icepatch2client`**)

1. IcePatch2 supersedes IcePatch, which was a previous version of this service.

- a text-based tool to compress files and calculate checksums (**`icepatch2calc`**)
- a Slice API and C++ convenience library for developing custom IcePatch2 clients

As with all Ice services, IcePatch2 can be configured to use Ice facilities such as Glacier2 for firewall support and IceSSL for secure communication.

IcePatch2 is conceptually quite simple. The server is given responsibility for a file system directory (the *data directory*) containing the files and subdirectories that are to be distributed to IcePatch2 clients. You use **`icepatch2calc`** to compress these files and to generate an index containing a checksum for each file. The server transmits the compressed files to the client, which recreates the data directory and its contents on the client side, patching any files that have changed since the previous run.

IcePatch2 is efficient: transfer rates for files are comparable to what you would get using **`ftp`**.

45.3 Using `icepatch2calc`

Suppose we have the directories and files shown in Figure 45.1 in the data directory on the server side.

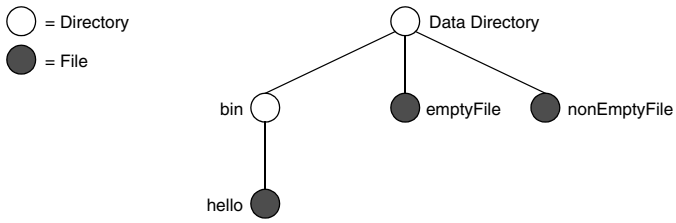


Figure 45.1. An example data directory.

Assume that the file named `emptyFile` is empty (contains zero bytes) and that the remaining files contain data.

To prepare this directory for the transmission by the server, you must first run **`icepatch2calc`**. (The command shown assumes that the data directory is the current directory.)

```
$ icepatch2calc .
```

After running this command, the contents of the data directory are as shown in Figure 45.2.

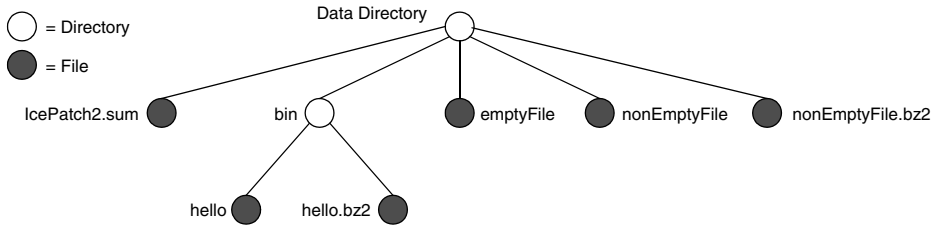


Figure 45.2. Contents of the data directory after running `icepatch2calc`.

Note that **`icepatch2calc`** compresses the files in the data directory (except for `emptyFile`, which is not compressed). Also note that **`icepatch2calc`** creates an additional file, `IcePatch2.sum` in the data directory. The contents of this file are as follows:

```
. 3a52ce780950d4d969792a2559cd519d7ee8c727 -1
./bin bd362140a3074eb3edb5e4657561e029092c3d91 -1
./bin/hello 77b11db586a1f20aab8553284241bb3cd532b3d5 70
./emptyFile 082c37fc2641db68d195df83844168f8a464eada 0
./nonEmptyFile aec7301c408e6ce184ae5a34e0ea46e0f0563746 72
```

Each line in the checksum file contains the name of the *uncompressed* file or directory (relative to the data directory), the checksum of the *uncompressed* file, and a byte count. For directories, the count is `-1`; for uncompressed files, the count is `0`; for compressed files, the count is the number of bytes in the *compressed* file. The lines in the file are sorted alphabetically by their pathname.

If you add files or delete files from the data directory or make changes to existing files, you must stop the server, run **`icepatch2calc`** again to update the `IcePatch2.sum` checksum file, and restart the server.

45.3.1 `icepatch2calc` Options

`icepatch2calc` has the following syntax:

```
icepatch2calc [options] data_dir [file...]
```

Normally, you will run **`icepatch2calc`** by simply specifying a data directory, in which case the program traverses the data directory, compresses all files, and creates an entry in the checksum file for each file and directory.

You can also nominate specific files or directories on the command line. In this case, **icepatch2calc** only compresses and calculates checksums for the specified files and directories. This is useful if you have a very large file tree and want to refresh the checksum entries for only a few selected files or directories that you have updated. (In this case, the program does not traverse the entire data directory and, therefore, will also not detect any updated, added, or deleted files, except in any of the specified directories.) Any file or directory names you specify on the command line must either be pathnames relative to the data directory or, if you use absolute pathnames, those pathnames must have the data directory as a prefix.

The command supports the following options:

- **-h, --help**

  Displays a help message.

- **-v, --version**

  Displays the version number.

- **-z, --compress**

  Normally, **icepatch2calc** scans the data directory and compresses a file only if no compressed version exists, or if the compressed version of a file has a modification time that predates that of the uncompressed version. If you specify **-z**, the tool re-scans and recompresses the entire data directory, regardless of the time stamps on files. This option is useful if you suspect that time stamps in the data directory may be incorrect.

- **-Z, --no-compress**

  This option allows you to create a client-side checksum file.

  Do not use this option when creating the checksum file for the server—the option is for creating a client-side `IcePatch2.sum` file for updates of software on distribution media (see page 1820).

- **-i, --case-insensitive**

  This option disallows file names that differ only in case. (An error message will be printed if **icepatch2calc** encounters any files that differ in case only.) This is particularly useful for Unix servers with Windows clients, since Windows folds the case of file names, and therefore such files would override each other on the Windows client.

- **-V, --verbose**

  This option prints a progress message for each file that is compressed and for each checksum that is computed.

45.4 Running the Server

Once you have run **icepatch2calc** on the data directory, you can start the **icepatch2server**:

```
$ icepatch2server .
```

The server expects the data directory as its single command-line argument. If you omit to specify the data directory, the server uses the setting of the `IcePatch2.Directory` property (see Appendix D) to determine the data directory.

The server has two different sets of endpoints, one for regular operations, and one for administration:

- `IcePatch2.Endpoints`

  This property determines the endpoint at which the server listens for client requests. This property must be specified.

- `IcePatch2.Admin.Endpoints`

  If this property is not set, the only way to shut down the server is to kill it somehow, such as by interrupting the server from the command line. If this property is set, the server offers an additional `IcePatch2::Admin` interface:

  ```
  interface Admin {
      void shutdown();
  };
  ```

  By default, the identity of this object is `IcePatch2/admin`. You can change the category of this identity by setting the property `IcePatch2.Instance-Name`.

  Calling the `shutdown` operation shuts down the server. Note that any client with access to the `Admin` interface's port can stop the server. Typically, you would set this property to a port that is not accessible to potentially hostile clients.

45.4.1 `icepatch2server` Options

Regardless of whether you run the server under Windows or a Unix-like operating system, it provides the following options:

- **`-h, --help`**

  Displays a help message.

- **`-v, --version`**

  Displays a version number.

Additional command line options are supported, including those that allow the router to run as a Windows service or Unix daemon. See Section 8.3.2 for more information.

45.5 Running the Client

Once the **`icepatch2server`** is running, you can use **`icepatch2client`** to get a copy of the data directory that is maintained by the server. For example:

```
$ icepatch2client --IcePatch2.Endpoints="tcp -h somehost.com \
> -p 10000" .
```

The client expects the data directory as its single command-line argument. As for the server, you must specify the `IcePatch2.Endpoints` property so the client knows where to find the server.

If you have not run the client previously, it asks you whether you want to do a thorough patch. You must reply "yes" at this point (or run the client with the **`-t`** option—see page 1819). The client then executes the following steps:

1. It traverses the local data directory and creates a local `IcePatch2.sum` checksum file.
2. It obtains the relevant list of checksums from the server and compares it to the list of checksums it has built locally:
   1. The client deletes each file that appears in the local checksum file but not in the server's file.
   2. The client retrieves every file that appears in the server's checksum file, but not in the local checksum file.
   3. The client patches every file that, locally, has a checksum that differs from the corresponding checksum on the server side.

When the client finishes, the contents of the data directory on the client side exactly match the contents of the data directory on the server side. However, only the uncompressed files are created on the client side—the server stores the compressed version of the files simply to avoid redundantly compressing a file every time it is retrieved by a client.

Note that, on the initial patch, any files that exist in the client's data directory are deleted or, if they have the same name as a file on the server, will be overwritten with the corresponding file as it exists on the server.

Using `icepatch2client` for Partial Updates

Once you have run the client, the client-side data directory contains an `IcePatch2.sum` file that reflects the contents of the data directory. If you run **`icepatch2client`** a second time, the program uses the contents of the local checksum file: for each entry in the local checksum file, the client compares the local checksum with the server-side checksum for the same file; if the checksums differ, the client updates the corresponding file. In addition, the client deletes any files that appear in the client's checksum file but not in the server's checksum file, and it fetches any files that appear in the server's checksum file but are missing from the client's checksum file.

If you edit a client-side file and change its contents, **`icepatch2client`** does *not* realize that this has happened and therefore will not patch the file to be in sync with the version on the server again. This is because the client does not automatically recompute the checksum for a file to see whether the stored checksum in `IcePatch2.sum` still agrees with the actual checksum for the current file contents.

Similarly, if you create an arbitrary file on the client side, but that file is not mentioned in either the client's or the server's checksum file, that file will simply be left alone. In other words, a normal patch operates on the differences between the client's and server's checksum files, not on any differences that could be detected by examining the contents of the file system.

If you have locally created files that have nothing to do with the distribution or if you have locally modified some files and want to make sure that those modified files are updated to reflect the contents of the same files on the server side, you must run a thorough patch with the **`-t`** option. This forces the client to traverse the local data directory and recompute the checksum for each file, and then compare these checksums against the server-side ones. As a result, if you edit a file locally so it differs from the server-side version, **`-t`** forces that file to be updated. Simi-

larly, if you have added a file of your own on the client side that does not have a counterpart on the server side, that file will be deleted by a thorough patch.

Preventing Deletion of Local Files

By default, a normal patch deletes any files that appear in the client's checksum file but that are absent in the server's checksum file. Similarly, by default, a thorough patch deletes all files in the local data directory that do not appear in the server's checksum file. If you do not want this behavior, you can set the `IcePatch2.Remove` property to 0 (the default value is 1). This prevents deletion of files and directories that exist only on the client side, whether the patch is a normal patch or a thorough patch.

Patching Software Installed from Media

Suppose you distribute your application on a DVD that clients use to install the software. The DVD might be out of date so, after installation, the install script needs to perform a patch to update the application to the latest version. The script can perform a thorough patch to do this but, for large file sets, this is expensive because the client has to recompute the checksum for every file in the distribution.

To avoid this cost, you can place all the files for the distribution into a directory on the server and run **`icepatch2calc -Z`** on that directory. With the **`-Z`** option, **`icepatch2calc`** creates a checksum file with the correct checksums, but with a file size of 0 for each file, that is, the **`-Z`** option omits compressing the files (and the considerable cost associated with that). Once you have created the new `IcePatch2.sum` file in this way, you can include it on the DVD and install it on the client along with all the other files.

This guarantees that the checksum file on the client is in agreement with the actual files that were just installed and, therefore, it is sufficient for the install script to do a normal patch to update the distribution and so avoid the cost of recomputing the checksum for every file.

Setting Transfer Size

You can set the `IcePatch2.ChunkSize` property to control the number of bytes that the client fetches per request. The default value is 100 kilobytes.

45.5.1 `icepatch2client` Options

The client supports the following options:

- **`-h, --help`**

  Displays a help message.

- **`-v, --version`**

  Displays a version number.

- **`-t, --thorough`**

  Do a thorough patch, recomputing all checksums.

45.6 Object Identities

An IcePatch2 service hosts two well-known objects, which implement the `IcePatch2::FileServer` and `IcePatch2::Admin` interfaces and have the default identity `IcePatch2/server` and `IcePatch2/admin`, respectively. If an application requires the use of multiple IcePatch2 services, it is a good idea to assign unique identities to the well-known objects by configuring the servers with different values for the `IcePatch2.InstanceName` property, as shown in the following example:

```
$ icepatch2server --IcePatch2.InstanceName=PublicFiles ...
```

This property changes the category of the objects' identities, which become `PublicFiles/server` and `PublicFiles/admin`, respectively. The client's configuration must also be changed to reflect the new identity:

```
$ icepatch2client --IcePatch2.Endpoints="tcp -h somehost.com \
> -p 10000" --IcePatch2.InstanceName=PublicFiles .
```

45.7 The IcePatch2 Client Utility Library

IcePatch2 includes a pair of C++ classes that simplify the task of writing your own patch client, along with a Microsoft Foundation Classes (MFC) example that shows how to use these classes. You can find the MFC example in the subdirectory `demo/IcePatch2/MFC` of your Ice distribution.

The remainder of this section discusses the classes. To incorporate them into a custom patch client, your program must include the header file `IcePatch2/ClientUtil.h` and link with the `IcePatch2` library.

45.7.1 Performing a Patch

The `Patcher` class encapsulates all of the patching logic required by a client:

```
namespace IcePatch2 {
class Patcher : ... {
public:

    Patcher(const Ice::CommunicatorPtr& communicator,
            const PatcherFeedbackPtr& feedback);

    Patcher(const FileServerPrx& server,
            const PatcherFeedbackPtr& feedback,
            const std::string& dataDir, bool thorough,
            Ice::Int chunkSize, Ice::Int remove);

    bool prepare();

    bool patch(const std::string& dir);

    void finish();
};
typedef IceUtil::Handle<Patcher> PatcherPtr;
}
```

Constructing a Patcher

The constructors provide two ways of configuring a `Patcher` instance. The first form obtains the following IcePatch2 configuration properties from the supplied communicator:

- `IcePatch2.InstanceName`
- `IcePatch2.Endpoints`
- `IcePatch2.Directory`
- `IcePatch2.Thorough`
- `IcePatch2.ChunkSize`
- `IcePatch2.Remove`

The second constructor accepts arguments that correspond to each of these properties.

Both constructors also accept a `PatcherFeedback` object (see Section 45.7.2), which allows the client to monitor the progress of the patch.

Executing the Patch

`Patcher` provides three methods that reflect the three stages of a patch:

- `bool prepare()`

  The first stage of a patch includes reading the contents of the checksum file (if present), retrieving the file information from the server, and examining the local data directory to compose the list of files that require updates. The `PatcherFeedback` object is notified incrementally about each local task and has the option of aborting the patch at any time. This method returns true if patch preparation completed successfully, or false if the `PatcherFeedback` object aborted the patch. If an error occurs, `prepare` raises an exception in the form of a `std::string` containing a description of the problem.

- `bool patch(const std::string& dir)`

  The second stage of a patch updates the files in the local data directory. If the `dir` argument is an empty string or `"."`, `patch` updates the entire data directory. Otherwise, `patch` updates only those files whose path names begin with the path in `dir`. For each file requiring an update, `Patcher` downloads its compressed data from the server and writes it to the local data directory. The `PatcherFeedback` object is notified about the progress of each file and, as in the preparation stage, may abort the patch if necessary. This method returns true if patching completed successfully, or false if the `PatcherFeedback` object aborted the patch. If an error occurs, `patch` raises an exception in the form of a `std::string` containing a description of the problem.

- `void finish()`

  The final stage of a patch writes a new checksum file to the local data directory. If an error occurs, `finish` raises an exception in the form of a `std::string` containing a description of the problem.

The code below demonstrates a simple patch client:

```
#include <IcePatch2/ClientUtil.h>
...
Ice::CommunicatorPtr communicator = ...;
IcePatch2::PatcherFeedbackPtr feedback = new MyPatcherFeedbackI;
IcePatch2::PatcherPtr patcher =
    new IcePatch2::Patcher(communicator, feedback);

try {
    bool aborted = !patcher->prepare();
    if(!aborted)
```

```
        aborted = !patcher->patch("");
    if(!aborted)
        patcher->finish();
    if(aborted)
        cerr << "Patch aborted" << endl;
} catch(const string& reason) {
    cerr << "Patch error: " << reason << endl;
}
```

For a more sophisticated example, see `demo/IcePatch2/MFC` in your Ice distribution.

45.7.2 Monitoring Patch Progress

The class `PatcherFeedback` is an abstract base class that allows you to monitor the progress of a `Patcher` object (see Section 45.7.1). The class declaration is shown below:

```
namespace IcePatch2 {
class PatcherFeedback : ... {
public:

    virtual bool noFileSummary(const std::string& reason) = 0;

    virtual bool checksumStart() = 0;
    virtual bool checksumProgress(const std::string& path) = 0;
    virtual bool checksumEnd() = 0;

    virtual bool fileListStart() = 0;
    virtual bool fileListProgress(Ice::Int percent) = 0;
    virtual bool fileListEnd() = 0;

    virtual bool patchStart(
        const std::string& path, Ice::Long size,
        Ice::Long updated, Ice::Long total) = 0;
    virtual bool patchProgress(
        Ice::Long pos, Ice::Long size,
        Ice::Long updated, Ice::Long total) = 0;
    virtual bool patchEnd() = 0;
};
typedef IceUtil::Handle<PatcherFeedback> PatcherFeedbackPtr;
}
```

Each of these methods returns a boolean value: true allows `Patcher` to continue, and false directs `Patcher` to abort the patch. The methods are described below:

- `bool noFileSummary(const std::string& reason)`

  Invoked when the local checksum file cannot be found. Returning true initiates a thorough patch, while returning false causes `Patcher::prepare` to return false (see page 1823).

- ```
  bool checksumStart()
  bool checksumProgress(const std::string& path)
  bool checksumEnd()
  ```

  Invoked by `Patcher::prepare` during a thorough patch. The `checksumProgress` method is invoked as each file's checksum is being computed.

- ```
  bool fileListStart()
  bool fileListProgress(Ice::Int percent)
  bool fileListEnd()
  ```

  Invoked by `Patcher::prepare` when collecting the list of files to be updated. The `percent` argument to `fileListProgress` indicates how much of the collection process has completed so far.

- ```
  bool patchStart(
      const std::string& path, Ice::Long size,
      Ice::Long updated, Ice::Long total)
  bool patchProgress(
      Ice::Long pos, Ice::Long size,
      Ice::Long updated, Ice::Long total)
  bool patchEnd()
  ```

  For each file that requires updating, `Patcher::patch` invokes `patchStart` to indicate the beginning of the patch, `patchProgress` one or more times as chunks of the file are downloaded and written, and finally `patchEnd` to signal the completion of the file's patch. The `path` argument supplies the path name of the file, and `size` provides the file's compressed size. The `pos` argument denotes the number of bytes written so far, while `updated` and `total` represent the cumulative number of bytes updated so far and the total number of bytes to be updated, respectively, of the entire patch operation.
  ```

45.8 Summary

IcePatch2 addresses a requirement common to both development and deployment scenarios: the safe, secure, and efficient replication of a directory tree. The IcePatch2 server is easy to configure and efficient. For simple uses, IcePatch2 provides a client that can be used to patch directory hierarchies from the command line. With the C++ utility library, you can also create custom patch clients if you require better integration of the client with your application.

Appendixes

Appendix A
Slice Keywords

The following identifiers are Slice keywords:

`bool`	`enum`	`implements`	`module`	`struct`
`byte`	`exception`	`int`	`Object`	`throws`
`class`	`extends`	`interface`	`out`	`true`
`const`	`false`	`local`	`sequence`	`void`
`dictionary`	`float`	`LocalObject`	`short`	
`double`	`idempotent`	`long`	`string`	

Keywords must be capitalized as shown.

Appendix B
Slice Metadata Directives

B.1 General Metadata Directives

`ami`

This directives applies to interfaces, classes, and individual operations. It enables code generation for asynchronous method invocation, described in Appendix K.

NOTE: This directive applies to the deprecated AMI mapping described in Appendix K. For the new AMI mapping there is no need for this directive.

`amd`

This directive applies to interfaces, classes, and individual operations. It enables code generation for asynchronous method dispatch. (See the relevant language mapping chapter for details.)

`deprecated`

This directive allows you to emit a deprecation warning for Slice constructs. See Section 4.19 for more detail.

`protected`

This directive applies to data members of classes and changes code generation to make these members protected. See class mapping of the relevant language mapping chapter for more information.

`UserException`

This directive applies only to operations on local interfaces. The metadata directive indicates that the operation can throw any user exception, regardless of its specific definition. (This directive is used for the `locate` and `finished` operations on servant locators, which can throw any user exception.)

B.2 Metadata Directives for C++

`cpp:array`
`cpp:range:array`

These directives apply to sequences. They direct the code generator to create zero-copy APIs for passing sequences as parameters. See Section 6.7.4 for details.

`cpp:class`

This directive applies to structures. It directs the code generator to create a C++ class (instead of a C++ structure) to represent a Slice structure. (See Section 6.7.2.)

`cpp:const`

This directive applies to operations. It directs the code generator to create a `const` pure virtual member function for the skeleton class. (See Section 8.4.2.)

`cpp:type:wstring`

This directive applies to data members of type string as well as to containers, such as structures, classes, and exceptions. It changes the default mapping for strings from `std::string` to `std::wstring`. (See Section 6.6.1.)

cpp:header-ext

This global directive allows you to use a file extension for C++ header files other than the default `.h` extension. (See Section 6.16.)

cpp:include

This global directive allows you inject additional #include directives into the generated code. This is useful for custom types. (See Section 6.7.4.)

cpp:virtual

This directive applies to classes. If the directive is present and a class has base classes, the generated C++ class derives virtually from its bases; without this directive, **`slice2cpp`** generates the class so it derives non-virtually from its bases.

This directive is useful if you use Slice classes as servants and want to inherit the implementation of operations in the base class in the derived class. For example:

```
class Base {
    int baseOp();
};

["cpp:virtual"]
class Derived extends Base {
    string derivedOp();
};
```

The metadata directive causes **`slice2cpp`** to generate the class definition for `Derived` using virtual inheritance:

```
class Base : virtual public Ice::Object {
    // ...
};

class Derived : virtual public Base {
    // ...
};
```

This allows you to reuse the implementation of `baseOp` in the servant for `Derived` using ladder inheritance:

```
class BaseI : public virtual Base {
    Ice::Int baseOp(const Ice::Current&);
    // ...
};

class DerivedI : public virtual Derived,
                 public virtual BaseI {
    // Re-use inherited baseOp()
};
```

Note that, if you have data member in classes and use virtual inheritance, you need to take care to correctly call base class constructors if you implement your own one-shot constructor. For example:

```
class Base {
    int baseInt;
};

class Derived extends Base {
    int derivedInt;
};
```

The generated one-shot constructor for `Derived` initializes both `baseInt` and `derivedInt`:

```
Derived::Derived(Ice::Int __ice_baseInt,
                 Ice::Int __ice_derivedInt)
    : M::Base(__ice_baseInt),
      derivedInt(__ice_derivedInt)
{
}
```

If you derive your own class from `Derived` and add a one-shot constructor to your class, you must explicitly call the constructor of all the base classes, including `Base`. Failure to call the `Base` constructor will result in `Base` being default-constructed instead of getting a defined value. For example:

```
class DerivedI : public virtual Derived {
public:
    DerivedI(int baseInt, int derivedInt, const string& s)
        : Base(baseInt), Derived(baseInt, derivedInt), _s(s)
    {
    }

private:
    string _s;
};
```

This code correctly initializes the `baseInt` member of the `Base` part of the class. Note that the following does *not* work as intended and leaves the `Base` part default-constructed (meaning that `baseInt` is not initialized):

```
class DerivedI : public virtual Derived {
public:
    DerivedI(int baseInt, int derivedInt, const string& s)
        : Derived(baseInt, derivedInt), _s(s)
    {
        // WRONG: Base::baseInt is not initialized.
    }

private:
    string _s;
};
```

B.3 Metadata Directives for Java

`java:package`

This global directive instructs the code generator to place the generated classes into a specific package. (See Section 10.16.1.)

`java:getset`

This directive applies to data members and structures, classes, and exceptions. It adds accessor and modifier methods (JavaBean methods) for data members. (See Section 10.16.3.)

`java:serializable`

This directive allows you to use Ice to transmit serializable Java classes as native objects, without having to define corresponding Slice definitions for these classes. See Section 10.15 for more detail.

`java:type`

This directive allows to use custom types for sequences and dictionaries. (See Section 10.16.2.)

B.4 Metadata Directives for C#

Note that C# (and other Common Language Runtime languages) are also affected by metadata with a `clr:` prefix. (See Section B.5.)

`cs:attribute`

This directive can be used both as a global directive and as directive for specific Slice constructs. It injects C# attribute definitions into the generated code. (See Section 14.15.)

B.5 Metadata Directives for .NET and Mono

`clr:class`

This directive applies to Slice structures. It directs the code generator to emit a C# class instead of a structure. (See Section 14.7.2.)

`clr:collection`

This directive applies to sequences and dictionaries and maps them to types that are derived from `CollectionBase` and `DictionaryBase`, respectively. (See Section 14.7.3 and Section 14.7.4.)

`clr:generic:List`
`clr:generic:LinkedList`
`clr:generic:Queue`
`clr:generic:Stack`

This directives apply to sequences and map them to the specified sequence type. (See Section 14.7.3.)

`clr:generic:SortedDictionary`

This directive applies to dictionaries and maps them to `SortedDictionary`. (See Section 14.7.4.)

`clr:generic`

This directive applies to sequences and allows you map them to custom types. (See Section 14.7.3.)

`clr:property`

This directive applies to Slice structures and classes. It directs the code generator to create C# property definitions for data members. (See Section 14.7.2.)

`clr:serializable`

This directive allows you to use Ice to transmit serializable CLR classes as native objects, without having to define corresponding Slice definitions for these classes. See Section 14.14 for more detail.

B.6 Metadata Directives for Objective-C

`objc:prefix`

This directive applies to modules and changes the default mapping for modules to use a specified prefix. (See Section 18.3.)

B.7 Metadata Directives for Python

`python:package`

This global directive instructs the code generator to place the generated code into a specified Python package. (See Section 22.16.5.)

`python:seq:default`
`python:seq:list`
`python:seq:tuple`

These directives allow you change the mapping for Slice sequences. (See Section 22.7.3.)

B.8 Metadata Directives for Freeze

`freeze:read`
`freeze:write`

These directives inform a Freeze evictor whether an operation updates the state of an object, so the evictor knows whether it must save an object before evicting it. (See Section 39.3.5.)

Appendix C
Slice API Reference

See http://www.zeroc.com/doc/3.4.1/reference for the online Slice API Reference.

Appendix D
Properties

This chapter provides a reference for all properties used by the Ice run time and its services.

Unless stated otherwise in the description of an individual property, its default value is the empty string. If a property takes a numeric value, the empty string is interpreted as zero.

Note that Ice reads properties that control the run time and its services only once on start-up, when you create a communicator. This means that you must set Ice-related properties to their correct values *before* you create a communicator. If you change the value of an Ice-related property after that point, it is likely that the new setting will simply be ignored.

D.1 Ice Configuration Property

Ice.Config

Synopsis

```
--Ice.Config
--Ice.Config=1
--Ice.Config=config_file[,config_file,...]
```

Description

This property must be set from the command line with the `--Ice.Config`, `--Ice.Config=1`, or `--Ice.Config=`*`config_file`* option.

If the `Ice.Config` property is empty or set to 1, the Ice run time examines the contents of the **`ICE_CONFIG`** environment variable to retrieve the path names of one or more configuration files. Otherwise, `Ice.Config` must be set to the path names of one or more configuration files, separated by commas. (Path names can be relative or absolute.) Further property values are read from the configuration files thus specified.

In Java, Ice first attempts to open a configuration file as a class loader resource. If that attempt fails, Ice opens the configuration file in the local file system. Section 30.3 describes the semantics in more detail.

Configuration File Syntax

A configuration file contains a number of property settings, one setting per line. Property settings have one of the forms

```
property_name= # Set property to the empty string or zero
```

```
property_name=value # Assign value to property
```

The # character indicates a comment: the # character and anything following the # character on the same line are ignored. A line that has the # character as its first non-white space character is ignored in its entirety.

A configuration file is free-form: blank, tab, and newline characters serve as token delimiters and are otherwise ignored.

Any setting of the `Ice.Config` property inside the configuration file itself is ignored.

Refer to Chapter 30 for more information on configuration files.

D.2 Ice Trace Properties

Ice.Trace.GC

Synopsis

`Ice.Trace.GC=`*`num`*

Description

The garbage collector trace level:

0	No garbage collector trace (default).
1	Show the total number of instances collected, the total number of instances examined, the time spent in the collector in milliseconds, and the total number of runs of the collector.
2	Like 1, but also produces a trace message for each run of the collector.

Ice.Trace.Locator

Synopsis

`Ice.Trace.Locator=`*`num`*

Description

The locator trace level:

0	No locator trace (default).
1	Trace Ice locator and locator registry requests. The Ice run time makes locator requests to resolve the endpoints of object adapters and well-known objects. Requests on the locator registry are used to update object adapter endpoints and set the server process proxy.
2	Like 1, but also trace the removal of endpoints from the cache.

Ice.Trace.Network

Synopsis

`Ice.Trace.Network=`*`num`*

Description

The network trace level:

0	No network trace (default).

1	Trace successful connection establishment and closure.
2	Like 1, but also trace attempts to bind, connect, and disconnect sockets.
3	Like 2, but also trace data transfer, the published endpoints for an object adapter, and the current list of local addresses for an endpoint that uses the wildcard address.

Ice.Trace.Protocol

Synopsis

`Ice.Trace.Protocol=`*`num`*

Description

The protocol trace level:

0	No protocol trace (default).
1	Trace Ice protocol messages.

Ice.Trace.Retry

Synopsis

`Ice.Trace.Retry=`*`num`*

Description

The request retry trace level:

0	No request retry trace (default).
1	Trace Ice operation call retries.
2	Also trace Ice endpoint usage.

Ice.Trace.Slicing

Synopsis

`Ice.Trace.Slicing=`*`num`*

Description

The slicing trace level:

0	No trace of slicing activity (default).
1	Trace all exception and class types that are unknown to the receiver and therefore sliced.

Ice.Trace.ThreadPool

Synopsis

`Ice.Trace.ThreadPool=`*`num`*

Description

The thread pool trace level:

0	No trace of thread pool activity (default).
1	Trace the creation, growing, and shrinking of thread pools.

D.3 Ice Warning Properties

Ice.Warn.Connections

Synopsis

`Ice.Warn.Connections=`*`num`*

Description

If *num* is set to a value larger than zero, Ice applications print warning messages for certain exceptional conditions in connections. The default value is 0.

Ice.Warn.Datagrams

Synopsis

`Ice.Warn.Datagrams=`*`num`*

Description

If *num* is set to a value larger than zero, servers print a warning message if they receive a datagram that exceeds the servers' receive buffer size. (Note that this condition is not detected by all UDP implementations—some implementations silently drop received datagrams that are too large.) The default value is 0.

Ice.Warn.Dispatch

Synopsis

`Ice.Warn.Dispatch=`*`num`*

Description

If *num* is set to a value larger than zero, Ice applications print warning messages for certain exceptions that are raised while an incoming request is dispatched.

0	No warnings.
1	Print warnings for unexpected `Ice::LocalException`, `Ice::UserException`, C++ exceptions, and Java run-time exceptions (default).
2	Like 1, but also issue warnings for `Ice::ObjectNotExistException`, `Ice::FacetNotExistException`, and `Ice::OperationNotExistException`.

Ice.Warn.Endpoints

Synopsis

`Ice.Warn.Endpoints=`*`num`*

Description

If *num* is set to a value larger than zero, a warning is printed if a stringified proxy contains an endpoint that cannot be parsed. (For example, on versions of Ice that do not support SSL, stringified proxies containing SSL endpoints cause this warning.) The default value is 1.

Ice.Warn.AMICallback

Synopsis

`Ice.Warn.AMICallback=`*num*

Description

If *num* is set to a value larger than zero, warnings are printed if an AMI callback raises an exception. The default value is 1.

Ice.Warn.UnknownProperties

Synopsis

`Ice.Warn.UnknownProperties=`*num*

Description

If *num* is set to a value larger than zero, the Ice run time prints a warning about unknown properties for object adapters (see Section D.4) and proxies (see Section D.9). The default value is 1.

Ice.Warn.UnusedProperties

Synopsis

`Ice.Warn.UnusedProperties=`*num*

Description

If *num* is set to a value larger than zero, the Ice run time prints a warning about properties that were set but not read. The warning is emitted when a communicator is destroyed; it is useful to detect mis-spelled properties, such as `Filesystem.MaxFilSize`. The default value is 0.

D.4 Ice Object Adapter Properties

*adapter.*ACM

Synopsis

adapter.ACM=*num*

Description

If *num* is set to a value larger than zero, the adapter automatically closes an incoming connection after it has been idle for *num* seconds. The default value is the setting of `Ice.ACM.Server`.

ACM can cause incoming oneway requests to be silently discarded. See Section 36.4 for more information.

*adapter.*AdapterId

Synopsis

adapter.AdapterId=*id*

Description

Specifies an identifier for the object adapter with the name *adapter.* This identifier must be unique among all object adapters using the same locator instance. If a locator proxy is defined using *adapter*.`Locator` or `Ice.Default.Locator`, this object adapter sets its endpoints with the locator registry upon activation.

*adapter.*Endpoints

Synopsis

adapter.Endpoints=*endpoints*

Description

Sets the endpoints for the object adapter *adapter* to *endpoints*. These endpoints specify the network interfaces on which the object adapter receives requests. Proxies created by the object adapter contain these endpoints, unless the *adapter*.`PublishedEndpoints` property is also specified.

*adapter.*Locator

Synopsis

adapter.`Locator=`*locator*

Description

Specifies a locator for the object adapter with the name *adapter.* The value is a stringified proxy to the Ice locator interface.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

*adapter.*ProxyOptions

Synopsis

adapter.`ProxyOptions=`*options*

Description

Specifies the proxy options for proxies created by the object adapter. The value is a string representing the proxy options as they would be specified in a stringified proxy. See Appendix E for more information on proxy options. The default value is `"-t"`, that is, proxies created by the object adapter are configured to use twoway invocations by default.

*adapter.*PublishedEndpoints

Synopsis

adapter.`PublishedEndpoints=`*endpoints*

Description

When creating a proxy, the object adapter *adapter* normally includes the endpoints defined by *adapter.*`Endpoints`. If *adapter.*`PublishedEndpoints` is defined, the object adapter uses these endpoints instead. This is useful in many situations, such as when a server resides behind a port-forwarding firewall, in which case the object adapter's public endpoints must specify the address and port of the firewall. The *adapter.*`ProxyOptions` property also influences the proxies created by an object adapter.

*adapter.*ReplicaGroupId

Synopsis

adapter.ReplicaGroupId=*id*

Description

Identifies the group of replicated object adapters to which this adapter belongs. The replica group is treated as a virtual object adapter, so that an indirect proxy of the form identity@*id* refers to the object adapters in the group. During binding, a client will attempt to establish a connection to an endpoint of one of the participating object adapters, and automatically try others until a connection is successfully established or all attempts have failed. Similarly, an outstanding request will, when permitted, automatically fail over to another object adapter of the replica group upon connection failure. The set of endpoints actually used by the client during binding is determined by the locator's configuration policies.

Defining a value for this property has no effect unless *adapter*.AdapterId is also defined. Furthermore, the locator registry may require replica groups to be defined in advance (see IceGrid.Registry.DynamicRegistration), otherwise Ice.NotRegisteredException is raised upon adapter activation. Regardless of whether an object adapter is replicated, it can always be addressed individually in an indirect proxy if it defines a value for *adapter*.AdapterId.

*adapter.*Router

Synopsis

adapter.Router=*router*

Description

Specifies a router for the object adapter with the name *adapter*. The value is a stringified proxy to the Ice router control interface. Defining a router allows the object adapter to receive callbacks from the router over outgoing connections from this process to the router, thereby avoiding the need for the router to establish a connection back to the object adapter.

A router can only be assigned to one object adapter. Specifying the same router for more than one object adapter results in undefined behavior. The default value is no router.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

*adapter.*ThreadPool.Serialize

Synopsis

adapter.ThreadPool.Serialize=*num*

Description

If *num* is a value greater than zero, the adapter's thread pool serializes all messages from each connection. It is not necessary to enable this feature in a thread pool whose maximum size is one thread. In a multi-threaded pool, enabling serialization allows requests from different connections to be dispatched concurrently while preserving the order of messages on each connection. Note that serialization has a signficant impact on latency and throughput. See Section 32.10 for more information on thread pools. If not defined, the default value is zero.

*adapter.*ThreadPool.Size

Synopsis

adapter.ThreadPool.Size=*num*

Description

A communicator creates a default server thread pool that dispatches requests to its object adapters. An object adapter can also be configured with its own thread pool. This is useful in avoiding deadlocks due to thread starvation by ensuring that a minimum number of threads is available for dispatching requests to certain Ice objects. Section 32.10 describes thread pools in greater detail.

num is the initial number of threads in the thread pool. The default value is zero, meaning that an object adapter by default uses the communicator's server thread pool. See `Ice.ThreadPool.Server.Size` for more information.

*adapter.*ThreadPool.SizeMax

Synopsis

adapter.ThreadPool.SizeMax=*num*

Description

num is the maximum number of threads for the thread pool. See `Ice.ThreadPool.Server.SizeMax` for more information.

The default value is the value of *adapter*`.ThreadPool.Size`, meaning the thread pool can never grow larger than its initial size.

*adapter.*ThreadPool.SizeWarn

Synopsis

adapter`.ThreadPool.SizeWarn=`*num*

Description

Whenever *num* threads are active in a thread pool, a “low on threads” warning is printed. The default value is zero, which disables the warning.

*adapter.*ThreadPool.StackSize

Synopsis

adapter`.ThreadPool.StackSize=`*num*

Description

num is the stack size (in bytes) of threads in the thread pool. The default value is zero, meaning the operating system's default is used.

*adapter.*ThreadPool.ThreadIdleTime

Synopsis

adapter`.ThreadPool.ThreadIdleTime=`*num*

Description

In a dynamically-sized thread pool, Ice reaps a thread after it is idle for *num* seconds. Setting this property to zero disables idle thread reaping. If not specified, the default value is 60 seconds. See `Ice.ThreadPool.Server.ThreadIdleTime` for more information.

*adapter.*ThreadPool.ThreadPriority

Synopsis

adapter.ThreadPool.ThreadPriority=*num*

Description

num specifes a thread priority. The object adapter creates its threads with the specified priority. Leaving this property unset causes the adapter to create threads with the priority specified by `Ice.ThreadPool.Server.ThreadPriority` or, if that property is unset, the priority specified by `Ice.ThreadPriority`.

D.5 Ice Administrative Properties

Ice.Admin.*name*

Synopsis

Ice.Admin.*name*=*value*

Description

The Ice run time creates an internal object adapter named `Ice.Admin` if `Ice.Admin.Endpoints` is defined and one of the following are true:

- `Ice.Admin.InstanceName` is defined
- `Ice.Admin.ServerId` and `Ice.Default.Locator` are defined

The purpose of this object adapter is to host an Ice object whose facets provide administrative capabilities to remote clients. All of the adapter properties detailed in Section D.4 can be used to configure the `Ice.Admin` object adapter.

Note that enabling the `Ice.Admin` object adapter is a security risk because a hostile client could use the administrative object to shut down the process. As a result, the endpoints for this object adapter should be carefully defined so that only trusted clients are allowed to use it.

See Section 32.18 for more information on the administrative object.

Ice.Admin.DelayCreation

Synopsis

`Ice.Admin.DelayCreation=`*`num`*

Description

If *num* is a value greater than zero, the Ice run time delays the creation of the administrative object adapter until `getAdmin` is invoked on the communicator. If not specified, the default value is zero, meaning the object adapter is created immediately after all plug-ins are initialized. See Section 32.18.2 for more information on the administrative object adapter.

Ice.Admin.Facets

Synopsis

`Ice.Admin.Facets=`*`name`* `[`*`name`* `...]`

Description

Specifies the facets enabled by the administrative object. See Section 32.18.6 for a discussion of the facets that the administrative object enables by default. Facet names are delimited by commas or white space. A facet name that contains white space must be enclosed in single or double quotes. If not specified, all facets are enabled.

Ice.Admin.InstanceName

Synopsis

`Ice.Admin.InstanceName=`*`name`*

Description

Specifies an identity category for the administrative object (see Section 32.18). If defined, the identity of the object becomes *`name`*`/admin`. If not specified, the default identity category is a UUID.

Ice.Admin.ServerId

Synopsis

`Ice.Admin.ServerId=`*`id`*

Description

Specifies an identifier that uniquely identifies the process when the `Ice.Admin` object adapter registers with the locator registry. See Section 38.22 for more information.

D.6 Ice Plug-In Properties

Ice.Plugin.*name*.cpp

Synopsis

`Ice.Plugin.`*`name`*`.cpp=`*`basename`*`[,`*`version`*`]:`*`function`* `[`*`args`*`]`

Description

Defines a C++ plug-in to be installed during communicator initialization. The *`basename`* and optional *`version`* components are used to construct the name of a DLL or shared library. If no version is supplied, the Ice version is used. The *`function`* component is the name of a function with C linkage. For example, the entry point `MyPlugin,34:create` would imply a shared library name of `libMyPlugin.so.34` on Unix and `MyPlugin34.dll` on Windows. Furthermore, if Ice is built on Windows with debugging, a `d` is automatically appended to the version (for example, `MyPlugin34d.dll`).

The function must be declared with external linkage and have the following signature:

```
<Plugin>* function(const Ice::CommunicatorPtr& communicator,
                   const std::string& name,
                   const Ice::StringSeq& args);
```

Note that the function must return a pointer and not a smart pointer. The Ice core deallocates the object when it unloads the library.

Any arguments that follow the entry point are passed to the `create` method. For example:

```
Ice.Plugin.MyPlugin=MyFactory,34:create arg1 arg2
```

Ice.Plugin.*name*.java

Synopsis

`Ice.Plugin.`*`name`*`.java=`*`class`* `[`*`args`*`]`

Description

Defines a Java plug-in to be installed during communicator initialization. The specified class must implement the `Ice.PluginFactory` interface. Any arguments that follow the class name are passed to the `create` method. For example:

```
Ice.Plugin.MyPlugin=MyFactory arg1 arg2
```

Ice.Plugin.*name*.clr

Synopsis

`Ice.Plugin.`*`name`*`.clr=`*`assembly`*`:`*`class`* `[`*`args`*`]`

Description

Defines a .NET plug-in to be installed during communicator initialization. The assembly can be the full assembly name, such as `myplugin, Version=0.0.0.0, Culture=neutral`, or an assembly DLL name such as `myplugin.dll`. The specified class must implement the `Ice.PluginFactory` interface. Any arguments that follow the class name are passed to the `create` method. For example:

```
Ice.Plugin.MyPlugin=MyFactory, Version=1.2.3.4, Culture=neutral:My
Factory arg1 arg2
```

Ice.Plugin.*name*

Synopsis

`Ice.Plugin.`*`name`*`=`*`entry_point`* `[`*`args`*`]`

Description

Defines a plug-in to be installed during communicator initialization. The format of *entry_point* varies by Ice implementation language, therefore this property cannot be defined in a configuration file that is shared by programs in different languages. Ice provides an alternate syntax that facilitates such sharing:

- `Ice.Plugin.`*name*`.cpp` for C++
- `Ice.Plugin.`*name*`.java` for Java
- `Ice.Plugin.`*name*`.clr` for the .NET Common Language Runtime

Refer to the description of each of these properties for more information on the expected entry point syntax.

Ice.PluginLoadOrder

Synopsis

```
Ice.PluginLoadOrder=names
```

Description

Determines the order in which plug-ins are loaded. The Ice run time loads the plug-ins in the order they appear in *names*, where each plug-in name is separated by a comma or white space. Any plug-ins not mentioned in *names* are loaded afterward, in an undefined order.

Ice.InitPlugins

Synopsis

```
Ice.InitPlugins=num
```

Description

If *num* is a value greater than zero, the Ice run time automatically initializes the plug-ins it has loaded. The order in which plug-ins are loaded and initialized is determined by `Ice.PluginLoadOrder`. An application may need to set this property to zero in order to interact directly with a plug-in after it has been loaded but before it is initialized. In this case, the application must invoke `initializePlugins` on the plug-in manager to complete the initialization process. If not defined, the default value is 1.

D.7 Ice Thread Pool Properties

Ice.ThreadPool.Client.Serialize
Ice.ThreadPool.Server.Serialize

Synopsis

```
Ice.ThreadPool.Client.Serialize=num
Ice.ThreadPool.Server.Serialize=num
```

Description

If *num* is a value greater than zero, the thread pool serializes all messages from each connection. It is not necessary to enable this feature in a thread pool whose maximum size is one thread. In a multi-threaded pool, enabling serialization allows requests from different connections to be dispatched concurrently while preserving the order of messages on each connection. Note that serialization has a signficant impact on latency and throughput. See Section 32.10 for more information on thread pools. If not defined, the default value is zero.

Ice.ThreadPool.Client.Size
Ice.ThreadPool.Server.Size

Synopsis

```
Ice.ThreadPool.Client.Size=num
Ice.ThreadPool.Server.Size=num
```

Description

A communicator creates two thread pools: the client thread pool dispatches AMI callbacks and incoming requests on bidirectional connections, and the server thread pool dispatches requests to object adapters.

Thread pools in Ice can grow and shrink dynamically, based on an average load factor. A thread pool always has at least one thread and may grow as load increases up to the maximum size specified by `SizeMax`. If `SizeMax` is not specified, Ice uses the value of `Size` as the pool's maximum size. A thread pool is initialized with *num* active threads, but the pool may shrink to only one thread during idle periods as determined by `ThreadIdleTime`.

If not specified, the default value is one for both properties.

An object adapter can also be configured with its own thread pool. See the object adapter properties for more information.

Note that multiple threads for the client thread pool are only required for nested AMI invocations, or to allow multiple AMI callbacks to be dispatched concurrently.

To monitor the thread pool activities of the Ice run time, enable the `Ice.Trace.ThreadPool` property.

Ice.ThreadPool.Client.SizeMax
Ice.ThreadPool.Server.SizeMax

Synopsis

```
Ice.ThreadPool.Client.SizeMax=num
Ice.ThreadPool.Server.SizeMax=num
```

Description

num is the maximum number of threads for the thread pool. Refer to the `Size` property for more information on configuring the size of a thread pool.

The default value for `SizeMax` is the value of `Size`, meaning the thread pool can never grow larger than its initial size.

To monitor the thread pool activities of the Ice run time, enable the `Ice.Trace.ThreadPool` property.

Ice.ThreadPool.Client.SizeWarn
Ice.ThreadPool.Server.SizeWarn

Synopsis

```
Ice.ThreadPool.Client.SizeWarn=num
Ice.ThreadPool.Server.SizeWarn=num
```

Description

Whenever *num* threads are active in a thread pool, a “low on threads” warning is printed. The default value is zero, which disables the warning.

To monitor the thread pool activities of the Ice run time, enable the `Ice.Trace.ThreadPool` property.

Ice.ThreadPool.Client.StackSize
Ice.ThreadPool.Server.StackSize

Synopsis

```
Ice.ThreadPool.Client.StackSize=num
Ice.ThreadPool.Server.StackSize=num
```

Description

num is the stack size (in bytes) of threads in the thread pool. The default value is zero meaning the operating system's default is used.

Ice.ThreadPool.Client.ThreadIdleTime
Ice.ThreadPool.Server.ThreadIdleTime

Synopsis

```
Ice.ThreadPool.Client.ThreadIdleTime=num
Ice.ThreadPool.Server.ThreadIdleTime=num
```

Description

Ice can automatically reap idle threads in a thread pool to conserve resources. The `ThreadIdleTime` property specifies the number of seconds a thread must be idle before it is reaped. If not specified, the default value is 60 seconds.

To disable the reaping of idle threads, set `ThreadIdleTime` to zero. In this situation, the thread pool is initialized with `Size` active threads and may grow to contain `SizeMax` active threads, but the size of the pool never decreases.

To monitor the thread pool activities of the Ice run time, enable the `Ice.Trace.ThreadPool` property.

Ice.ThreadPool.Client.ThreadPriority
Ice.ThreadPool.Server.ThreadPriority

Synopsis

```
Ice.ThreadPool.Client.ThreadPriority=num
Ice.ThreadPool.Server.ThreadPriority=num
```

Description

num specifes a thread priority. Threads created by the Ice run time are by default created with the specified priority for the client and server thread pool, respectively. Leaving these properties unset causes the run time to create threads with the default priority specified by `Ice.ThreadPriority`.

These properties are unset by default.

You can override the default priority for a specific object adapter (see `adapter.ThreadPool.Priority`).

Ice.ThreadPriority

Synopsis

`Ice.ThreadPriority=`*num*

Description

num specifes a thread priority. Threads created by the Ice run time are by default created with the specified priority. Leaving this property unset causes the run time to create threads with the system default priority. This property is unset by default. You can separately override the default priorities for the client and server thread pool (see `Ice.ThreadPool.Client.ThreadPriority` and `Ice.ThreadPool.Server.ThreadPriority`) as well as for a specific object adapter (see `adapter.ThreadPool.ThreadPriority`).

D.8 Ice Default and Override Properties

Ice.Default.CollocationOptimized

Synopsis

`Ice.Default.CollocationOptimized=`*num*

Description

Specifies whether proxy invocations use collocation optimization (see Section 32.21) by default. When enabled, proxy invocations on a collocated servant (i.e., a servant whose object adapter was created by the same communicator as the proxy) are made as a direct method call if possible. Collocated invoca-

tions are more efficient because they avoid the overhead of marshaling parameters and sending requests over the network.

Collocation optimization is not supported for asynchronous or Dynamic Ice invocations, nor is it supported in Ice for Python and Ice for Ruby.

If not specified, the default value is 1. Set the property to 0 to disable collocation optimization by default.

Ice.Default.EndpointSelection

Synopsis

`Ice.Default.EndpointSelection=`*`policy`*

Description

This property controls the default endpoint selection policy for proxies with multiple endpoints. Permissible values are `Ordered` and `Random`. The default value of this property is `Random`.

Ice.Default.Host

Synopsis

`Ice.Default.Host=`*`host`*

Description

If an endpoint is specified without a host name (i.e., without a `-h host` option), the *`host`* value from this property is used instead. The property has no default value.

Ice.Default.Locator

Synopsis

`Ice.Default.Locator=`*`locator`*

Description

Specifies a default locator for all proxies and object adapters. The value is a stringified proxy to the IceGrid locator interface. The default locator can be overridden on a proxy using the `ice_locator` operation. The default value is no locator.

The default identity of the IceGrid locator object is `IceGrid/Locator` (see `IceGrid.InstanceName`). It is listening on the IceGrid client endpoints. For example, if `IceGrid.Registry.Client.Endpoints` is set to `tcp -p 12000 -h localhost`, the stringified proxy for the IceGrid locator is `IceGrid/Locator:tcp -p 12000 -h localhost`.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

Ice.Default.LocatorCacheTimeout

Synopsis

`Ice.Default.LocatorCacheTimeout=`*`num`*

Description

Specifies the default locator cache timeout for indirect proxies. If *`num`* is set to a value larger than zero, locator cache entries older than *`num`* seconds will be ignored. If set to 0, the locator cache won't be used. If set to `-1`, locator cache entries won't expire.

Once a cache entry has expired, the Ice run time performs a new locate request to refresh the cache before sending the next invocation; therefore, the invocation is delayed until the run time has refreshed the entry. If you set `Ice.BackgroundLocatorCacheUpdates` to a non-zero value, the lookup to refresh the cache is still performed but happens in the background; this avoids the delay for the first invocation that follows expiry of a cache entry.

Ice.Default.Package

Synopsis

`Ice.Default.Package=`*`package`*

Description

Specifies a default package to use if other attempts by the Ice run time to dynamically load a generated class have failed. If global metadata is used to enclose generated Java classes in a user-defined package, the Ice run time must be configured in order to successfully unmarshal exceptions and concrete class types. See also `Ice.Package.`*`module`*.

Ice.Default.PreferSecure

Synopsis

`Ice.Default.PreferSecure=`*`num`*

Description

Specifies whether secure endpoints are given precedence in proxies by default. The default value of *num* is zero, meaning that insecure endpoints are given preference.

Setting this property to a non-zero value is the equivalent of invoking `ice_preferSecure(true)` on proxies created by the Ice run time, such as those returned by `stringToProxy` or received as the result of an invocation. Proxies created by proxy factory methods such as `ice_oneway` inherit the setting of the original proxy. If you want to force all proxies to use only secure endpoints, use `Ice.Override.Secure` instead.

See Section 36.3.1 for more information on endpoint selection and Section 41.4.7 for a discussion of secure proxies.

Ice.Default.Protocol

Synopsis

`Ice.Default.Protocol=protocol`

Description

Sets the protocol that is being used if an endpoint uses default as the protocol specification. The default value is `tcp`.

Ice.Default.Router

Synopsis

`Ice.Default.Router=`*`router`*

Description

Specifies the default router for all proxies. The value is a stringified proxy to the Glacier2 router control interface. The default router can be overridden on a proxy using the `ice_router` operation. The default value is no router.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

Ice.Override.Compress

Synopsis

`Ice.Override.Compress=`*`num`*

Description

If set, this property overrides compression settings in all proxies. If *num* is set to a value larger than zero, compression is enabled. If zero, compression is disabled.

The setting of this property is ignored in the server role.

Note that, if a client sets `Ice.Override.Compress=1` and sends a compressed request to a server that does not support compression, the server will close the connection and the client will receive `ConnectionLostException`.

If a client does not support compression and `Ice.Override.Compress=1`, the setting is ignored and a warning message is printed on `stderr`.

Regardless of the setting of this property, requests smaller than 100 bytes are never compressed.

Ice.Override.ConnectTimeout

Synopsis

`Ice.Override.ConnectTimeout=`*`num`*

Description

This property overrides timeout settings used to establish connections. *num* is the timeout value in milliseconds, or `-1` for no timeout. If this property is not set, then `Ice.Override.Timeout` is used.

Ice.Override.Secure

Synopsis

`Ice.Override.Secure=`*num*

Description

If set to a value larger than zero, this property overrides security settings in all proxies by allowing only secure endpoints. Defining this property is equivalent to invoking `ice_secure(true)` on every proxy. If you wish to give priority to secure endpoints without precluding the use of non-secure endpoints, see `Ice.Default.PreferSecure`. Refer to Section 41.4.7 for more information on secure proxies.

Ice.Override.Timeout

Synopsis

`Ice.Override.Timeout=`*num*

Description

If set, this property overrides timeout settings in all endpoints. *num* is the timeout value in milliseconds, or `-1` for no timeout.

D.9 Ice Proxy Properties

The communicator operation `propertyToProxy` creates a proxy from a group of configuration properties (see Section 32.2). The argument to `propertyToProxy` is a string representing the base name of the property group. This name must correspond to a property that supplies the stringified form of the proxy. Subordinate properties can be defined to customize the proxy's local configuration.

The communicator operation `proxyToProperty` performs the inverse operation, that is, returns the property group for a proxy.

name

Synopsis

name=*proxy*

Description

A property with an application-specific *name* supplies the stringified representation of a proxy. The application uses the communicator operation `propertyToProxy` to retrieve the property and convert it into a proxy (see Section 32.11.1).

name.CollocationOptimized

Synopsis

name.CollocationOptimized=*num*

Description

If *num* is a value greater than zero, the proxy is configured to use collocation invocations (see Section 32.21) when possible. Defining this property is equivalent to invoking the `ice_collocationOptimized` factory method, which is described in Section 32.11.2.

name.ConnectionCached

Synopsis

name.ConnectionCached=*num*

Description

If *num* is a value greater than zero, the proxy caches its chosen connection for use in subsequent requests. Defining this property is equivalent to invoking the `ice_connectionCached` factory method (see Section 32.11.2).

name.EndpointSelection

Synopsis

name.EndpointSelection=*type*

Description

Specifies the proxy's endpoint selection type. Legal values are `Random` and `Ordered`. Defining this property is equivalent to invoking the `ice_endpointSelection` factory method (see Section 32.11.2).

name.Locator

Synopsis

name.Locator=*proxy*

Description

Specifies the proxy of the locator to be used by this proxy. Defining this property is equivalent to invoking the `ice_locator` factory method (see Section 32.11.2).

This is a proxy property, so you can configure additional local aspects of the proxy with subordinate properties.

name.LocatorCacheTimeout

Synopsis

name.LocatorCacheTimeout=*num*

Description

Specifies the locator cache timeout to be used by this proxy. Defining this property is equivalent to invoking the `ice_locatorCacheTimeout` factory method (see Section 32.11.2).

name.PreferSecure

Synopsis

name.PreferSecure=*num*

Description

If *num* is a value greater than zero, the proxy gives precedence to secure endpoints. If not defined, the proxy uses the value of `Ice.Default.PreferSecure`.

Defining this property is equivalent to invoking the `ice_preferSecure` factory method (see Section 32.11.2).

name.Router

Synopsis

name.Router=*proxy*

Description

Specifies the proxy of the router to be used by this proxy. Defining this property is equivalent to invoking the `ice_router` factory method (see Section 32.11.2).

This is a proxy property, so you can configure additional local aspects of the proxy with subordinate properties.

D.10 Ice Transport Properties

Ice.IPv4

Synopsis

Ice.IPv4=*num*

Description

Specifies whether Ice uses IPv4. If *num* is a value greater than zero, IPv4 is enabled. If not specified, the default value is `1`.

Ice.IPv6

Synopsis

`Ice.IPv6=`*`num`*

Description

Specifies whether Ice uses IPv6. If *num* is a value greater than zero, IPv6 is enabled. If not specified, the default value is zero.

Ice.TCP.Backlog

Synopsis

`Ice.TCP.Backlog=`*`num`*

Description

Specifies the size of the listen queue for each TCP or SSL server endpoint. If not defined, the default value for C++ programs uses the value of `SOMAXCONN` if present, or `511` otherwise. In Java and .NET, the default value is 511.

Ice.TCP.RcvSize
Ice.TCP.SndSize

Synopsis

`Ice.TCP.RcvSize=`*`num`*
`Ice.TCP.SndSize=`*`num`*

Description

These properties set the TCP receive and send buffer sizes to the specified value in bytes. The default value depends on the configuration of the local TCP stack. (A common default values is 65535 bytes.)

The OS may impose lower and upper limits on the receive and send buffer sizes or otherwise adjust the buffer sizes. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

Ice.UDP.RcvSize
Ice.UDP.SndSize

Synopsis

`Ice.UDP.RcvSize=`*`num`*
`Ice.UDP.SndSize=`*`num`*

Description

These properties set the UDP receive and send buffer sizes to the specified value in bytes. Ice messages larger than *`num`* `- 28` bytes cause a `DatagramLimitException`. The default value depends on the configuration of the local UDP stack. (Common default values are 65535 and 8192 bytes.)

The OS may impose lower and upper limits on the receive and send buffer sizes or otherwise adjust the buffer sizes. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

Settings of these properties less than 28 are ignored.

Note that, on many operating systems, it is possible to set buffer sizes greater than 65535. Such settings do not change the hard limit of 65507 bytes for the payload of a UDP packet, but merely affect how much data can be buffered by the kernel.

Settings less than 65535 limit the size of Ice datagrams as well as adjust the kernel buffer sizes.

D.11 Ice Miscellaneous Properties

Ice.ACM.Client

Synopsis

`Ice.ACM.Client=`*`num`*

Description

If *`num`* is set to a value larger than zero, client-side Active Connection Management (ACM) is enabled. This means that connections are automatically closed by

the client after they have been idle for *num* seconds. This is transparent to applications because connections closed by ACM are automatically reestablished if they are needed again. The default value is `60`, meaning that idle connections are automatically closed after one minute.

Ice.ACM.Server

Synopsis

`Ice.ACM.Server=`*num*

Description

This property is the server-side equivalent of `Ice.ACM.Client`. If *num* is set to a value larger than zero, server-side Active Connection Management (ACM) is enabled, in which the server automatically closes an incoming connection after it has been idle for *num* seconds. The default value is `0`, meaning that server-side ACM is disabled by default.

Server-side ACM can cause incoming oneway requests to be silently discarded. See Section 36.4 for more information.

Ice.BackgroundLocatorCacheUpdates

Synopsis

`Ice.BackgroundLocatorCacheUpdates=`*num*

Description

If *num* is set to zero (the default), an invocation on an indirect proxy whose endpoints are older than the configured locator cache timeout triggers a locator cache update; the run time delays the invocation until the new endpoints are returned by the locator.

If *num* is set to a value larger than zero, an invocation on an indirect proxy with expired endpoints still triggers a locator cache update, but the update is performed in the background, and the run time uses the expired enpoints for the invocation. This avoids delaying the first invocation that follows expiry of a cache entry.

Ice.BatchAutoFlush

Synopsis

`Ice.BatchAutoFlush=`*`num`*

Description

This property controls how the Ice run time deals with flushing of batch messages. If *num* is set to a non-zero value (the default), the run time automatically forces a flush of the current batch when a new message is added to a batch and that message would cause the batch to exceed `Ice.MessageSizeMax`. If *num* is set to zero, batches must be flushed explicitly by the application; in this case, if the application adds more messages to a batch than permitted by `Ice.MessageSizeMax`, the application receives a `MemoryLimitException` when it flushes the batch.

Ice.CacheMessageBuffers

Synopsis

`Ice.CacheMessageBuffers=`*`num`* (Java, .NET)

Description

If *num* is a value greater than zero, the Ice run time caches message buffers for future reuse. This can improve performance and reduce the amount of garbage produced by Ice internals that the garbage collector would eventually spend time to reclaim. However, for applications that exchange very large messages, this cache may consume excessive amounts of memory and therefore should be disabled by setting this property to zero.

Platform Notes

Java

Ice allocates non-direct message buffers when this property is set to 1 and direct message buffers when set to 2. Use of direct message buffers minimizes copying and typically results in improved throughput. If not defined, the default value is 2.

.NET

If not defined, the default value is 1.

Ice.ChangeUser

Synopsis

`Ice.ChangeUser=`*`user`*

Description

If set, Ice changes the user and group id to the respective ids of *`user`* in `/etc/passwd`. This only works if the Ice application is executed by the super-user (Unix only).

Ice.Compression.Level

Synopsis

`Ice.Compression.Level=`*`num`*

Description

Specifies the bzip2 compression level. Legal values for *`num`* are `1` to `9`, where `1` represents the fastest compression and `9` represents the best compression. Note that higher levels cause the bzip2 algorithm to devote more resources to the compression effort, and may not result in a significant improvement over lower levels. If not specified, the default value is `1`.

Ice.ConsoleListener

Synopsis

`Ice.ConsoleListener=`*`num`*

Description

This property applies only to Ice for .NET. If *`num`* is non-zero, the Ice run time installs a `ConsoleTraceListener` that writes its messages to `stderr`. If *`num`* is zero, logging is disabled. Note that the setting of `Ice.LogFile` overrides this property: if `Ice.LogFile` is set, messages are written to the log file regardless of the setting of `Ice.ConsoleListener`.

Ice.EventLog.Source

Synopsis

`Ice.EventLog.Source=`*`name`*

Description

Specifies the name of an event log source to be used by a Windows service that subclasses `Ice::Service` (see Section 8.3.2). The value of *`name`* represents a subkey of the `Eventlog` registry key. An application (or administrator) typically prepares the registry key when the service is installed. If no matching registry key is found, Windows logs events in the `Application` log. Any backslashes in *`name`* are silently converted to forward slashes. If not defined, `Ice::Service` uses the service name as specified by the **`--service`** option.

Ice.GC.Interval

Synopsis

`Ice.GC.Interval=`*`num`*

Description

This property determines the frequency with which the class garbage collector runs. If the interval is set to zero (the default), no collector thread is created. Otherwise, the collector thread runs every *`num`* seconds.

Ice.ImplicitContext

Synopsis

`Ice.ImplicitContext=`*`type`*

Description

Specifies whether a communicator has an implicit context and, if so, at what scope the context applies. Legal values for this property are `None` (equivalent to the empty string), `PerThread`, and `Shared`. If not specified, the default value is `None`. See Section 32.12.4 for more information on implicit contexts.

Ice.LogFile

Synopsis

`Ice.LogFile=`*`file`*

Description

Replaces the communicator's default logger (see Section 32.19.1) with a simple file-based logger implementation. This property does not affect the per-process logger (see Section 32.19.5). The logger creates the specified file if necessary, otherwise it appends to the file. If the logger is unable to open the file, the application receives an `InitializationException` during communicator initialization. If a logger object is supplied in the `InitializationData` argument during communicator initialization, it takes precedence over this property.

The logger does not provide any built-in support for log file maintenance (such as log rotation), but it can coexist with system tools such as **`logrotate`**.

Ice.MessageSizeMax

Synopsis

`Ice.MessageSizeMax=`*`num`*

Description

This property controls the maximum size (in kilobytes) of an uncompressed protocol message that is accepted or sent by the Ice run time. The size includes the size of the Ice protocol header. The default size is `1024` (`1` Megabyte). Settings with a value less than `1` are ignored.

If a client sends a message that exceeds the client's `Ice.MessageSizeMax`, or the server returns a reply that exceeds the client's `Ice.MessageSizeMax`, the client receives a `MemoryLimitException`.

If a client sends a message that exceeds the server's `Ice.MessageSizeMax`, the server immediately closes its connection, so the client receives a `ConnectionLostException` in that case. In addition, the server logs a `MemoryLimitException` if `Ice.Warn.Connections` is set.

If the server returns a reply that exceeds the server's `Ice.MessageSizeMax`, the server logs a `MemoryLimitException` (if `Ice.Warn.Connections` is set) but does not close its connection to the client. The client receives an `UnknownLocalException` in this case.

Ice.MonitorConnections

Synopsis

`Ice.MonitorConnections=`*`num`*

Description

If active connection management (ACM) is enabled (see Section 36.4), the Ice run time scans for idle connections to be closed once every *num* seconds. If you to not set this property or set *num* to zero or a negative value, the run time applies a heuristic to determine how often to scan for idle connections: the default scanning interval is the 10% of smallest configured ACM timeout, with a minimum of 5 seconds, and a maximum of 5 minutes.

Ice.Nohup

Synopsis

`Ice.Nohup=`*`num`* (C++)

Description

If *num* is set to a value larger than zero, the C++ classes `Ice::Application` and `Ice::Service` ignore `SIGHUP` (for Unix) and `CTRL_LOGOFF_EVENT` (for Windows). As a result, an application that sets `Ice.Nohup` continues to run if the user that started the application logs off. The default value for `Ice::Application` is `0`, and the default value for `Ice::Service` is `1`.

Ice.NullHandleAbort

Synopsis

`Ice.NullHandleAbort=`*`num`* (C++)

Description

If *num* is set to a value larger than zero, invoking an operation using a null smart pointer causes the program to abort immediately instead of raising `IceUtil::NullHandleException`.

Ice.Package.*module*

Synopsis

`Ice.Package.`*`module`*`=`*`package`* (Java)

Description

Associates a top-level Slice *`module`* with a Java *`package`*. If global metadata is used to enclose generated Java classes in a user-defined package, the Ice run time must be configured in order to successfully unmarshal exceptions and concrete class types. If all top-level modules are generated into the same user-defined package, it is easier to use `Ice.Default.Package` instead.

Ice.PrintAdapterReady

Synopsis

`Ice.PrintAdapterReady=`*`num`*

Description

If *`num`* is set to a value larger than zero, an object adapter prints "*adapter_name* ready" on standard output after initialization is complete. This is useful for scripts that need to wait until an object adapter is ready to be used.

Ice.PrintProcessId

Synopsis

`Ice.PrintProcessId=`*`num`*

Description

If *`num`* is set to a value larger than zero, the process ID is printed on standard output upon startup.

Ice.PrintStackTraces

Synopsis

`Ice.PrintStackTraces=`*`num`* (C++)

Description

If *num* is set to a value larger than zero, inserting an exception that derives from `IceUtil::Exception` into a logger helper class (such as `Ice::Warning`) also displays the exception's stack trace. If not defined, the default value depends on how the Ice run time is compiled: 0 for an optimized build and 1 for a debug build. Stack traces are only available when Ice is compiled with GCC; this property has no effect for other compilers.

Ice.ProgramName

Synopsis

`Ice.ProgramName=`*`name`*

Description

name is the program name, which is set automatically from `argv[0]` (C++) and from `AppDomain.CurrentDomain.FriendlyName` (.NET) during initialization. (For Java, `Ice.ProgramName` is initialized to the empty string.) For all languages, the default name can be overridden by setting this property.

Ice.RetryIntervals

Synopsis

`Ice.RetryIntervals=`*`num`* `[`*`num ...`*`]`

Description

This property defines the number of times an operation is retried and the delay between each retry. For example, if the property is set to `0 100 500`, the operation is retried 3 times: immediately after the first failure, again after waiting 100ms after the second failure, and again after waiting 500ms after the third failure. The default value (`0`) is to retry once immediately. If set to `-1`, no retry occurs.

Ice.ServerIdleTime

Synopsis

`Ice.ServerIdleTime=`*`num`*

Description

If *num* is set to a value larger than zero, Ice automatically calls `Communicator::shutdown` once the communicator has been idle for *num* seconds. This shuts down the Communicator's server side and causes all threads waiting in `Communicator::waitForShutdown` to return. After that, a server will typically do some clean-up work before exiting. The default value is zero, meaning that the server will not shut down automatically.

Ice.StdErr

Synopsis

`Ice.StdErr=`*`filename`*

Description

If *`filename`* is not empty, the standard error stream of this process is redirected to this file, in append mode. This property is checked only for the first communicator that is created in a process.

Ice.StdOut

Synopsis

`Ice.StdOut=`*`filename`*

Description

If *`filename`* is not empty, the standard output stream of this process is redirected to this file, in append mode. This property is checked only for the first communicator created in a process.

Ice.SyslogFacility

Synopsis

`Ice.SyslogFacility=`*`string`*

Description

This property sets the syslog facility to *string* (Unix only). This property has no effect if `Ice.UseSyslog` is not set.

string can be any of syslog facilities: `LOG_AUTH`, `LOG_AUTHPRIV`, `LOG_CRON`, `LOG_DAEMON`, `LOG_FTP`, `LOG_KERN`, `LOG_LOCAL0`, `LOG_LOCAL1`, `LOG_LOCAL2`, `LOG_LOCAL3`, `LOG_LOCAL4`, `LOG_LOCAL5`, `LOG_LOCAL6`, `LOG_LOCAL7`, `LOG_LPR`, `LOG_MAIL`, `LOG_NEWS`, `LOG_SYSLOG`, `LOG_USER`, `LOG_UUCP`.

The default value is `LOG_USER`.

Ice.UseSyslog

Synopsis

`Ice.UseSyslog=`*num*

Description

If *num* is set to a value larger than zero, a special logger is installed that logs to the `syslog` facility instead of standard error. The identifier for `syslog` is the value of `Ice.ProgramName` (Unix only).

D.12 IceSSL Properties

IceSSL.Alias

Synopsis

`IceSSL.Alias=`*alias* (Java)

Description

Selects a particular certificate from the key store specified by `IceSSL.Keystore`. The certificate identified by *alias* is presented to the peer request during authentication.

IceSSL.CertAuthDir

Synopsis

`IceSSL.CertAuthDir=`*`path`* (C++)

Description

Specifies the directory containing the certificates of trusted certificate authorities. The directory must be prepared in advance using the OpenSSL utility **`c_rehash`**. The path name may be specified relative to the default directory defined by `IceSSL.DefaultDir`.

IceSSL.CertAuthFile

Synopsis

`IceSSL.CertAuthFile=`*`file`* (C++, Ice Touch)

Description

Specifies a file containing the certificate of a trusted certificate authority. The file name may be specified relative to the default directory defined by `IceSSL.DefaultDir`.

Platform Notes

C++

The certificate must be encoded using the PEM format.

Ice Touch

The certificate must be encoded using the DER format.

If `IceSSL.DefaultDir` is also defined, IceSSL attempts to open the specified CA certificate file as `Resources/`*`DefaultDir/file`* in the application's resource bundle and as *`DefaultDir/file`* in the file system.

If `IceSSL.DefaultDir` is not defined, IceSSL attempts to open the specified CA certificate file as `Resources/`*`file`* in the application's resource bundle and as *`file`* in the file system.

If this property is not defined, IceSSL looks for suitable CA certificates in the user's keychains and in the system keychain.

IceSSL.CertFile

Synopsis

`IceSSL.CertFile=`*`file`* (.NET, Ice Touch)
`IceSSL.CertFile=`*`file[:file]`* (C++ - Unix)
`IceSSL.CertFile=`*`file[;file]`* (C++ - Windows)

Description

Specifies a file that contains the program's certificate, and may also contain the corresponding private key. The file name may be specified relative to the default directory defined by `IceSSL.DefaultDir`.

Platform Notes

C++

The private key is optional; if not present, the file containing the private key must be identified by `IceSSL.KeyFile`. If a password is required, OpenSSL will prompt the user at the terminal unless the application has installed a password handler or supplied the password using `IceSSL.Password`. The certificate must be encoded using the PEM format.

OpenSSL allows you to specify certificates for both RSA and DSA. To specify both certificates, separate the filenames using the platform's path separator character.

.NET

The file must use the PFX (PKCS#12) format and contain the certificate and its private key. The password for the file must be supplied using `IceSSL.Password`.

Ice Touch

The file must use the PFX (PKCS#12) format and contain the certificate and its private key. The password for the file must be supplied using

`IceSSL.Password`. The certificate is imported into the keychain identified by the `IceSSL.Keychain` property.

If `IceSSL.DefaultDir` is also defined, IceSSL attempts to open the specified certificate file as `Resources/`*`DefaultDir/file`* in the application's resource bundle and as *`DefaultDir/file`* in the file system.

If `IceSSL.DefaultDir` is not defined, IceSSL attempts to open the specified certificate file as `Resources/`*`file`* in the application's resource bundle and as *`file`* in the file system.

IceSSL.CertVerifier

Synopsis

`IceSSL.CertVerifier=`*`classname`* (Java, .NET)

Description

Specifies the name of a Java or .NET class that implements the `IceSSL.CertificateVerifier` interface (see Section 41.5).

IceSSL.CheckCertName

Synopsis

`IceSSL.CheckCertName=`*`num`*

Description

If *`num`* is a value greater than zero, IceSSL attempts to match the server's host name as specified in the proxy endpoint against the common name component of the server certificate's subject name. If no match is found, IceSSL attempts to match the host name against the DNS and IP address fields of the server certificate's subject alternative name. The search does not issue any DNS queries but simply performs a case-insensitive string match. The server's certificate is accepted if its common name or any of its DNS or IP addresses matches the host name in the proxy endpoint. IceSSL skips this validation step if the server does not supply a certificate, or if the proxy endpoint does not include a host name and `Ice.Default.Host` is not defined. This property has no affect on a server's validation of a client's certificate. If no match is found, IceSSL aborts the connection attempt and raises an exception. If not defined, the default value is zero.

IceSSL.CheckCRL

Synopsis

`IceSSL.CheckCRL=`*`num`* (.NET)

Description

If *num* is a value greater than zero, IceSSL checks the certificate revocation list to determine if the peer's certificate has been revoked. If so, IceSSL aborts the connection and raises an exception.

IceSSL.Ciphers

Synopsis

`IceSSL.Ciphers=`*`ciphers`* (C++, Java)

Description

Specifies the cipher suites that IceSSL is allowed to negotiate. A cipher suite is a set of algorithms that satisfies the four requirements for establishing a secure connection: signing and authentication, key exchange, secure hashing, and encryption. Some algorithms satisfy more than one requirement, and there are many possible combinations.

Platform Notes

C++

The value of this attribute is given directly to the OpenSSL library and is dependent on how OpenSSL was compiled. You can obtain a complete list of the supported cipher suites using the command **`openssl ciphers`**. This command will likely generate a long list. To simplify the selection process, OpenSSL supports several classes of ciphers. Classes and ciphers can be excluded by prefixing them with an exclamation point. The special keyword `@STRENGTH` sorts the cipher list in order of their strength, so that SSL gives

preference to the more secure ciphers when negotiating a cipher suite. The `@STRENGTH` keyword must be the last element in the list. The classes are:

`ALL`	Enables all supported cipher suites. This class should be used with caution, as it may enable low-security cipher suites.
`ADH`	Anonymous ciphers.
`LOW`	Low bit-strength ciphers.
`EXP`	Export-crippled ciphers.

Here is an example of a reasonable setting:

```
ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH
```

This value excludes the ciphers with low bit-strength and known problems, and orders the remaining ciphers according to their strength. Note that no warning is given if an unrecognized cipher is specified.

Java

The property value is interpreted as a list of tokens delimited by white space. The plug-in executes the tokens in the order of appearance in order to

assemble the list of enabled cipher suites. The table below describes the tokens:

`NONE`	Disables all cipher suites. If specified, it must be the first token in the list.
`ALL`	Enables all supported cipher suites. If specified, it must be the first token in the list. This token should be used with caution, as it may enable low-security cipher suites.
NAME	Enables the cipher suite matching the given name.
`!`*NAME*	Disables the cipher suite matching the given name.
`(`*EXP*`)`	Enables cipher suites whose names contain the regular expression *EXP*. For example, the value `NONE (.*DH_anon.*)` selects only cipher suites that use anonymous Diffie-Hellman authentication.
`!(`*EXP*`)`	Disables cipher suites whose names contain the regular expression *EXP*. For example, the value `ALL !(.*DH_anon.*)` enables all cipher suites except those that use anonymous Diffie-Hellman authentication.

If not specified, the plug-in uses the security provider's default cipher suites. Set `IceSSL.Trace.Security=1` to determine which cipher suites are enabled by default, or to verify your cipher suite configuration.

IceSSL.DefaultDir

Synopsis

`IceSSL.DefaultDir=`*`path`*

Description

Specifies the default directory in which to look for certificate, key, and key store files. See the descriptions of the relevant properties for more information.

IceSSL.DH.*bits*

Synopsis

`IceSSL.DH.`*`bits`*`=`*`file`* (C++)

Description

Specifies a *file* containing Diffie Hellman parameters whose key length is *bits*, as shown in the following example:

```
IceSSL.DH.1024=dhparams1024.pem
```

IceSSL supplies default parameters for key lengths of 512, 1024, 2048, and 4096 bits, which are used if no user-defined parameters of the desired key length are specified. The file name may be specified relative to the default directory defined by `IceSSL.DefaultDir`. The parameters must be encoded using the PEM format.

IceSSL.EntropyDaemon

Synopsis

`IceSSL.EntropyDaemon=`*file* (C++)

Description

Specifies a Unix domain socket for the entropy gathering daemon, from which OpenSSL gathers entropy data to initialize its random number generator.

IceSSL.FindCert.*location.name*

Synopsis

`IceSSL.FindCert.`*location.name*`=`*criteria* (.NET)

Description

Queries the certificate repository for matching certificates and adds them to the application's collection of certificates that are used for authentication. The value for *location* must be `LocalMachine` or `CurrentUser`.

The *name* corresponds to the .NET enumeration `StoreName` and may be one of the following values: `AddressBook`, `AuthRoot`, `CertificateAuthority`, `Disallowed`, `My`, `Root`, `TrustedPeople`, `TrustedPublisher`. It is also possible to use an arbitrary value for *name*.

The value for *criteria* may be *, in which case all of the certificates in the store are selected. Otherwise, *criteria* must be one or more *field*:*value* pairs separated by white space. The valid field names are described below:

Issuer	Matches a substring of the issuer's name.
IssuerDN	Matches the issuer's entire distinguished name.
Serial	Matches the certificate's serial number.
Subject	Matches a substring of the subject's name.
SubjectDN	Matches the subject's entire distinguished name.
SubjectKeyId	Matches the certificate's subject key identifier.
Thumbprint	Matches the certificate's thumbprint.

The field names are case-insensitive. If multiple criteria are specified, only certificates that match all criteria are selected. Values must be enclosed in single or double quotes to preserve white space.

Multiple occurrences of the property are allowed, but only one query is possible for each location/name combination. The certificates from all queries are combined to form the certificate collection, including a certificate loaded using IceSSL.CertFile. Here are some sample queries:

```
IceSSL.FindCert.LocalMachine.My=issuer:verisign serial:219336
IceSSL.FindCert.CurrentUser.Root=subject:"Joe's Certificate"
```

A server requires a certificate for authentication purposes, therefore IceSSL selects the first certificate in the accumulated collection. This is normally the certificate loaded via IceSSL.CertFile, if that property was defined. Otherwise, one of the certificates from IceSSL.FindCert is selected. Since IceSSL does not guarantee the order in which it evaluates IceSSL.FindCert properties, it is recommended that the criteria select only one certificate.

IceSSL.ImportCert.*location.name*

Synopsis

IceSSL.ImportCert.*location.name*=*file*[;*password*] (.NET)

Description

Imports the certificate in *file* into the specified certificate store. The value for *location* must be `LocalMachine` or `CurrentUser`. The *name* corresponds to the .NET enumeration `StoreName` and may be one of the following values: `AddressBook`, `AuthRoot`, `CertificateAuthority`, `Disallowed`, `My`, `Root`, `TrustedPeople`, `TrustedPublisher`. It is also possible to use an arbitrary value for *name*, which adds a new store to the repository. If you are importing a trusted CA certificate, it must be added to `AuthRoot` or `Root`.

The *password* is optional; it is only necessary if the certificate file also contains a private key or uses a secure storage format such as PFX.

The file name and password may be enclosed in single or double quotes if necessary. The file name may be specified relative to the default directory defined by `IceSSL.DefaultDir`.

Importing a certificate into `LocalMachine` requires administrator privileges, while importing into `CurrentUser` may cause the platform to prompt the user for confirmation.

IceSSL.Keychain

Synopsis

`IceSSL.Keychain=`*name* (Ice Touch)

Description

Specifies the name of keychain in which to import the certificate identified by `IceSSL.CertFile`. If not defined, the keychain named `login` is used by default. Note that this property is only relevant for the iPhone simulator and Mac OS X targets.

IceSSL.KeychainPassword

Synopsis

`IceSSL.KeychainPassword=`*password* (Ice Touch)

Description

Specifies the password for the keychain identified by `IceSSL.Keychain`. If not defined, IceSSL attempts to open the keychain without a password. Note that this property is only relevant for the iPhone simulator and Mac OS X targets.

IceSSL.KeyFile

Synopsis

`IceSSL.KeyFile=`*`file`* (C++)

Description

Specifies a file containing the private key associated with the certificate identified by `IceSSL.CertFile`. The file name may be specified relative to the default directory defined by `IceSSL.DefaultDir`. The key must be encoded using the PEM format.

IceSSL.Keystore

Synopsis

`IceSSL.Keystore=`*`file`* (Java)

Description

Specifies a key store file containing certificates and their private keys. If the key store contains multiple certificates, you should specify a particular one to use for authentication using `IceSSL.Alias`. IceSSL first attempts to open the file as a class loader resource and then as a regular file. If the file cannot be found in the file system, IceSSL attempts to open the file relative to the directory specified by `IceSSL.DefaultDir`. The format of the file is determined by `IceSSL.KeystoreType`.

If this property is not defined, the application will not be able to supply a certificate during SSL handshaking. As a result, the application may not be able to negotiate a secure connection, or might be required to use an anonymous cipher suite.

IceSSL.KeystorePassword

Synopsis

`IceSSL.KeystorePassword=`*`password`* (Java)

Description

Specifies the password used to verify the integrity of the key store defined by `IceSSL.Keystore`. The integrity check is skipped if this property is not defined. It is a security risk to use a plain-text password in a configuration file.

IceSSL.KeystoreType

Synopsis

`IceSSL.KeystoreType=`*`type`* (Java)

Description

Specifies the format of the key store file defined by `IceSSL.Keystore`. Legal values are `JKS` and `PKCS12`. If not defined, the JVM's default value is used (normally `JKS`).

IceSSL.Password

Synopsis

`IceSSL.Password=`*`password`*

Description

Specifies the password necessary to decrypt the private key.

Platform Notes

C++

This property supplies the password that was used to secure the private key contained in the file defined by `IceSSL.CertFile` or `IceSSL.KeyFile`. If this property is not defined and you have not installed a password callback object, OpenSSL will prompt the user for a password if one is necessary.

Java

This property supplies the password that was used to secure the private key contained in the key store defined by `IceSSL.Keystore`. All of the keys in the key store must use the same password.

.NET

This property supplies the password that was used to secure the file defined by `IceSSL.CertFile`.

Ice Touch

This property supplies the password that was used to secure the file defined by `IceSSL.CertFile`.

It is a security risk to use a plain-text password in a configuration file.

IceSSL.PasswordCallback

Synopsis

`IceSSL.PasswordCallback=`*`classname`* (Java, .NET)

Description

Specifies the name of a Java or .NET class that implements the `IceSSL.PasswordCallback` interface (see Section 41.6.1).

IceSSL.PasswordRetryMax

Synopsis

`IceSSL.PasswordRetryMax=`*`num`* (C++)

Description

Specifies the number of attempts IceSSL should allow the user to make when entering a password. If not defined, the default value is `3`.

IceSSL.Protocols

Synopsis

`IceSSL.Protocols=`*`list`* (C++, Java, .NET)

Description

Specifies the protocols to allow during SSL handshaking. Legal values are `SSL3` and `TLS1`. You may also specify both values, separate by commas or white space. If this property is not defined, the platform's default is used.

IceSSL.Random

Synopsis

`IceSSL.Random=`*`filelist`* (C++, Java)

Description

Specifies one or more files containing data to use when seeding the random number generator. The file names should be separated using the platform's path separator (a colon on Unix and a semicolon on Windows). The file names may be specified relative to the default directory defined by `IceSSL.DefaultDir`.

In Java, IceSSL first attempts to open the files as class loader resources and then as regular files.

IceSSL.Trace.Security

Synopsis

`IceSSL.Trace.Security=`*`num`* (C++, Java, .NET)

Description

The SSL plug-in trace level:

`0`	No security tracing (default).
`1`	Display diagnostic information about SSL connections.

IceSSL.TrustOnly

Synopsis

`IceSSL.TrustOnly=`*`ENTRY`*`[;`*`ENTRY`*`;...]` (C++, Java, .NET)

Description

Identifies trusted and untrusted peers. This family of properties provides an additional level of authentication by using the peer certificate's distinguished name (DN) to decide whether to accept or reject a connection.

Each *ENTRY* in the property value consists of relative distinguished name (RDN) components, formatted according to the rules in RFC 2253. Specifically, the components must be separated by commas, and any component that contains a comma must be escaped or enclosed in quotes. For example, the following two property definitions are equivalent:

```
IceSSL.TrustOnly=O="Acme, Inc.",OU=Sales
IceSSL.TrustOnly=O=Acme\, Inc.,OU="Sales"
```

Use a semicolon to separate multiple entries in a property:

```
IceSSL.TrustOnly=O=Acme\, Inc.,OU=Sales;O=Acme\, Inc.,OU=Marketing
```

By default, each entry represents an acceptance entry. A `!` character appearing at the beginning of an entry signifies a rejection entry. The order of the entries in a property is not important.

After the SSL engine has successfully completed its authentication process, IceSSL evaluates the relevant `IceSSL.TrustOnly` properties in an attempt to find an entry that matches the peer certificate's DN. For a match to be successful, the peer DN must contain an exact match for all of the RDN components in an entry. An entry may contain as many RDN components as you wish, depending on how narrowly you need to restrict access. The order of the RDN components in an entry is not important.

The connection semantics are described below:

1. IceSSL aborts the connection if any rejection or acceptance entries are defined and the peer does not supply a certificate.
2. IceSSL aborts the connection if the peer DN matches any rejection entry. (This is true even if the peer DN also matches an acceptance entry.)
3. IceSSL accepts the connection if the peer DN matches any acceptance entry, or if no acceptance entries are defined.

The example shown above limits access to people in the sales and marketing departments:

```
IceSSL.TrustOnly=O=Acme\, Inc.,OU=Sales;O=Acme\, Inc.,OU=Marketing
```

If it later becomes necessary to deny access to certain individuals in these departments, you can add a rejection entry and restart the program:

```
IceSSL.TrustOnly=O=Acme\, Inc.,OU=Sales; O=Acme\,
Inc.,OU=Marketing; !O=Acme\, Inc.,CN=John Smith
```

While testing your trust configuration, you may find it helpful to set the `IceSSL.Trace.Security` property to a non-zero value, which causes IceSSL to display the DN of each peer during connection establishment.

This property affects incoming and outgoing connections. IceSSL also supports similar properties that affect only incoming connections or only outgoing connections.

IceSSL.TrustOnly.Client

Synopsis

`IceSSL.TrustOnly.Client=`*`ENTRY`*`[;`*`ENTRY`*`;...]` (C++, Java, .NET)
`IceSSL.TrustOnly.Client=`*`ID`* (Ice Touch)

Description

Identifies trusted and untrusted peers for outgoing (client) connections. The entries defined in this property are combined with those of `IceSSL.TrustOnly`.

Platform Notes

Ice Touch

For an outgoing connection to succeed, the peer certificate's subject key identifier must match the property value exactly. The property value is formatted as a series of hexadecimal values separated by colons or spaces, as shown in the example below:

```
C2:E8:D3:33:D7:83:99:6E:08:F7:C2:34:31:F7:1E:8E:44:87:38:57
```

Since this value is specific to a single certificate authority, this property is valid only when used in conjunction with `IceSSL.CertAuthFile`.

IceSSL.TrustOnly.Server

Synopsis

`IceSSL.TrustOnly.Server=`*`ENTRY`*`[;`*`ENTRY`*`;...]` (C++, Java, .NET)

Description

Identifies trusted and untrusted peers for incoming ("server") connections. The entries defined in this property are combined with those of `IceSSL.TrustOnly`. To configure trusted and untrusted peers for a particular object adapter, use `IceSSL.TrustOnly.Server.`*`AdapterName`*.

IceSSL.TrustOnly.Server.*AdapterName*

Synopsis

`IceSSL.TrustOnly.Server.`*`AdapterName`*`=`*`ENTRY`*`[;`*`ENTRY`*`;...]` (C++, Java, .NET)

Description

Identifies trusted and untrusted peers for incoming (server) connections to the object adapter *`AdapterName`*. The entries defined in this property are combined with those of `IceSSL.TrustOnly` and `IceSSL.TrustOnly.Server`.

IceSSL.Truststore

Synopsis

`IceSSL.Truststore=`*`file`* (Java)

Description

Specifies a key store file containing the certificates of trusted certificate authorities. IceSSL first attempts to open the file as a class loader resource and then as a regular file. If the file cannot be found in the file system, IceSSL attempts to open the file relative to the directory specified by `IceSSL.DefaultDir`. The format of the file is determined by `IceSSL.TruststoreType`.

If this property is not defined, IceSSL uses the value of `IceSSL.Keystore` by default. If no truststore is specified and the keystore does not contain a valid certificate chain, the application will not be able to authenticate the peer's certificate during SSL handshaking. As a result, the application may not be able to negotiate a secure connection, or might be required to use an anonymous cipher suite.

IceSSL.TruststorePassword

Synopsis

`IceSSL.TruststorePassword=`*`password`* (Java)

Description

Specifies the password used to verify the integrity of the key store defined by `IceSSL.Truststore`. The integrity check is skipped if this property is not defined. It is a security risk to use a plain-text password in a configuration file.

IceSSL.TruststoreType

Synopsis

`IceSSL.TruststoreType=`*`type`* (Java)

Description

Specifies the format of the key store file defined by `IceSSL.Truststore`. Legal values are `JKS` and `PKCS12`. If not defined, the default value is `JKS`.

IceSSL.VerifyDepthMax

Synopsis

`IceSSL.VerifyDepthMax=`*`num`* (C++, Java, .NET)

Description

Specifies the maximum depth of a trusted peer's certificate chain, including the peer's certificate. A value of zero accepts chains of any length. If not defined, the default value is `2`.

IceSSL.VerifyPeer

Synopsis

`IceSSL.VerifyPeer=`*`num`* (C++, Java, .NET)

Description

Specifies the verification requirements to use during SSL handshaking. The legal values are shown in the table below. If this property is not defined, the default value is `2`.

`0`	For an outgoing connection, the client verifies the server's certificate (if an anonymous cipher is not used) but does not abort the connection if verification fails. For an incoming connection, the server does not request a certificate from the client.
`1`	For an outgoing connection, the client verifies the server's certificate and aborts the connection if verification fails. For an incoming connection, the server requests a certificate from the client and verifies it if one is provided, aborting the connection if verification fails.
`2`	For an outgoing connection, the semantics are the same as for the value `1`. For an incoming connection, the server requires a certificate from the client and aborts the connection if verification fails.

Platform Notes

.NET

This property has no effect on outgoing connections, since .NET always uses the semantics of value `2`. For an incoming connection, the value `0` has the same semantics as the value `1`.

D.13 IceBox Properties

IceBox.InheritProperties

Synopsis

`IceBox.InheritProperties=`*`num`*

Description

If *num* is set to a value larger than zero, each service inherits the configuration properties of the IceBox server's communicator. If not defined, the default value is zero.

IceBox.InstanceName

Synopsis

`IceBox.InstanceName=`*`name`*

Description

Specifies an alternate identity category for the IceBox service manager object. If defined, the identity of the object becomes *`name`*`/ServiceManager`. If not specified, the default identity category is `IceBox`.

IceBox.LoadOrder

Synopsis

`IceBox.LoadOrder=`*`names`*

Description

Determines the order in which services are loaded. The service manager loads the services in the order they appear in *`names`*, where each service name is separated by a comma or white space. Any services not mentioned in *`names`* are loaded afterward, in an undefined order.

IceBox.PrintServicesReady

Synopsis

`IceBox.PrintServicesReady=`*`token`*

Description

If this property is set to a value greater than zero, the service manager prints "*`token`* ready" on standard output once initialization of all the services is complete. This is useful for scripts that wish to wait until all services are ready to be used.

IceBox.Service.*name*

Synopsis

`IceBox.Service.`*`name`*`=`*`entry_point`*`[,`*`version`*`] [`*`args`*`]`

Description

Defines a service to be loaded during IceBox initialization. Any arguments that follow the entry point are examined; those matching the `--`*`name`*`=`*`value` pattern are interpreted as property definitions and appear in the property set of the communicator that is passed to the service `start` method, and all remaining arguments are passed to the `start` method in the `args` parameter.

Platform Notes

C++

The value of *`entry_point`* has the form *`basename`*`[,`*`version`*`]:`*`function`*. The *`basename`* and optional *`version`* components are used to construct the name of a DLL or shared library. If no version is supplied, the version is the empty string. The *`function`* component is the name of a function with extern C linkage. For example, the entry point `IceStormService,34:createIceStorm` implies a shared library name of `libIceStormService.so.34` on Unix and `IceStormService34.dll` on Windows. Furthermore, if IceBox is built on Windows with debugging, a `d` is automatically appended to the version (e.g., `IceStormService34d.dll`).

The function must be declared with extern C linkage and have the following signature:

```
IceBox::Service* function(Ice::CommunicatorPtr c);
```

Note that the function must return a pointer and not a smart pointer. The Ice core deallocates the object when it unloads the library. The communicator instance passed to this function is the server's communicator, which is not the same as the communicator passed to the service's `start` method.

Java

The value of *`entry_point`* is the name of a class that must implement the `IceBox.Service` interface. The class must provide a public default constructor.

.NET

The value of *entry_point* has the form *assembly*:*class*. The assembly can be the full assembly name, such as `myplugin, Version=0.0.0.0, Culture=neutral`, or an assembly DLL name such as `myplugin.dll`. The specified class must implement the `IceBox.Service` interface and define a public default constructor.

IceBox.ServiceManager.*name*

Synopsis

`IceBox.ServiceManager.`*name*`=`*value*

Description

IceBox uses the adapter name `IceBox.ServiceManager` for its object adapter. Therefore, all the adapter properties detailed in Section D.4 can be used to configure the IceBox object adapter.

IceBox.UseSharedCommunicator.*name*

Synopsis

`IceBox.UseSharedCommunicator.`*name*`=`*num*

Description

If *num* is set to a value larger than zero, the service manager supplies the service with the given *name* a communicator that might be shared by other services. If the `IceBox.InheritProperties` property is also defined, the shared communicator inherits the properties of the IceBox server. If not defined, the default value is zero.

D.14 IceBoxAdmin Properties

IceBoxAdmin.ServiceManager.Proxy

Synopsis

`IceBoxAdmin.ServiceManager.Proxy=`*`proxy`*

Description

This property configures the proxy that is used by the **`iceboxadmin`** utility to locate the service manager.

D.15 IceGrid Properties

Ice.Plugin.DB

Synopsis

`Ice.Plugin.DB=IceGridSqlDB:createSqlDB`

Description

The database storage mechanism used by the IceGrid registry is abstracted in such a way that an alternate mechanism can be selected using the standard Ice plug-in facility. If this property is not defined, the registry uses Freeze as its default storage mechanism. Alternatively, you can define the `Ice.Plugin.DB` property as shown above to use a SQL database instead, in which case additional properties may also be necessary (see the `IceGrid.SQL` properties below).

IceGrid.InstanceName

Synopsis

`IceGrid.InstanceName=`*`name`*

Description

Specifies an alternate identity category for the IceGrid objects. If defined, the identities of the IceGrid objects become:

name/AdminSessionManager
name/AdminSessionManager-*replica*
name/AdminSSLSessionManager
name/AdminSSLSessionManager-*replica*
name/NullPermissionsVerifier
name/NullSSLPermissionsVerifier
name/Locator
name/Query
name/Registry
name/Registry-*replica*
name/RegistryUserAccountMapper
name/RegistryUserAccountMapper-*replica*
name/SessionManager
name/SSLSessionManager

If not specified, the default identity category is `IceGrid`.

IceGrid.Node.AllowEndpointsOverride

Synopsis

`IceGrid.Node.AllowEndpointsOverride=`*num*

If *num* is set to a non-zero value, an IceGrid node permits servers to override previously set endpoints even if the server is active. Setting this property to a non-zero value is necessary if the servers managed by the node call the object adapter `refreshPublishedEndpoints` method. The default value of *num* is zero.

IceGrid.Node.AllowRunningServersAsRoot

Synopsis

`IceGrid.Node.AllowRunningServersAsRoot=`*value*

If *num* is set to a non-zero value, an IceGrid node will permit servers started by the node to run with super-user privileges. Note that you should not set this property unless the node uses a secure endpoint; otherwise, clients can start arbitrary processes with super-user privileges on the node's machine.

The default value of *num* is zero.

IceGrid.Node.*name*

Synopsis

`IceGrid.Node.`*`name`*`=`*`value`*

Description

An IceGrid node uses the adapter name `IceGrid.Node` for the object adapter that the registry contacts to communicate with the node. Therefore, the adapter properties detailed in Section D.4 can be used to configure this adapter.

IceGrid.Node.CollocateRegistry

Synopsis

`IceGrid.Node.CollocateRegistry=`*`num`*

Description

If *num* is set to a value larger than zero, the node collocates the IceGrid registry.

The collocated registry is configured with the same properties as the standalone IceGrid registry.

IceGrid.Node.Data

Synopsis

`IceGrid.Node.Data=`*`path`*

Description

Defines the path of the IceGrid node data directory. The node creates `distrib`, `servers`, and `tmp` subdirectories in this directory if they do not already exist. The `distrib` directory contains distribution files downloaded by the node from an IcePatch2 server. The `servers` directory contains configuration data for each deployed server. The `tmp` directory holds temporary files.

IceGrid.Node.DisableOnFailure

Synopsis

`IceGrid.Node.DisableOnFailure=`*`num`*

Description

The node considers a server to have terminated improperly if it has a non-zero exit code or if it exits due to one of the signals `SIGABRT`, `SIGBUS`, `SIGILL`, `SIGFPE`, or `SIGSEGV`. The node marks such a server as disabled if *num* is a non-zero value; a disabled server cannot be activated on demand. For values of *num* greater than zero, the server is disabled for *num* seconds. If *num* is a negative value, the server is disabled indefinitely, or until it is explicitly enabled or started via an administrative action. The default value is zero, meaning the node does not disable servers in this situation.

IceGrid.Node.Name

Synopsis

`IceGrid.Node.Name=`*`name`*

Description

Defines the *name* of the IceGrid node. All nodes using the same registry must have unique names; a node refuses to start if there is a node with the same name running already. This property must be defined for each node.

IceGrid.Node.Output

Synopsis

`IceGrid.Node.Output=`*`path`*

Description

Defines the path of the IceGrid node output directory. If set, the node redirects the `stdout` and `stderr` output of the started servers to files named `server.out` and `server.err` in this directory. Otherwise, the started servers share the `stdout` and `stderr` of the node's process.

IceGrid.Node.PrintServersReady

Synopsis

`IceGrid.Node.PrintServersReady=`*`token`*

Description

The IceGrid node prints "*`token`* ready" on standard output after all the servers managed by the node are ready. This is useful for scripts that wish to wait until all servers have been started and are ready for use.

IceGrid.Node.ProcessorSocketCount

Synopsis

`IceGrid.Node.ProcessorSocketCount=`*`num`*

Description

This property sets the number of processor sockets. This value is reported by the **`icegridadmin node processors`** command (Section 38.24.1). On Windows Vista, Windows Server 2008, and Linux systems, the number of processors is set automatically by the Ice run time. On other systems, the run time cannot obtain the socket count from the operating system; you can use this property to set the number of processor sockets manually on such systems.

IceGrid.Node.PropertiesOverride

Synopsis

`IceGrid.Node.PropertiesOverride=`*`overrides`*

Description

Defines a list of properties that override the properties defined in server deployment descriptors. For example, in some cases it is desirable to set the property `Ice.Default.Host` for servers, but not in server deployment descriptors. The property definitions must be separated by white space.

IceGrid.Node.RedirectErrToOut

Synopsis

`IceGrid.Node.RedirectErrToOut=`*`num`*

Description

If *num* is set to a value larger than zero, the `stderr` of each started server is redirected to the server's `stdout`.

IceGrid.Node.Trace.Activator

Synopsis

`IceGrid.Node.Trace.Activator=`*`num`*

Description

The activator trace level:

0	No activator trace (default).
1	Trace process activation, termination.
2	Like 1, but more verbose, including process signaling and more diagnostic messages on process activation.
3	Like 2, but more verbose, including more diagnostic messages on process activation (e.g., path, working directory, and arguments of the activated process).

IceGrid.Node.Trace.Adapter

Synopsis

`IceGrid.Node.Trace.Adapter=`*`num`*

Description

The object adapter trace level:

0	No object adapter trace (default).

1	Trace object adapter addition, removal.
2	Like 1, but more verbose, including object adapter activation and deactivation and more diagnostic messages.
3	Like 2, but more verbose, including object adapter transitional state change (for example, “waiting for activation”).

IceGrid.Node.Trace.Patch

Synopsis

`IceGrid.Node.Trace.Patch=`*`num`*

Description

The patch trace level:

0	No patching trace (default).
1	Show summary of patch progress.
2	Like 1, but more verbose, including download statistics.
3	Like 2, but more verbose, including checksum information.

IceGrid.Node.Trace.Replica

Synopsis

`IceGrid.Node.Trace.Replica=`*`num`*

Description

The replica trace level:

0	No replica trace (default).
1	Trace session lifecycle between nodes and replicas.
2	Like 1, but more verbose, including session establishment attempts and failures.

3	Like 2, but more verbose, including keep alive messages sent to the replica.

IceGrid.Node.Trace.Server

Synopsis

`IceGrid.Node.Trace.Server=`*`num`*

Description

The server trace level:

0	No server trace (default).
1	Trace server addition, removal.
2	Like 1, but more verbose, including server activation and deactivation and more diagnostic messages.
3	Like 2, but more verbose, including server transitional state change (activating and deactivating).

IceGrid.Node.UserAccountMapper

Synopsis

`IceGrid.Node.UserAccountMapper=`*`proxy`*

Description

Specifies the proxy of an object that implements the `IceGrid::UserAccount-Mapper` interface. The IceGrid node invokes this proxy to map session identifiers (the user id for sessions created with a user name and password, or the distinguished name for sessions created from a secure connection) to user accounts.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

IceGrid.Node.UserAccounts

Synopsis

`IceGrid.Node.UserAccounts=`*`file`*

Description

Specifies the file name of an IceGrid node user account map file. Each line of the file must contain an identifier and a user account, separated by white space. The identifier will be matched against the client session identifier (the user id for sessions created with a user name and password, or the distinguished name for sessions created from a secure connection). This user account map file is used by the node to map session identifiers to user accounts. This property is ignored if `IceGrid.Node.UserAccountMapper` is defined.

IceGrid.Node.WaitTime

Synopsis

`IceGrid.Node.WaitTime=`*`num`*

Description

Defines the interval in seconds that IceGrid waits for server activation and deactivation.

If a server is automatically activated and does not register its object adapter endpoints within this time interval, the node assumes there is a problem with the server and return an empty set of endpoints to the client.

If a server is being gracefully deactivated and IceGrid does not detect the server deactivation during this time interval, IceGrid kills the server.

The default value is 60 seconds.

IceGrid.Registry.AdminCryptPasswords

Synopsis

`IceGrid.Registry.AdminCryptPasswords=`*`file`*

Description

Specifies the file name of an IceGrid registry access control list for admin clients (see Section 38.11.2). Each line of the file must contain a user name and a password, separated by white space. The password must be a 13-character crypt-encoded string. If this property is not defined, the default value is `admin-passwords`. This property is ignored if `IceGrid.Registry.AdminPermissionsVerifier` is defined.

IceGrid.Registry.AdminPermissionsVerifier

Synopsis

`IceGrid.Registry.AdminPermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::PermissionsVerifier` interface (see Section 38.11.2). The IceGrid registry invokes this proxy to validate each new admin session created by a client with the `IceGrid::Registry` interface.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

IceGrid.Registry.AdminSessionFilters

Synopsis

`IceGrid.Registry.AdminSessionFilters=`*`num`*

Description

This property controls whether IceGrid establishes filters for sessions created with the IceGrid session manager (see Section 38.15.2). If *`num`* is set to a value larger than zero, IceGrid establishes these filters, so Glacier2 limits access to the `IceGrid::AdminSession` object and the `IceGrid::Admin` object that is returned by the `getAdmin` operation. If *`num`* is set to zero, IceGrid does not establish filters, so access to these objects is controlled solely by Glacier2's configuration.

The default value is 1.

IceGrid.Registry.AdminSessionManager.*name*

Synopsis

`IceGrid.Registry.SessionManager.`*`name`*`=`*`value`*

Description

The IceGrid registry uses the adapter name `IceGrid.Registry.AdminSession-Manager` for the object adapter that processes incoming requests from IceGrid administrative sessions (see Section 38.14). Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter. (Note any setting of `IceGrid.Registry.SessionManager.AdapterId` is ignored because the registry always provides a direct adapter.)

For security reasons, defining endpoints for this object adapter is optional. If you do define endpoints, they should only be accessible to Glacier2 routers used to create IceGrid administrative sessions.

IceGrid.Registry.AdminSSLPermissionsVerifier

Synopsis

`IceGrid.Registry.AdminSSLPermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SSLPermis-sionsVerifier` interface (see Section 38.11.2). The IceGrid registry invokes this proxy to validate each new admin session created by a client from a secure connection with the `IceGrid::Registry` interface.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

IceGrid.Registry.Client.*name*

Synopsis

`IceGrid.Registry.Client.`*`name`*`=`*`value`*

Description

IceGrid uses the adapter name `IceGrid.Registry.Client` for the object adapter that processes incoming requests from clients. Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter. (Note any setting of `IceGrid.Registry.Client.AdapterId` is ignored because the registry always provides a direct adapter.)

Note that `IceGrid.Registry.Client.Endpoints` controls the client endpoint for the registry. The port numbers 4061 (for TCP) and 4062 (for SSL) are reserved for the registry by the Internet Assigned Numbers Authority (IANA).

IceGrid.Registry.CryptPasswords

Synopsis

`IceGrid.Registry.CryptPasswords=`*`file`*

Description

Specifies the file name of an IceGrid registry access control list (see Section 38.11.2). Each line of the file must contain a user name and a password, separated by white space. The password must be a 13-character crypt-encoded string. If this property is not defined, the default value is `passwords`. This property is ignored if `IceGrid.Registry.PermissionsVerifier` is defined.

IceGrid.Registry.Data

Synopsis

`IceGrid.Registry.Data=`*`path`*

Description

Defines the path of the IceGrid registry data directory. The directory specified in *`path`* must already exist. This property must be defined only when the registry uses Freeze or SQLite for its database storage.

IceGrid.Registry.DefaultTemplates

Synopsis

`IceGrid.Registry.DefaultTemplates=`*`path`*

Description

Defines the path name of an XML file containing default template descriptors. A sample file named `config/templates.xml` that contains convenient server templates for Ice services is provided in the Ice distribution.

IceGrid.Registry.DynamicRegistration

Synopsis

`IceGrid.Registry.DynamicRegistration=`*`num`*

Description

If *`num`* is set to a value larger than zero, the locator registry does not require Ice servers to preregister object adapters and replica groups, but rather creates them automatically if they do not exist. If this property is not defined, or *`num`* is set to zero, an attempt to register an unknown object adapter or replica group causes adapter activation to fail with `Ice.NotRegisteredException`. An object adapter registers itself when the *`adapter`*`.AdapterId` property is defined. The *`adapter`*`.ReplicaGroupId` property identifies the replica group.

IceGrid.Registry.Internal.*name*

Synopsis

`IceGrid.Registry.Internal.`*`name`*`=`*`value`*

Description

The IceGrid registry uses the adapter name `IceGrid.Registry.Internal` for the object adapter that processes incoming requests from nodes and slave replicas. Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter. (Note any setting of `IceGrid.Registry.Internal.AdapterId` is ignored because the registry always provides a direct adapter.)

IceGrid.Registry.NodeSessionTimeout

Synopsis

`IceGrid.Registry.NodeSessionTimeout=`*`num`*

Description

Each IceGrid node establishes a session with the registry that must be refreshed periodically. If a node does not refresh its session within *`num`* seconds, the node's session is destroyed and the servers deployed on that node become unavailable to new clients. If not specified, the default value is 30 seconds.

IceGrid.Registry.PermissionsVerifier

Synopsis

`IceGrid.Registry.PermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::Permissions-Verifier` interface (see Section 38.11.2). The IceGrid registry invokes this proxy to validate each new client session created by a client with the `IceGrid::Registry` interface.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

IceGrid.Registry.ReplicaName

Synopsis

`IceGrid.Registry.ReplicaName=`*`name`*

Description

Specifies the name of a registry replica. If not defined, the default value is `Master`, which is the name reserved for the master replica. Each registry replica must have a unique name. See Section 38.12 for more information on registry replication.

IceGrid.Registry.ReplicaSessionTimeout

Synopsis

`IceGrid.Registry.ReplicaSessionTimeout=`*`num`*

Description

Each IceGrid registry replica establishes a session with the master registry that must be refreshed periodically. If a replica does not refresh its session within *num* seconds, the replica's session is destroyed and the replica doesn't receive anymore replication information from the master registry. If not specified, the default value is 30 seconds.

IceGrid.Registry.Server.*name*

Synopsis

`IceGrid.Registry.Server.`*`name`*`=`*`value`*

Description

The IceGrid registry uses the adapter name `IceGrid.Registry.Server` for the object adapter that processes incoming requests from servers. Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter. (Note any setting of `IceGrid.Registry.Server.AdapterId` is ignored because the registry always provides a direct adapter.)

IceGrid.Registry.SessionFilters

Synopsis

`IceGrid.Registry.SessionFilters=`*`num`*

Description

This property controls whether IceGrid establishes filters for sessions created with the IceGrid session manager (see Section 38.15.3). If *num* is set to a value larger than zero, IceGrid establishes these filters, so Glacier2 limits access to the `IceGrid::Query` and `IceGrid::Session` objects, and to objects and adapters allocated by the session. If *num* is set to zero, IceGrid does not establish filters, so access to objects is controlled solely by Glacier2's configuration.

The default value is 0.

IceGrid.Registry.SessionManager.*name*

Synopsis

`IceGrid.Registry.SessionManager.`*`name`*`=`*`value`*

Description

The IceGrid registry uses the adapter name `IceGrid.Registry.SessionManager` for the object adapter that processes incoming requests from client sessions (see Section 38.11). Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter. (Note any setting of `IceGrid.Registry.Session-Manager.AdapterId` is ignored because the registry always provides a direct adapter.)

For security reasons, defining endpoints for this object adapter is optional. If you do define endpoints, they should only be accessible to Glacier2 routers used to create IceGrid client sessions.

IceGrid.Registry.SessionTimeout

Synopsis

`IceGrid.Registry.SessionTimeout=`*`num`*

Description

IceGrid clients or administrative clients might establish a session with the registry. This session must be refreshed periodically. If the client does not refresh its session within *`num`* seconds, the session is destroyed. If not specified, the default value is 30 seconds.

IceGrid.Registry.SSLPermissionsVerifier

Synopsis

`IceGrid.Registry.SSLPermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SSLPermissionsVerifier` interface (see Section 38.11.2). The IceGrid registry invokes this proxy to validate each new client session created by a client from a secure connection with the `IceGrid::Registry` interface.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

IceGrid.Registry.Trace.Adapter

Synopsis

`IceGrid.Registry.Trace.Adapter=`*`num`*

Description

The object adapter trace level:

`0`	No object adapter trace (default).
`1`	Trace object adapter registration, removal, and replication.

IceGrid.Registry.Trace.Application

Synopsis

`IceGrid.Registry.Trace.Adapter=`*`num`*

Description

The application trace level:

0	No application trace (default).
1	Trace application addition, update and removal.

IceGrid.Registry.Trace.Locator

Synopsis

`IceGrid.Registry.Trace.Locator=`*`num`*

Description

The locator and locator registry trace level:

0	No locator trace (default).
1	Trace failures to locate an adapter or object, and failures to register adapter endpoints.
2	Like 1, but more verbose, including registration of adapter endpoints.

IceGrid.Registry.Trace.Node

Synopsis

`IceGrid.Registry.Trace.Node=`*`num`*

Description

The node trace level:

0	No node trace (default).
1	Trace node registration, removal.
2	Like 1, but more verbose, including load statistics.

IceGrid.Registry.Trace.Object

Synopsis

`IceGrid.Registry.Trace.Object=`*`num`*

Description

The object trace level:

0	No object trace (default).
1	Trace object registration, removal.

IceGrid.Registry.Trace.Patch

Synopsis

`IceGrid.Registry.Trace.Patch=`*`num`*

Description

The patch trace level:

0	No patching trace (default).
1	Show summary of patch progress.

IceGrid.Registry.Trace.Replica

Synopsis

`IceGrid.Registry.Trace.Replica=`*`num`*

Description

The server trace level:

0	No server trace (default).
1	Trace session lifecycle between master replica and slaves.

IceGrid.Registry.Trace.Server

Synopsis

`IceGrid.Registry.Trace.Server=`*`num`*

Description

The server trace level:

0	No server trace (default).
1	Trace server registration, removal.

IceGrid.Registry.Trace.Session

Synopsis

`IceGrid.Registry.Trace.Session=`*`num`*

Description

The session trace level:

0	No client or admin session trace (default).
1	Trace client or admin session registration, removal.
2	Like 1, but more verbose, includes keep alive messages.

IceGrid.Registry.UserAccounts

Synopsis

`IceGrid.Registry.UserAccounts=`*`file`*

Description

Specifies the file name of an IceGrid registry user account map file. Each line of the file must contain an identifier and a user account, separated by white space. The identifier will be matched against the client session identifier (the user id for sessions created with a user name and password, or the distinguished name for sessions created from a secure connection). This user account map file is used by IceGrid nodes to map session identifiers to user accounts if the nodes' `IceGrid.Node.UserAccountMapper` property is set to the proxy `IceGrid/RegistryUserAccountMapper`.

IceGrid.SQL.DatabaseType

Synopsis

`IceGrid.SQL.DatabaseType=`*`type`*

Description

This property determines the type of the database for `QSqlDatabase::addDatabase`. Permissible values of *`type`* are `QSQLITE`, `QMSQL`, `QPSQL`, and `QODBC`.

IceGrid.SQL.DatabaseName

Synopsis

`IceGrid.SQL.Database=`*`name`*

Description

This property determines the name of the database for `QSqlDatabase::setDatabaseName`. For SQLite, *`name`* is the file name for the database. For MySQL and PostgreSQL, `name` is the database name. For SQL Server, *`name`* is the DSN name.

IceGrid.SQL.HostName

Synopsis

`IceGrid.SQL.HostName=`*`name`*

Description

This property determines the host name for `QSqlDatabase::setHostName`. For PostgreSQL and MySQL, *`name`* is the database host name. For SQL Server, *`name`* is the host name and server name separated by a blackslash, such `localhost\SQLExpress`. For SQLite, this property has no effect and need not be set.

IceGrid.SQL.Port

Synopsis

`IceGrid.SQL.Port=`*`num`*

Description

This property determines the port number for `QSqlDatabase::setPort`.

IceGrid.SQL.UserName

Synopsis

`IceGrid.SQL.UserName=`*`name`*

Description

This property determines the user name for `QSqlDatabase::setUserName`. For SQLite, this property has no effect and need not be set.

IceGrid.SQL.Password

Synopsis

`IceGrid.SQL.Password=`*`password`*

Description

This property determines the password for `QSqlDatabase::setPassword`. For SQLite, this property has no effect and need not be set.

D.16 IceGrid Administrative Client Properties

IceGridAdmin.AuthenticateUsingSSL

Synopsis

`IceGridAdmin.AuthenticateUsingSSL=`*`num`*

Description

If *num* is a value greater than zero, **icegridadmin** uses SSL authentication when establishing its session with the IceGrid registry. If not defined or the value is zero, **icegridadmin** uses user name and password authentication. See Section 38.24.1 for more information.

IceGridAdmin.Password

Synopsis

`IceGridAdmin.Password=`*`password`*

Description

Specifies the password that **icegridadmin** should use when authenticating its session with the IceGrid registry. For security reasons you may prefer not to define a password in a plain-text configuration property, in which case you should omit this property and allow **icegridadmin** to prompt you for it interactively. This property is ignored when SSL authentication is enabled via `IceGrid-Admin.AuthenticateUsingSSL`. See Section 38.24.1 for more information.

IceGridAdmin.Replica

Synopsis

`IceGridAdmin.Replica=`*`name`*

Description

Specifies the name of the registry replica to contact. If not defined, the default value is `Master`. See Section 38.24.1 for more information.

IceGridAdmin.Trace.Observers

Synopsis

`IceGridAdmin.Trace.Observers=`*`num`*

Description

If *num* is a value greater than zero, the IceGrid graphical administrative client displays trace information about the observer callbacks it receives from the registry. If not defined, the default value is zero.

IceGridAdmin.Trace.SaveToRegistry

Synopsis

`IceGridAdmin.Trace.SaveToRegistry=num`

Description

If *num* is a value greater than zero, the IceGrid graphical administrative client displays trace information about the modifications it commits to the registry. If not defined, the default value is zero.

IceGridAdmin.Username

Synopsis

`IceGridAdmin.Username=name`

Description

Specifies the username that **`icegridadmin`** should use when authenticating its session with the IceGrid registry. This property is ignored when SSL authentication is enabled via `IceGridAdmin.AuthenticateUsingSSL`. See Section 38.24.1 for more information.

D.17 IceStorm Properties

All IceStorm properties use the IceStorm service name as their prefix. For example, suppose an IceBox configuration loads IceStorm as shown below:

```
IceBox.Service.DataFeed=IceStormService,...
```

IceStorm properties defined for this service must use `DataFeed` as the prefix, such as `DataFeed.Discard.Interval=10`.

Ice.Plugin.DB

Synopsis

`Ice.Plugin.DB=IceStormSqlDB:createSqlDB`

Description

The database storage mechanism used by IceStorm is abstracted in such a way that an alternate mechanism can be selected using the standard Ice plug-in facility. If this property is not defined, IceStorm uses Freeze as its default storage mechanism. Alternatively, you can define the `Ice.Plugin.DB` property as shown above to use a SQL database instead, in which case additional properties may also be necessary (see the *`service`*`.SQL` properties below).

service.Discard.Interval

Synopsis

`service``.Discard.Interval=`*`num`*

Description

An IceStorm server detects when a subscriber to which it forwards events becomes non-functional and, at that point, stops delivery attempts to that subscriber for *`num`* seconds before trying to forward events to that subscriber again. The default value of this property is 60 seconds.

service.Election.ElectionTimeout

Synopsis

`service``.Election.ElectionTimeout=`*`num`*

Description

This property is used by a replicated IceStorm deployment (see Section 44.7). It specifies the interval in seconds at which a coordinator attempts to form larger groups of replicas. If not defined, the default value is 10.

service.Election.MasterTimeout

Synopsis

service.Election.MasterTimeout=*num*

Description

This property is used by a replicated IceStorm deployment (see Section 44.7). It specifies the interval in seconds at which a slave checks the status of the coordinator. If not defined, the default value is 10.

service.Election.ResponseTimeout

Synopsis

service.Election.ResponseTimeout=*num*

Description

This property is used by a replicated IceStorm deployment (see Section 44.7). It specifies the interval in seconds that a replica waits for replies to an invitation to form a larger group. Lower priority replicas wait for intervals inversely proportional to the maximum priority:

```
ResponseTimeout + ResponseTimeout * (max - pri)
```

If not defined, the default value is 10.

service.Flush.Timeout

Synopsis

service.Flush.Timeout=*num*

Description

Defines the interval in milliseconds with which batch reliability events are sent to subscribers. The default is 1000ms.

service.InstanceName

Synopsis

service.InstanceName=*name*

Description

Specifies an alternate identity category for all objects hosted by the IceStorm object adapters. If not specified, the default identity category is `IceStorm`.

service.Node.*name*

Synopsis

service.Node.*name*=*value*

Description

In a replicated deployment, IceStorm uses the adapter name *service*.Node for the replica node's object adapter (see Section 44.7). Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter.

service.NodeId

Synopsis

service.NodeId=*value*

Description

Specifies the node id of an IceStorm replica, where *value* is a non-negative integer. The node id is also used as the replica's priority, such that a larger value assigns higher priority to the replica. As described in Section 44.7, the replica with the highest priority becomes the coordinator of its group. This property must be defined for each replica.

service.Nodes.*id*

Synopsis

service.Nodes.*id*=*value*

Description

This property is used for a manual deployment of HA IceStorm (see Section 44.12.3), in which each of the replicas must be explicitly configured with the proxies of all other replicas. The value is a proxy for the replica with the given node id. A replica's object identity has the form *instance-name*/node*id*, such as DemoIceStorm/node2.

service.Publish.*name*

Synopsis

service.Publish.*name*=*value*

Description

IceStorm uses the adapter name *service*.Publish for the object adapter that processes incoming requests from publishers. Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter.

service.ReplicatedPublishEndpoints

Synopsis

service.ReplicatedPublishEndpoints=*value*

Description

This property is used for a manual deployment of HA IceStorm (see Section 44.12.3). It specifies the set of endpoints returned for the publisher proxy returned from IceStorm::Topic::getPublisher.

If this property is not defined, the publisher proxy returned by a topic instance points directly at that replica and, should the replica become unavailable, publishers will not transparently failover to other replicas.

service.ReplicatedTopicManagerEndpoints

Synopsis

service.ReplicatedTopicManagerEndpoints=*value*

Description

This property is used for a manual deployment of HA IceStorm (see Section 44.12.3). It specifies the set of endpoints used in proxies that refer to a replicated topic. This set of endpoints should contain the endpoints of each IceStorm replica.

For example, the operation `IceStorm::TopicManager::create` returns a proxy that contains this set of endpoints.

service.Send.Timeout

Synopsis

service`.Send.Timeout=`*num*

Description

IceStorm applies a send timeout when it forwards events to subscribers. The value of this property determines how long IceStorm will wait for forwarding of an event to complete. If an event cannot be forwarded within *num* milliseconds, the subscriber is considered dead and its subscription is cancelled. The default value is 60 seconds. Setting this property to a negative value disables timeouts.

service.SQL.DatabaseType

Synopsis

service`.SQL.DatabaseType=`*type*

Description

This property determines the type of the database for `QSqlDatabase::addDatabase`. Permissible values of *type* are `QSQLITE`, `QMSQL`, `QPSQL`, and `QODBC`.

service.SQL.DatabaseName

Synopsis

service`.SQL.Database=`*name*

Description

This property determines the name of the database for `QSqlDatabase::setDatabaseName`. For SQLite, *name* is the file name for the database. For MySQL and PostgreSQL, *name* is the database name. For SQL Server, *name* is the DSN name.

service.SQL.HostName

Synopsis

service.SQL.HostName=*name*

Description

This property determines the host name for `QSqlDatabase::setHostName`. For PostgreSQL and MySQL, *name* is the database host name. For SQL Server, *name* is the host name and server name separated by a blackslash, such as `localhost\SQLExpress`. For SQLite, this property has no effect and need not be set.

service.SQL.Port

Synopsis

service.SQL.Port=*num*

Description

This property determines the port number for `QSqlDatabase::setPort`.

service.SQL.UserName

Synopsis

service.SQL.UserName=*name*

Description

This property determines the user name for `QSqlDatabase::setUserName`. For SQLite, this property has no effect and need not be set.

service.SQL.Password

Synopsis

`service.SQL.Password=password`

Description

This property determines the password for `QSqlDatabase::setPassword`. For SQLite, this property has no effect and need not be set.

service.TopicManager.*name*

Synopsis

`service.TopicManager.name=value`

Description

IceStorm uses the adapter name `service.TopicManager` for the topic manager's object adapter. Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter.

service.Trace.Election

Synopsis

`service.Trace.Election=num`

Description

Trace activity related to elections:

0	No election trace (default).
1	Trace election activity.

service.Trace.Replication

Synopsis

`service.Trace.Replication=num`

Description

Trace activity related to replication:

0	No replication trace (default).
1	Trace replication activity.

service.Trace.Subscriber

Synopsis

`service.Trace.Subscriber=num`

Description

The subscriber trace level:

0	No subscriber trace (default).
1	Trace topic diagnostic information on subscription and unsubscription.
2	Like 1, but more verbose, including state transitions for a subscriber (such as going offline after a temporary network failure, and going online again after a successful retry, etc.).

service.Trace.Topic

Synopsis

`service.Trace.Topic=num`

Description

The topic trace level:

0	No topic trace (default).
1	Trace topic links, subscription, and unsubscription.
2	Like 1, but more verbose, including QoS information, and other diagnostic information.

service.Trace.TopicManager

Synopsis

`service.Trace.TopicManager=num`

Description

The topic manager trace level:

0	No topic manager trace (default).
1	Trace topic creation.

service.Transient

Synopsis

`service.Transient=num`

Description

If *num* is a value greater than zero, IceStorm runs in a fully transient mode in which no database is required. Replication is not supported in this mode. If not defined, the default value is zero.

IceStormAdmin.TopicManager.Default

Synopsis

`IceStormAdmin.TopicManager.Default=proxy`

Description

Defines the proxy for the default IceStorm topic manager. This property is used by **`icegridadmin`**. IceStorm applications may choose to use this property for their configuration as well.

IceStormAdmin.TopicManager.*name*

Synopsis

`IceStormAdmin.TopicManager.`*`name`*`=`*`proxy`*

Description

Defines a proxy for an IceStorm topic manager for **icegridadmin**. Properties with this pattern are used by **icestormadmin** if multiple topic managers are in use, for example:

```
IceStormAdmin.TopicManager.A=A/TopicManager:tcp -h x -p 9995
IceStormAdmin.TopicManager.B=Foo/TopicManager:tcp -h x -p 9995
IceStormAdmin.TopicManager.C=Bar/TopicManager:tcp -h x -p 9987
```

This sets the proxies for three topic managers. Not that *name* need not match the instance name of the corresponding topic manager—*name* simply serves as tag. With these property settings, the **icestormadmin** commands that accept a topic can now specify a topic manager other than the default topic manager that is configured with `IceStormAdmin.TopicManager.Default`. For example:

```
current Foo
create myTopic
create Bar/myOtherTopic
```

This sets the current topic manager to the one with instance name `Foo`; the first `create` command then creates the topic within that topic manager, whereas the second `create` command uses the topic manager with instance name `Bar`.

D.18 Glacier2 Properties

Glacier2.AddConnectionContext

Synopsis

`Glacier2.AddConnectionContext=`*`num`*

Description

If *num* is set to 1 or 2, Glacier2 adds a number of key–value pairs to the context that it sends with each request. If *num* is set to 1, these entries are adde to the context for all forwarded requests. If *num* is set to 2, the contexts are added only to

calls to `checkPermissions` and `authorize` on permission verifiers, and to calls to `create` on session managers.

If num is non-zero, Glacier2 adds the following context entries:

`_con.type`	The type of the connection as returned by `Connection::type`.
`_con.localAddress`	The local address (TCP and SSL only).
`_con.localPort`	The local port (TCP and SSL only).
`_con.remoteAddress`	The remote address (TCP and SSL only).
`_con.remotePort`	The remote port (TCP and SSL only).
`_con.cypher`	The cypher (SSL only).
`_con.peerCert`	The first certificate of the client certificate chain (SSL only).

The default value is zero, meaning that no context are added.

Glacier2.AddSSLContext

Synopsis

`Glacier2.AddSSLContext=`*`num`*

Description

For sessions created with `createSessionFromSecureConnection`, this property, when set to a value greater than zero, instructs Glacier2 to provide additional information in the context of each request:

`SSL.Active`	If the client established an SSL connection to the router, this entry is present and has the value `1`. This entry is not present if SSL was not used.
`SSL.Cipher`	A description of the ciphersuite negotiated for the SSL connection.
`SSL.Remote.Host`	The client's originating host name or address.
`SSL.Remote.Port`	The client's originating port number.

`SSL.Local.Host`	The router's local host name or address.
`SSL.Local.Port`	The router's local port number.
`SSL.PeerCert`	If the client supplied a certificate, this entry is present and contains the encoded certificate in PEM format.

Note that these SSL context entries are forwarded regardless of the setting of `Glacier2.Client.ForwardContext`.

If `Glacier2.AddSSLContext` is not defined, its default value is zero.

This property is deprecated and supported only for backward-compatibility. New applications should use `Glacier2.AddConnectionContext`.

Glacier2.AddUserToAllowCategories

Synopsis

`Glacier2.AddUserToAllowCategories=`*`num`*

Description

Specifies whether to add an authenticated user id to the `Glacier2.AllowCategories` property when creating of a new session. The legal values are shown below:

0	Do not add the user id (default).
1	Add the user id.
2	Add the user id with a leading underscore.

This property is deprecated and supported only for backward-compatibility. New applications should use `Glacier2.Filter.Category.AcceptUser`.

Glacier2.AllowCategories

Synopsis

`Glacier2.AllowCategories=`*`list`*

Description

Specifies a white space-separated list of identity categories. If this property is defined, then the Glacier2 router only allows requests to Ice objects with an identity that matches one of the categories from this list. If `Glacier2.AddUserToAllowCategories` is defined with a non-zero value, the router automatically adds the user id of each session to this list.

This property is deprecated and supported only for backward-compatibility. New applications should use `Glacier2.Filter.Category.Accept`.

Glacier2.Client.AlwaysBatch

Synopsis

`Glacier2.Client.AlwaysBatch=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router always batches queued oneway requests from clients to servers regardless of the value of their `_fwd` contexts. This property is only relevant when `Glacier2.Client.Buffered=1`. The default value is `0`.

Glacier2.Client.Buffered

Synopsis

`Glacier2.Client.Buffered=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router operates in buffered mode, in which incoming requests from clients are queued and processed in a separate thread. If *`num`* is set to zero, the router operates in unbuffered mode in which a request is forwarded in the same thread that received it. The default value is `1`. See Section 42.9 for more information.

Glacier2.Client.*name*

Synopsis

`Glacier2.Client.`*`name`*`=`*`value`*

Description

Glacier2 uses the adapter name `Glacier2.Client` for the object adapter that it provides to clients. Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter.

This adapter must be accessible to clients of Glacier2. Use of a secure transport for this adapter is highly recommended.

Note that `Glacier2.Registry.Client.Endpoints` controls the client endpoint for Glacier2. The port numbers 4063 (for TCP) and 4064 (for SSL) are reserved for Glacier2 by the Internet Assigned Numbers Authority (IANA).

Glacier2.Client.ForwardContext

Synopsis

`Glacier2.Client.ForwardContext=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router includes the context in forwarded requests from clients to servers. The default value is `0`.

Glacier2.Client.SleepTime

Synopsis

`Glacier2.Client.SleepTime=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router sleeps for the specified number of milliseconds after forwarding all queued requests from a client. This delay is useful for batched delivery because it makes it more likely for events to accumulate in a single batch. Similarly, if overrides are used, the delay makes it more likely for overrides to actually take effect. This property is only relevant when `Glacier2.Client.Buffered=1`. The default value is `0`.

Glacier2.Client.Trace.Override

Synopsis

`Glacier2.Client.Trace.Override=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router logs a trace message whenever a request was overridden. The default value is `0`.

Glacier2.Client.Trace.Reject

Synopsis

`Glacier2.Client.Trace.Reject=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router logs a trace message whenever the router's configured filters reject a client's request. The default value is `0`.

Glacier2.Client.Trace.Request

Synopsis

`Glacier2.Client.Trace.Request=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router logs a trace message for each request that is forwarded from a client. The default value is `0`.

Glacier2.CryptPasswords

Synopsis

`Glacier2.CryptPasswords=`*`file`*

Description

Specifies the file name of a Glacier2 access control list (see Section 42.6.1). Each line of the file must contain a user name and a password, separated by white space. The password must be a 13-character crypt-encoded string. This property is ignored if `Glacier2.PermissionsVerifier` is defined.

Glacier2.Filter.AdapterId.Accept

Synopsis

`Glacier2.Filter.AdapterId.Accept=`*`string`*

Description

Specifies a space-separated list of adapter identifiers. If defined, the Glacier2 router only allows requests to Ice objects with an adapter identifier that matches one of the entries in this list.

Identifiers that contain spaces must be enclosed in single or double quotes. Single or double quotes that appear within an identifier must be escaped with a leading backslash.

Glacier2.Filter.Address.Accept

Synopsis

`Glacier2.Filter.Address.Accept=`*`string`*

Description

Specifies a space-separated list of address–port pairs. When defined, the Glacier2 router only allows requests to Ice objects through proxies that contain network endpoint information that matches an address–port pair listed in this property. If not defined, the default value is `*`, which indicates that any network address is permitted. Requests accepted by this property may be rejected by the `Glacier2.Filter.Address.Reject` property.

Each pair is of the form *`address`*`:`*`port`*. The *`address`* or *`port`* number portion can include wildcards ('`*`') or value ranges or groups. Ranges and groups are in the form `[`*`value1`*`, `*`value2`*`, `*`value3`*`, ...]` and/or `[`*`value1`*`-`*`value2`*`]`. Wildcards, ranges, and groups may appear anywhere in the address–port pair string.

Glacier2.Filter.Address.Reject

Synopsis

`Glacier2.Filter.Address.Reject=`*`string`*

Description

Specifies a space-separated list of address–port pairs. When defined, the Glacier2 router rejects requests to Ice objects through proxies that contain network endpoint information that matches an address–port pair listed in this property. If not set, the Glacier2 router allows requests to any network address unless the `Glacier2.Filter.Address.Accept` property is set, in which case requests will be accepted or rejected based on the `Glacier2.Filter.Address.Accept` property. If both the `Glacier2.Filter.Address.Accept` and `Glacier2.Filter.Address.Reject` properties are defined, the `Glacier2.Filter.Address.Reject` property takes precedence.

Each pair is of the form *`address`*`:`*`port`*. The *`address`* or *`port`* number portion can include wildcards ('`*`') or value ranges or groups. Ranges and groups are in the form of `[`*`value1`*`, `*`value2`*`, `*`value3`*`, ...]` and/or `[`*`value1`*`-`*`value2`*`]`. Wildcards, ranges, and groups may appear anywhere in the address–port pair string.

Glacier2.Filter.Category.Accept

Synopsis

`Glacier2.Filter.Category.Accept=`*`string`*

Description

Specifies a space-separated list of identity categories. If defined, the Glacier2 router only allows requests to Ice objects with an identity that matches one of the categories in this list. If `Glacier2.Filter.CategoryAcceptUser` is defined with a non-zero value, the router automatically adds the user name of each session to this list.

Categories that contain spaces must be enclosed in single or double quotes. Single or double quotes that appear within a category must be escaped with a leading backslash.

Glacier2.Filter.Category.AcceptUser

Synopsis

`Glacier2.Filter.Category.AcceptUser=`*`num`*

Description

Specifies whether to add an authenticated user id to the `Glacier2.Filter.Category.Accept` property when creating of a new session. The legal values are shown below:

0	Do not add the user id (default).
1	Add the user id.
2	Add the user id with a leading underscore.

Glacier2.Filter.Identity.Accept

Synopsis

`Glacier2.Filter.Identity.Accept=`*`string`*

Description

Specifies a space-separated list of identities. If defined, the Glacier2 router only allows requests to Ice objects with an identity that matches one of the entries in this list.

Identities that contain spaces must be enclosed in single or double quotes. Single or double quotes that appear within an identity must be escaped with a leading backslash.

Glacier2.Filter.ProxySizeMax

Synopsis

`Glacier2.Filter.ProxySizeMax=`*`num`*

Description

If set, the Glacier2 router rejects requests whose stringified proxies are longer than *num*. This helps secure the system against attack. If not set, Glacier2 will accept requests using proxies of any length.

Glacier2.InstanceName

Synopsis

`Glacier2.InstanceName=`*`name`*

Description

Specifies a default identity category for the Glacier2 objects. If defined, the identity of the Glacier2 admin interface becomes *`name`*`/admin` and the identity of the Glacier2 router interface becomes *`name`*`/router`.

If not defined, the default value is `Glacier2`.

Glacier2.PermissionsVerifier

Synopsis

`Glacier2.PermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::PermissionsVerifier` interface (see Section 42.6.1). The router invokes this proxy to validate the user name and password of each new session. Sessions created from a secure connection are verified by the object specified in `Glacier2.SSLPermissionsVerifier`. For simple configurations, you can specify the name of a password file using `Glacier2.CryptPasswords`.

Glacier2 supplies a "null" permissions verifier object that accepts any username and password combination for situations in which no authentication is necessary. To enable this verifier, set the property value to *`instance`*`/NullPermissionsVerifier`, where *`instance`* is the value of `Glacier2.InstanceName`.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

Glacier2.ReturnClientProxy

Synopsis

`Glacier2.ReturnClientProxy=`*`num`*

Description

If *num* is a value greater than zero, Glacier2 maintains backward compatibility with clients using Ice versions prior to 3.2.0. In this case you should also define `Glacier2.Client.PublishedEndpoints` to specify the endpoints that clients should use to contact the router. For example, if the Glacier2 router resides behind a network firewall, the `Glacier2.Client.PublishedEndpoints` property should specify the firewall's external endpoints.

If not defined, the default value is zero.

Glacier2.RoutingTable.MaxSize

Synopsis

`Glacier2.RoutingTable.MaxSize=`*`num`*

Description

This property sets the size of the router's routing table to *num* entries. If more proxies are added to the table than this value, proxies are evicted from the table on a least-recently used basis.

Clients based on Ice version 3.1 and later automatically retry operation calls on evicted proxies and transparently re-add such proxies to the table. Clients based on Ice versions earlier than 3.1 receive an `ObjectNotExistException` for invocations on evicted proxies. For such older clients, *num* must be set to a sufficiently large value to prevent these clients from failing.

The default size of the routing table is 1000.

Glacier2.Server.*name*

Synopsis

`Glacier2.Server.`*`name`*`=`*`value`*

Description

Glacier2 uses the adapter name `Glacier2.Server` for the object adapter that it provides to servers. Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter.

This adapter provides access to the `SessionControl` interface and must be accessible to servers that call back to router clients.

Glacier2.Server.AlwaysBatch

Synopsis

`Glacier2.Server.AlwaysBatch=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router always batches queued oneway requests from servers to clients regardless of the value of their `_fwd` contexts. This property is only relevant when `Glacier2.Server.Buffered=1`. The default value is `0`.

Glacier2.Server.Buffered

Synopsis

`Glacier2.Server.Buffered=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router operates in buffered mode, in which incoming requests from servers are queued and processed in a separate thread. If *`num`* is set to zero, the router operates in unbuffered mode in which a request is forwarded in the same thread that received it. The default value is `1`. See Section 42.9 for more information.

Glacier2.Server.ForwardContext

Synopsis

`Glacier2.Server.ForwardContext=`*`num`*

Description

If *num* is set to a value larger than zero, the Glacier2 router includes the context in forwarded requests from servers to clients. The default value is `0`.

Glacier2.Server.SleepTime

Synopsis

`Glacier2.Server.SleepTime=`*num*

Description

If *num* is set to a value larger than zero, the Glacier2 router sleeps for the specified number of milliseconds after forwarding all queued requests from a server. This delay is useful for batched delivery because it makes it more likely for events to accumulate in a single batch. Similarly, if overrides are used, the delay makes it more likely for overrides to actually take effect. This property is only relevant when `Glacier2.Server.Buffered=1`. The default value is `0`.

Glacier2.Server.Trace.Override

Synopsis

`Glacier2.Server.Trace.Override=`*num*

Description

If *num* is set to a value larger than zero, the Glacier2 router logs a trace message whenever a request is overridden. The default value is `0`.

Glacier2.Server.Trace.Request

Synopsis

`Glacier2.Server.Trace.Request=`*num*

Description

If *num* is set to a value larger than zero, the Glacier2 router logs a trace message for each request that is forwarded from a server. The default value is `0`.

Glacier2.SessionManager

Synopsis

`Glacier2.SessionManager=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SessionManager` interface. The router invokes this proxy to create a new session for a client, but only after the router validates the client's user name and password.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

Glacier2.SessionTimeout

Synopsis

`Glacier2.SessionTimeout=`*`num`*

Description

If *`num`* is set to a value larger than zero, a client's session with the Glacier2 router expires after the specified *`num`* seconds of inactivity. The default value is `0`, meaning sessions do not expire due to inactivity.

It is important to choose *`num`* such that client sessions do not expire prematurely.

Setting the session timeout enables active connection management of client connections (by setting `Glacier2.Client.ACM`). By default, the ACM timeout is set to twice the session timeout. If no session timeout is defined, ACM is disabled.

Glacier2.SSLPermissionsVerifier

Synopsis

`Glacier2.SSLPermissionsVerifier=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SSLPermissionsVerifier` interface (see Section 42.6.1). The router invokes this proxy to verify the credentials of clients that attempt to create a session from a secure

connection. Sessions created with a user name and password are verified by the object specified in `Glacier2.PermissionsVerifier`.

Glacier2 supplies a "null" permissions verifier object that accepts the credentials of any client for situations in which no authentication is necessary. To enable this verifier, set the property value to *`instance`*`/NullSSLPermissionsVerifier`, where *`instance`* is the value of `Glacier2.InstanceName`.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

Glacier2.SSLSessionManager

Synopsis

`Glacier2.SSLSessionManager=`*`proxy`*

Description

Specifies the proxy of an object that implements the `Glacier2::SSLSessionManager` interface. The router invokes this proxy to create a new session for a client that has called `createSessionFromSecureConnection`.

As a proxy property, you can configure additional aspects of the proxy using the properties described in Section D.9.

Glacier2.Trace.RoutingTable

Synopsis

`Glacier2.Trace.RoutingTable=`*`num`*

Description

The routing table trace level:

0	No routing table trace (default).
1	Logs a message for each proxy that is added to the routing table.
2	Logs a message for each proxy that is evicted from the routing table (see `Glacier2.RoutingTable.MaxSize`).
3	Combines the output for trace levels 1 and 2.

Glacier2.Trace.Session

Synopsis

`Glacier2.Trace.Session=`*`num`*

Description

If *`num`* is set to a value larger than zero, the Glacier2 router logs trace messages about session-related activities. The default value is `0`.

D.19 Freeze Properties

Freeze.DbEnv.*env-name*.CheckpointPeriod

Synopsis

`Freeze.DbEnv.`*`env-name`*`.CheckpointPeriod=`*`num`*

Description

Every Berkeley DB environment created by Freeze has an associated thread that checkpoints this environment every *`num`* seconds. If *`num`* is less than `0`, no checkpointing is performed. The default is 120 seconds.

Freeze.DbEnv.*env-name*.DbHome

Synopsis

`Freeze.DbEnv.`*`env-name`*`.DbHome=`*`db-home`*

Description

Defines the home directory of this Freeze database environment. The default is *`env-name`*.

Freeze.DbEnv.*env-name*.DbPrivate

Synopsis

`Freeze.DbEnv.`*`env-name`*`.DbPrivate=`*`num`*

Description

If *num* is set to a value larger than zero, Freeze instructs Berkeley DB to use process-private memory instead of shared memory. The default value is `1`. Set this property to `0` in order to run a FreezeScript utility, or a Berkeley DB utility such as **`db_archive`**, on a running environment.

Freeze.DbEnv.*env-name*.DbRecoverFatal

Synopsis

`Freeze.DbEnv.`*`env-name`*`.DbRecoverFatal=`*`num`*

Description

If *num* is set to a value larger than zero, fatal recovery is performed when the environment is opened. The default value is `0`.

Freeze.DbEnv.*env-name*.LockFile

Synopsis

`Freeze.DbEnv.`*`env-name`*`.LockFile=`*`num`*

Description

If *num* is set to a value larger than zero, Freeze creates a lock file in the database environment to prevent other processes from opening the environment. The default value is `1`.

Note that applications should not normally disable the lock file because simultaneous access to the same environment by multiple processes can lead to data corruption.

FreezeScript utilities automatically disable the lock file when `Freeze.DbEnv.`*`env-name`*`.DbPrivate` is set to zero.

Freeze.DbEnv.*env-name*.OldLogsAutoDelete

Synopsis

`Freeze.DbEnv.`*`env-name`*`.OldLogsAutoDelete=`*`num`*

Description

If *num* is set to a value larger than zero, old transactional logs no longer in use are deleted after each periodic checkpoint (see `Freeze.DbEnv.`*`env-name`*`.CheckpointPeriod`). The default value is `1`.

Freeze.DbEnv.*env-name*.PeriodicCheckpointMinSize

Synopsis

`Freeze.DbEnv.`*`env-name`*`.PeriodicCheckpointMinSize=`*`num`*

Description

num is the minimum size in kilobytes for the periodic checkpoint (see `Freeze.DbEnv.`*`env-name`*`.CheckpointPeriod`). This value is passed to Berkeley DB's `checkpoint` function. The default is `0` (which means no minimum).

Freeze.Evictor.*env-name.filename.name*.BtreeMinKey

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.`*`name`*`.BtreeMinKey=`*`num`*

Description

name may represent a database name or an index name. This property sets the B-tree minkey of the corresponding Berkeley DB database. *num* is ignored if it is less than 2. Please refer to the Berkeley DB documentation for details.

Freeze.Evictor.*env-name.filename.name*.Checksum

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.Checksum=`*`num`*

Description

If *num* is greater than 0, checksums on the corresponding Berkeley DB database(s) are enabled. Please refer to the Berkeley DB documentation for details.

Freeze.Evictor.*env-name*.*filename*.MaxTxSize

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.MaxTxSize=`*`num`*

Description

Freeze uses a background thread to save updates to the database. Transactions are used to save many facets together. *num* defines the maximum number of facets saved per transaction. The default is `10 * SaveSizeTrigger` (see `Freeze.Evictor.`*`env-name`*`.`*`filename`*`.SaveSizeTrigger`); if this value is negative, the actual value is set to `100`.

Freeze.Evictor.*env-name*.*filename*.PageSize

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.PageSize=`*`num`*

Description

If *num* is greater than 0, it sets the page size of the corresponding Berkeley DB database(s). Please refer to the Berkeley DB documentation for details.

Freeze.Evictor.*env-name*.*filename*.PopulateEmptyIndices

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.PopulateEmptyIndices=`*`num`*

Description

When *num* is not `0` and you create an evictor with one or more empty indexes, the `createBackgroundSaveEvictor` or `createTransactionalEvictor` call will populate these indexes by iterating over all the corresponding facets. This is particularly useful after transforming a Freeze evictor with FreezeScript, since FreezeScript does not transform indexes; however this can significantly slow down the creation of the evictor if you have an empty index because none of the facets currently in the database match the type of this index. The default value for this property is `0`.

Freeze.Evictor.*env-name*.*filename*.RollbackOnUserException

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.RollbackOnUserException=`*`num`*

Description

If *num* is non-zero, a transactional evictor rolls back its transaction if the outcome of the dispatch is a user exception. If *num* is 0 (the default), the transactional evictor commits the transaction.

Freeze.Evictor.*env-name*.*filename*.SavePeriod

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.SavePeriod=`*`num`*

Description

Freeze uses a background thread to save updates to the database. After *num* milliseconds without saving, if any facet is created, modified, or destroyed, this background thread wakes up to save these facets. When num is `0`, there is no periodic saving. The default is `60000`.

Freeze.Evictor.*env-name*.*filename*.SaveSizeTrigger

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.SaveSizeTrigger=`*`num`*

Description

Freeze uses a background thread to save updates to the database. When num is `0` or positive, as soon as *num* or more facets have been created, modified, or destroyed, this background thread wakes up to save them. When *num* is negative, there is no size trigger. The default is `10`.

Freeze.Evictor.*env-name*.*filename*.StreamTimeout

Synopsis

`Freeze.Evictor.`*`env-name`*`.`*`filename`*`.StreamTimeout=`*`num`*

Description

When the saving thread saves an object, it needs to lock this object in order to get a consistent copy of the object's state. If the lock cannot be acquired within *num* seconds, a fatal error is generated. If a fatal error callback was registered by the application, this callback is called; otherwise the program is terminated immediately. When *num* is `0` or negative, there is no timeout. The default value is `0`.

Freeze.Map.*name*.BtreeMinKey

Synopsis

`Freeze.Map.`*`name`*`.BtreeMinKey=`*`num`*

Description

name may represent a database name or an index name. This property sets the **-**tree minkey of the corresponding Berkeley DB database. *num* is ignored if it is less than 2. Please refer to the Berkeley DB documentation for details.

Freeze.Map.*name*.Checksum

Synopsis

`Freeze.Map.`*`name`*`.Checksum=`*`num`*

Description

name may represent a database name or an index name. If *num* is greater than 0, checksums for the corresponding Berkeley DB database are enabled. Please refer to the Berkeley DB documentation for details.

Freeze.Map.*name*.PageSize

Synopsis

`Freeze.Map.`*`name`*`.PageSize=`*`num`*

Description

name may represent a database name or an index name. If *num* is greater than 0, it sets the page size of the corresponding Berkeley DB database. Please refer to the Berkeley DB documentation for details.

Freeze.Trace.DbEnv

Synopsis

`Freeze.Trace.DbEnv=`*`num`*

Description

The Freeze database environment activity trace level:

0	No database environment activity trace (default).
1	Trace database open and close.
2	Also trace checkpoints and the removal of old log files.

Freeze.Trace.Evictor

Synopsis

`Freeze.Trace.Evictor=`*`num`*

Description

The Freeze evictor activity trace level:

0	No evictor activity trace (default).
1	Trace Ice object and facet creation and destruction, facet streaming time, facet saving time, object eviction (every 50 objects) and evictor deactivation.

2	Also trace object lookups, and all object evictions.
3	Also trace object retrieval from the database.

Freeze.Trace.Map

Synopsis

`Freeze.Trace.Map=`*`num`*

Description

The Freeze map activity trace level:

0	No map activity trace (default).
1	Trace database open and close.
2	Also trace iterator and transaction operations, and reference counting of the underlying database.

Freeze.Trace.Transaction

Synopsis

`Freeze.Trace.Transaction=`*`num`*

Description

The Freeze transaction activity trace level:

0	No transaction activity trace (default).
1	Trace transaction IDs and commit and rollback activity.

Freeze.Warn.Deadlocks

Synopsis

`Freeze.Warn.Deadlocks=`*`num`*

Description

If *num* is set to a value larger than zero, Freeze logs a warning message when a deadlock occur. The default value is `0`.

Freeze.Warn.Rollback

Synopsis

`Freeze.Warn.Deadlocks=`*`num`*

Description

If *num* is set to a value larger than zero, Freeze logs a warning message when it rolls back a transaction that goes out of scope together with its associated connection. The default value is `1`. (C++ only)

D.20 IcePatch2 Properties

IcePatch2.*name*

Synopsis

`IcePatch2.`*`name`*`=`*`value`*

Description

IcePatch2 uses the adapter name `IcePatch2` for the server. Therefore, all the adapter properties detailed in Section D.4 can be used to configure this adapter.

Note that the property `IcePatch2.Endpoints` must be set for IcePatch2 clients, so they can locate the IcePatch2 server.

IcePatch2.ChunkSize

Synopsis

`IcePatch2.ChunkSize=`*`kilobytes`*

Description

The IcePatch2 client uses this property to determine how many kilobytes are retrieved with each call to `getFileCompressed`.

The default value is `100`.

IcePatch2.Directory

Synopsis

`IcePatch2.Directory=`*`dir`*

Description

The IcePatch2 server uses this property to determine the data directory if no data directory is specified on the command line.

This property is also used by IcePatch2 clients to determine the local data directory.

IcePatch2.InstanceName

Synopsis

`IcePatch2.InstanceName=`*`name`*

Description

Specifies the identity category for well-known IcePatch2 objects. If defined, the identity of the `IcePatch2::Admin` interface becomes *`name`*`/admin` and the identity of the `IcePatch2::FileServer` interface becomes *`name`*`/server`.

If not defined, the default value is `IcePatch2`.

IcePatch2.Remove

Synopsis

`IcePatch2.Remove=`*`num`*

Description

This property determines whether IcePatch2 clients delete files that exist locally, but not on the server. A negative or zero value prevents removal of files. A value

of `1` enables removal and causes the client to halt with an error if removal of a file fails. A value of `2` or greater also enables removal, but causes the client to silently ignore errors during removal.

The default value is `1`.

IcePatch2.Thorough

Synopsis

`IcePatch2.Thorough=`*`num`*

Description

This property determines whether IcePatch2 clients recompute checksums. Any value greater than zero is interpreted as true. The default value is `0` (false).

Appendix E
Proxies and Endpoints

E.1 Proxies

Synopsis

identity `-f` *facet* `-t -o -O -d -D -s @` *adapter_id* `:` *endpoints*

Description

A stringified proxy consists of an identity, proxy options, and an optional object adapter identifier or endpoint list. White space (the space, tab (`\t`), line feed (`\n`), and carriage return (`\r`) characters) act as token delimiters; if a white space character appears as part of a component of a stringified proxy (such as the identity), it must be quoted or escaped as described below.

A proxy containing an identity with no endpoints, or an identity with an object adapter identifier, represents an indirect proxy that will be resolved using the Ice locator (see the `Ice.Default.Locator` property).

Proxy options configure the invocation mode:

`-f` *facet*	Select a facet of the Ice object.
`-t`	Configures the proxy for twoway invocations (default).
`-o`	Configures the proxy for oneway invocations.

`-O`	Configures the proxy for batch oneway invocations.
`-d`	Configures the proxy for datagram invocations.
`-D`	Configures the proxy for batch datagram invocations.
`-s`	Configures the proxy for secure invocations.

The proxy options `-t`, `-o`, `-O`, `-d`, and `-D` are mutually exclusive.

The object identity `identity` is structured as `[`*`category/`*`]`*`name`*, where the *`category`* component and slash separator are optional. If *`identity`* contains white space or either of the characters `:` or `@`, it must be enclosed in single or double quotes. The *`category`* and *`name`* components are UTF-8 strings that use the encoding described below. Any occurrence of a slash (`/`) in *`category`* or *`name`* must be escaped with a backslash (i.e., `\/`).

The *`facet`* argument of the `-f` option represents a facet name. If *`facet`* contains white space, it must be enclosed in single or double quotes. A facet name is a UTF-8 string that uses the encoding described below.

The object adapter identifier *`adapter_id`* is a UTF-8 string that uses the encoding described below. If *`adapter_id`* contains white space, it must be enclosed in single or double quotes.

Single or double quotes can be used to prevent white space characters from being interpreted as delimiters. Double quotes prevent interpretation of a single quote a as an opening or closing quote, for example:

```
"a string with a ' quote"
```

Single quotes prevent interpretation of a double quote as an opening or closing quote. For example:

```
'a string with a " quote'
```

Escape sequences such as `\b` are interpreted within single and double quotes.

UTF-8 strings are encoded using ASCII characters for the ordinal range 32–126 (inclusive). Characters outside this range must be encoded using escape sequences (`\b`, `\f`, `\n`, `\r`, `\t`) or octal notation (e.g., `\007`). Single and double quotes can be escaped using a backslash, as can the backslash itself (`\\`).

If *`endpoints`* are specified, they must be separated with a colon (`:`) and formatted as described on page 1963. The order of endpoints in the stringified proxy is not necessarily the order in which connections are attempted during binding: when a stringified proxy is converted into a proxy instance, by default, the endpoint list is randomized as a form of load balancing. You can change this

this default behavior by setting properties (see the `Ice.Default.EndpointSelection` and *`name`*`.Endpoint.Selection` properties in Appendix D).

If the `-s` option is specified, only those endpoints that support secure invocations are considered during binding. If no valid endpoints are found, the application receives `Ice::NoEndpointException`.

Otherwise, if the `-s` option is not specified, the endpoint list is ordered so that non-secure endpoints have priority over secure endpoints during binding. In other words, connections are attempted on all non-secure endpoints before any secure endpoints are attempted.

If an unknown option is specified, or the stringified proxy is malformed, the application receives `Ice::ProxyParseException`. If an endpoint is malformed, the application receives `Ice::EndpointParseException`.

E.2 Endpoints

Synopsis

`endpoint : endpoint`

Description

An endpoint list comprises one or more endpoints separated by a colon (`:`).[1] An endpoint has the following format:

`protocol option`

The supported protocols are `tcp`, `udp`, `ssl`, and `default`. If `default` is used, it is replaced by the value of the `Ice.Default.Protocol` property. If an endpoint is malformed, or an unknown protocol is specified, the application receives `Ice::EndpointParseException`. The `ssl` protocol is only available if the IceSSL plug-in is installed.

Ice uses endpoints for two similar but distinct purposes:

1. In a client context (that is, in a proxy), endpoints determine how Ice establishes a connection to a server.
2. In a server context (that is, in an object adapter's configuration), endpoints define the addresses and transports over which new incoming connections are

1. See Section 32.4.6 for examples.

accepted. These endpoints are also embedded in the proxies created by the object adapter, unless a separate set of "published" endpoints are explicitly configured.

The sections that follow discuss the addressing component of endpoints, as well as the protocols and their supported options.

Addressing

Synopsis

```
host : hostname | x.x.x.x (IPv4)
host : hostname | ":x:x:x:x:x:x:x" (IPv6)
```

Description

Ice supports Internet Protocol (IP) versions 4 and 6 in all language mappings[2]. Support for these protocols is configured using the properties `Ice.IPv4` (enabled by default) and `Ice.IPv6` (disabled by default).

In the endpoint descriptions below, the *`host`* parameter represents either a host name that is resolved via the Domain Name System (DNS), an IPv4 address in dotted quad notation, or an IPv6 address in 128-bit hexadecimal format and enclosed in double quotes. Due to limitation of the DNS infrastructure, host and domain names are restricted to the ASCII character set.

The presence (or absence) of the *`host`* parameter has a significant influence on the behavior of the Ice run time. The table below describes these semantics:

Value	Client Semantics	Server Semantics
None	If *`host`* is not specified in a proxy, Ice uses the value of the `Ice.Default.Host` property. If that property is not defined, outgoing connections are only attempted over loopback interfaces.	If *`host`* is not specified in an object adapter endpoint, Ice uses the value of the `Ice.Default.Host` property. If that property is not defined, the adapter behaves as if the wildcard symbol `*` was specified (see below).

2. IPv6 is not supported when using Ice for Java on Windows due to a limitation in the JVM.

Host name	The host name is resolved via DNS. Outgoing connections are attempted to each address returned by the DNS query.	The host name is resolved via DNS, and the object adapter listens on the network interfaces corresponding to each address returned by the DNS query. The specified host name is embedded in proxies created by the adapter.
IPv4 address	An outgoing connection is attempted to the given address.	The object adapter listens on the network interface corresponding to the address. The specified address is embedded in proxies created by the adapter.
IPv6 address	An outgoing connection is attempted to the given address.	The object adapter listens on the network interface corresponding to the address. The specified address is embedded in proxies created by the adapter.
`0.0.0.0` (IPv4)	A “wildcard” IPv4 address that causes Ice to try all local interfaces when establishing an outgoing connection.	Equivalent to `*` (see below).
`"::"` (IPv6)	A “wildcard” IPv6 address that causes Ice to try all local interfaces when establishing an outgoing connection.	Equivalent to `*` (see below).
`*` (IPv4, IPv6)	Not supported in proxies.	The adapter listens on all network interfaces (including the loopback interface), that is, binds to `INADDR_ANY` for the enabled protocols (IPv4 and/or IPv6). Endpoints for all addresses except loopback and IPv6 link-local are published in proxies (unless loopback is the only available interface, in which case only loopback is published). Using Mono, proxies created by an object adapter listening on the IPv6 wildcard address contain only the IPv6 loopback address unless published endpoints are configured.

There is one additional benefit in specifying a wildcard address for *host* (or not specifying it at all) in an object adapter's endpoint: if the list of network interfaces on a host may change while the application is running, using a wildcard address for *host* ensures that the object adapter automatically includes the updated interfaces. Note however that the list of published endpoints is not changed automatically; see page 938 for more information. For diagnostic purposes, you can set the configuration property `Ice.Trace.Network=3` to cause Ice to log the current list of local addresses that it is substituting for the wildcard address.

When IPv4 and IPv6 are enabled, an object adapter endpoint that uses an IPv6 (or wildcard) address can accept both IPv4 and IPv6 connections. This is true for all supported platforms except Windows XP and Windows Server 2003, where you must define separate IPv4 and IPv6 endpoints if you want the object adapter to accept both types of connections.

Java Notes

Java's default network stack always accepts both IPv4 and IPv6 connections regardless of the settings of `Ice.IPv4` and `Ice.IPv6`. You can configure the Java run time to use only IPv4 by starting your application with the following JVM option:

```
java -Djava.net.preferIPv4Stack=true ...
```

TCP Endpoint

Synopsis

`tcp -h` *host* `-p` *port* `-t` *timeout* `-z`

Description

A `tcp` endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
`-h` *host*	Specifies the host name or IP address of the endpoint. If not specified, the value of `Ice.Default.Host` is used instead.	See page 1964.	See page 1964.

`-p` *`port`*	Specifies the port number of the endpoint.	Determines the port to which a connection attempt is made (required).	The port will be selected by the operating system if this option is not specified or *`port`* is zero.
`-t` *`timeout`*	Specifies the endpoint timeout in milliseconds.	If *`timeout`* is greater than zero, it specifies the timeout used by the client to open or close connections and to read or write data. It also specifies how long the run time waits for an invocation to complete. If a timeout occurs, the application receives `Ice::TimeoutException`.	If *`timeout`* is greater than zero, it specifies the timeout used by the server to accept or close connections and to read or write data (see page 938 and Section 32.13). *`timeout`* also controls the timeout that is published in proxies created by the object adapter.
`-z`	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compression is advertised in proxies created by the adapter.

UDP Endpoint

Synopsis

```
udp -v major.minor -e major.minor -h host -p port -z --ttl TTL
    --interface INTF
```

Description

A `udp` endpoint supports either unicast or multicast delivery; the address resolved by the *`host`* argument determines the delivery mode. To use multicast in IPv4, select an IP address in the range `233.0.0.0` to `239.255.255.255`. In IPv6, use an address that begins with `ff`, such as `ff01::1:1`.

A `udp` endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
`-v` *`major.minor`*	Specifies the protocol major and highest minor version number to be used for this endpoint. If not specified, the protocol major version and highest supported minor version of the client-side Ice run time is used.	Determines the protocol major version and highest minor version used by the client side when sending messages to this endpoint. The protocol major version number must match the protocol major version number of the server; the protocol minor version number must not be higher than the highest minor version number supported by the server.	Determines the protocol major version and highest minor version advertised by the server side for this endpoint. The protocol major version number must match the protocol major version number of the server; the protocol minor version number must not be higher than the highest minor version number supported by the server.

`-e` *`major.minor`*	Specifies the encoding major and highest minor version number to be used for this endpoint. If not specified, the encoding major version and highest supported minor version of the client-side Ice run time is used.	Determines the encoding major version and highest minor version used by the client side when sending messages to this endpoint. The encoding major version number must match the encoding major version number of the server; the encoding minor version number must not be higher than the highest minor version number supported by the server.	Determines the encoding version and highest minor version advertised by the server side for this endpoint. The protocol major version number must match the protocol major version number of the server; the protocol minor version number must not be higher than the highest minor version number supported by the server.
`-h` *`host`*	Specifies the host name or IP address of the endpoint. If not specified, the value of `Ice.Default.Host` is used instead.	See page 1964.	See page 1964.
`-p` *`port`*	Specifies the port number of the endpoint.	Determines the port to which datagrams are sent (required).	The port will be selected by the operating system if this option is not specified or port is zero.
`-z`	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compression is advertised in proxies created by the adapter.

`--ttl` *TTL*	Specifies the time-to-live (also known as "hops") of multicast messages.	Determines whether multicast messages are forwarded beyond the local network. If not specified, or the value of *TTL* is `-1`, multicast messages are not forwarded. The maximum value is `255`.	N/A
`--interface` *INTF*	Specifies the network interface or group for multicast messages (see below).	Selects the network interface for outgoing multicast messages. If not specified, multicast messages are sent using the default interface.	Selects the network interface to use when joining the multicast group. If not specified, the group is joined on the default network interface.

Multicast Interfaces

When *host* denotes a multicast address, the `--interface` *INTF* option selects a particular network interface to be used for communication. The format of *INTF* depends on the language and IP version:

- C++ and .NET (IPv4)

  INTF can be an interface name, such as `eth0`, or an IP address. Interface names on Windows may contain spaces, such as `Local Area Connection`, therefore they must be enclosed in double quotes.

- C++ and .NET (IPv6)

  INTF can be an interface name, such as `eth0`, or an interface index.

- Java

  INTF can be an interface name, such as `eth0`, or an IP address. On Windows, Java maps interface names to Unix-style nicknames.

SSL Endpoint

Synopsis

`ssl -h` *host* `-p` *port* `-t` *timeout* `-z`

Description

An `ssl` endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
`-h` *`host`*	Specifies the host name or IP address of the endpoint. If not specified, the value of `Ice.Default.Host` is used instead.	See page 1964.	See page 1964.
`-p` *`port`*	Specifies the port number of the endpoint.	Determines the port to which a connection attempt is made (required).	The port will be selected by the operating system if this option is not specified or port is zero.
`-t` *`timeout`*	Specifies the endpoint timeout in milliseconds.	If *`timeout`* is greater than zero, it specifies the timeout used by the client to open or close connections and to read or write data. It also specifies how long the run time waits for an invocation to complete. If a timeout occurs, the application receives `Ice::TimeoutException`.	If *`timeout`* is greater than zero, it specifies the timeout used by the server to accept or close connections and to read or write data (see page 938 and Section 32.13). *`timeout`* also controls the timeout that is published in proxies created by the object adapter.
`-z`	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compression is advertised in proxies created by the adapter.

Opaque Endpoint

Synopsis

`opaque -t type -v value`

Description

Proxies can contain endpoints that are not universally understood by Ice processes. For example, a proxy can contain an SSL endpoint; if that proxy is marshaled to a receiver without the IceSSL plug-in, the SSL endpoint does not make sense to the receiver.

Ice preserves such unknown endpoints when they are received over the wire. For the preceding example, if the receiver remarshals the proxy and sends it back to an Ice process that does have the IceSSL plug-in, that process can invoke on the proxy using its SSL transport. This mechanism allows proxies containing endpoints for arbitrary transports to pass through processes that do not understand these endpoints without losing information.

If an Ice process stringifies a proxy containing an unknown endpoint, it writes the endpoint as an opaque endpoint. For example:

```
opaque -t 2 -v CTEyNy4wLjAuMREnAAD/////AA==
```

This is how a process without the IceSSL plug-in stringifies an SSL endpoint. When a process with the IceSSL plug-in unstringifies this endpoint and converts it back into a string, it produces:

```
ssl -h 127.0.0.1 -p 10001
```

An `opaque` endpoint supports the following options:

Option	Description
`-t` *`type`*	Specifies the transport for the endpoint. Transports are indicated by positive integers (1 for TCP, 2 for SSL, and 3 for UDP).
`-v` *`value`*	Specifies the marshaled encoding of the endpoint (including its enclosing encapsulation) in base-64 encoding.

Exactly one each of the `-t` and `-v` options must be present in an opaque endpoint.

Appendix F
The C++ Utility Library

F.1 Introduction

Ice for C++ includes a number of utility classes and functions in the `IceUtil` namespace. This appendix summarizes the contents of `IceUtil` for reference. Many of the classes and functions in `IceUtil` are documented elsewhere in this manual so, where appropriate, the sections here simply reference the relevant sections.

F.2 `AbstractMutex`

`AbstractMutex` defines a mutex base interface used by the Freeze `BackgroundSaveEvictor` (see Section 39.3.9). The interface allows the evictor to synchronize with servants that are stored in a Freeze database. The class has the following definition:

```
class AbstractMutex {
public:
    typedef LockT<AbstractMutex> Lock;
    typedef TryLockT<AbstractMutex> TryLock;

    virtual ~AbstractMutex();
```

```
    virtual void lock() const = 0;
    virtual void unlock() const = 0;
    virtual bool tryLock() const = 0;
};
```

This class definition is provided in `IceUtil/AbstractMutex.h`. The same header file also defines a few template implementation classes that specialize `AbstractMutex`:

- `AbstractMutexI`

  This template class implements `AbstractMutex` by forwarding all member functions to its template argument:

  ```
  template <typename T>
  class AbstractMutexI : public AbstractMutex, public T {
  public:
      typedef LockT<AbstractMutexI> Lock;
      typedef TryLockT<AbstractMutexI> TryLock;

      virtual void lock() const {
          T::lock();
      }

      virtual void unlock() const {
          T::unlock();
      }

      virtual bool tryLock() const {
          return T::tryLock();
      }

      virtual ~AbstractMutexI() {}
  };
  ```

- `AbstractMutexReadI`

  This template class implements a read lock by forwarding the `lock` and `tryLock` functions to the `readLock` and `tryReadLock` functions of its template argument:

  ```
  template <typename T>
  class AbstractMutexReadI : public AbstractMutex, public T {
  public:
      typedef LockT<AbstractMutexReadI> Lock;
      typedef TryLockT<AbstractMutexReadI> TryLock;

      virtual void lock() const {
  ```

```
        T::readLock();
    }

    virtual void unlock() const {
        T::unlock();
    }

    virtual bool tryLock() const {
        return T::tryReadLock();
    }

    virtual ~AbstractMutexReadI() {}
};
```

- `AbstractMutexWriteI`

  This template class implements a write lock by forwarding the `lock` and `tryLock` functions to the `writeLock` and `tryWriteLock` functions of its template argument:

```
template <typename T>
class AbstractMutexWriteI : public AbstractMutex, public T {
public:
    typedef LockT<AbstractMutexWriteI> Lock;
    typedef TryLockT<AbstractMutexWriteI> TryLock;

    virtual void lock() const {
        T::writeLock();
    }

    virtual void unlock() const {
        T::unlock();
    }

    virtual bool tryLock() const {
        return T::tryWriteLock();
    }

    virtual ~AbstractMutexWriteI() {}
};
```

Apart from use with Freeze servants, these templates are also useful if, for example, you want to implement your own evictor.

F.3 Cache

This class allows you to efficiently maintain a cache that is backed by secondary storage, such as a Berkeley DB database, without holding a lock on the entire cache while values are being loaded from the database. If you want to create evictors (see Section 32.9.4) for servants that store their state in a database, the `Cache` class can simplify your evictor implementation considerably.[1]

The `Cache` class has the following interface:

```
template<typename Key, typename Value>
class Cache {
public:
    typedef typename
                std::map</* ... */, /* ... */>::iterator Position;

    bool pin(const Key& k, const Handle<Value>& v);
    Handle<Value> pin(const Key& k);
    void unpin(Position p);

    Handle<Value> putIfAbsent(const Key& k, const Handle<Value>& v
);

    Handle<Value> getIfPinned(const Key&, bool = false) const;

    void clear();
    size_t size() const;


protected:
    virtual Handle<Value> load(const Key& k) = 0;
    virtual void pinned(const Handle<Value>& v, Position p);

    virtual ~Cache();
};
```

Note that `Cache` is an abstract base class—you must derive a concrete implementation from `Cache` and provide an implementation of the `load` and, optionally, of the `pinned` member function.

1. You may also want to examine the implementation of the Freeze `BackgroundSaveEvictor` in the source distribution; it uses `IceUtil::Cache` for its implementation.

Internally, a `Cache` maintains a map of name–value pairs. The key and value type of the map are supplied by the `Key` and `Value` template arguments, respectively. The implementation of `Cache` takes care of maintaining the map; in particular, it ensures that concurrent lookups by callers are possible without blocking even if some of the callers are currently loading values from the backing store. In turn, this is useful for evictor implementations, such as the Freeze `BackgroundSaveEvictor`. The `Cache` class does not limit the number of entries in the cache—it is the job of the evictor implementation to limit the map size by calling `unpin` on elements of the map that it wants to evict.

Your concrete implementation class must implement the `load` function, whose job it is to load the value for the key `k` from the backing store and to return a handle to that value. Note that `load` returns a value of type `IceUtil::Handle` (see page 1981), that is, the value must be heap-allocated and support the usual reference-counting functions for smart pointers. (The easiest way to achieve this is to derive the value from `IceUtil::Shared`—see page 1990.)

If `load` cannot locate a record for the given key because no such record exists, it must return a null handle. If `load` fails for some other reason, it can throw an exception, which is propagated back to the application code.

Your concrete implementation class typically will also override the `pinned` function (unless you want to have a cache that does not limit the number of entries; the provided default implementation of `pinned` is a no-op). The `Cache` implementation calls `pinned` whenever it has added a value to the map as a result of a call to `pin`; the `pinned` function is therefore a callback that allows the derived class to find out when a value has been added to the cache and informs the derived class of the value and its position in the cache.

The `Position` parameter is a `std::iterator` into the cache's internal map that records the position of the corresponding map entry. (Note that the element type of map is opaque, so you should not rely on knowledge of the cache's internal key and value types.) Your implementation of `pinned` must remember the position of the entry because that position is necessary to remove the corresponding entry from the cache again.

The public member functions of `Cache` behave as follows:

- `bool pin(const Key& k, const Handle<Value>& v);`

  To add a key–value pair to the cache, your evictor can call `pin`. The return value is true if the key and value were added; a false return value indicates that

the map already contained an entry with the given key and the original value for that key is unchanged.

`pin` calls `pinned` if it adds an entry.

This version of `pin` does *not* call `load` to retrieve the entry from backing store if it is not yet in the cache. This is useful when you add a newly-created object to the cache.

Once an entry is in the cache, it is guaranteed to remain in the cache at the same position in memory, and without its value being overwritten by another thread, until that entry is unpinned by a call to `unpin`.

- `Handle<Value> pin(const Key& k);`

  This version of `pin` looks for the entry with the given key in the cache. If the entry is already in the cache, `pin` returns the entry's value. If no entry with the given key is in the cache, `pin` calls `load` to retrieve the corresponding entry. If `load` returns an entry, `pin` adds it to the cache and returns the entry's value. If the entry cannot be retrieved from the backing store, `pin` returns null.

  `pin` calls `pinned` if it adds an entry.

  The function is thread-safe, that is, it calls `load` only once all other threads have unpinned the entry.

  Once an entry is in the cache, it is guaranteed to remain in the cache at the same position in memory, and without its value being overwritten by another thread, until that entry is unpinned by a call to `unpin`.

- `Handle<Value> putIfAbsent(const Key& k,`
  `const Handle<Value>& v);`

  This function adds a a key–value pair to the cache and returns a smart pointer to the value. If the map already contains an entry with the given key, that entry's value remains unchanged and `putIfAbsent` returns its value. If no entry with the given key is in the cache, `putIfAbsent` calls `load` to retrieve the corresponding entry. If `load` returns an entry, `putIfAbsent`

adds it to the cache and returns the entry's value. If the entry cannot be retrieved from the backing store, `putIfAbsent` returns null.

`putIfAbsent` calls `pinned` if it adds an entry.

The function is thread-safe, that is, it calls `load` only once all other threads have unpinned the entry.

Once an entry is in the cache, it is guaranteed to remain in the cache at the same position in memory, and without its value being overwritten by another thread, until that entry is unpinned by a call to `unpin`.

- `Handle<Value> getIfPinned(const Key& k, bool wait = false) const;`

  This function returns the value stored for the key `k`.

  - If an entry for the given key is in the map, the function returns the value immediately, regardless of the value of `wait`.
  - If no entry for the given key is in the map and the `wait` parameter is false, the function returns a null handle.
  - If no entry for the given key is in the map and the `wait` parameter is true, the function blocks the calling thread if another thread is currently attempting to load the same entry; once the other thread completes, `getIfPinned` completes and returns the value added by the other thread.

- `void unpin(Position p);`

  This function removes an entry from the map. The iterator `p` determines which entry to remove. (It must be an iterator that previously was passed to `pinned`.) The iterator `p` is invalidated by this operation, so you must not use it again once `unpin` returns. (Note that the `Cache` implementation ensures that updates to the map never invalidate iterators to existing entries in the map; `unpin` invalidates only the iterator for the removed entry.)

- `void clear();`

  This function removes all entries in the map.

- `size_t size() const;`

  This function returns the number of entries in the map.

F.4 `CtrlCHandler`

See Section 31.10 for a description of this class.

F.5 Exception

This class is at the root of the derivation tree for Ice exceptions and encapsulates functionality that is common to all `Ice` and `IceUtil` exceptions:

```
class Exception : public std::exception {
public:
    Exception();
    Exception(const char* file, int line);
    virtual ~Exception() throw();

    virtual std::string ice_name() const;
    virtual void ice_print(std::ostream&) const;
    virtual const char* what() const throw();
    virtual Exception* ice_clone() const;
    virtual void ice_throw() const;
    const char* ice_file() const;
    int ice_line() const;
};
```

The second constructor stores a file name and line number in the exception that are returned by the `ice_file` and `ice_line` member functions, respectively. This allows you to identify the source of an exception by passing the `__FILE__` and `__LINE__` preprocessor macros to the constructor.

The `what` member function is a synonym for `ice_print`. The default implementation of `ice_print` prints the file name, line number, and the name of the exception.

The remaining member functions are described in Section 6.9.

F.6 generateUUID

The signature of `generateUUID` is:

```
std::string generateUUID();
```

The function returns a universally-unique identifier, such as

```
02b066f5-c762-431c-8dd3-9b1941355e41
```

Each invocation returns a new identifier that is differs from all previous ones.[2]

2. Or, rather, differs from all previous ones for the next few decades.

F.7 `Handle` Template

`IceUtil::Handle` implements a smart reference-counted pointer type. Smart pointers are used to guarantee automatic deletion of heap-allocated class instances. (See Section 6.14.6 for a detailed explanation of smart pointers.)

`Handle` is a template class with the following interface:[3]

```
template<typename T>
class Handle : /* ... */ {
public:

    typedef T element_type;

    T* _ptr;

    T* operator->() const;
    T& operator*() const;
    T* get() const;

    operator bool() const;

    void swap(HandleBase& other);

    Handle(T* p = 0);

    template<typename Y>
    Handle(const Handle<Y>& r);

    Handle(const Handle& r);

    ~Handle();

    Handle& operator=(T* p);

    template<typename Y>
    Handle& operator=(const Handle<Y>& r);

    Handle& operator=(const Handle& r);
```

3. Note that the actual implementation is split into a base and a derived class. For simplicity, we show the combined interface here. If you want to see the full implementation detail, it can be found in `IceUtil/Handle.h`.

```
    template<class Y>
    static Handle dynamicCast(const HandleBase<Y>& r);

    template<class Y>
    static Handle dynamicCast(Y* p);
};

template<typename T, typename U>
bool operator==(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator!=(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator<(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator<=(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator>(const Handle<T>& lhs, const Handle<U>& rhs);

template<typename T, typename U>
bool operator>=(const Handle<T>& lhs, const Handle<U>& rhs);
```

The template argument must be a class that derives from `Shared` or `SimpleShared` (or that implements reference counting with the same interface as these classes); see page 1990 for a description of these classes.

This is quite a large interface, but all it really does is to faithfully mimic the behavior of ordinary C++ class instance pointers. Rather than discussing each member function in detail, we provide a simple overview here that outlines the most important points. Please see Section 6.14.6 for more examples of how to use smart pointers.

- `element_type`

  This type definition follows the STL convention of defining the element type with the fixed name `element_type` so you can use it for template programming or the definition of generic containers.

- `_ptr`

  This data member stores the pointer to the underlying heap-allocated class instance.

- Constructors, copy constructor, and assignment operators

  These member functions allow you to construct, copy, and assign smart pointers as if they were ordinary pointers. In particular, the constructor and assignment operator are overloaded to work with raw C++ class instance pointers, which results in the "adoption" of the raw pointer by the smart pointer. For example, the following code works correctly and does not cause a memory leak:

  ```
  typedef Handle<MyClass> MyClassPtr;

  void foo(const MyClassPtr&);

  // ...

  foo(new MyClass); // OK, no leak here.
  ```

- `operator->`, `operator*`, and `get`

  The arrow and indirection operators allow you to apply the usual pointer syntax to smart pointers to use the target of a smart pointer. The `get` member function returns the class instance pointer to the underlying reference-counted class instance; the return value is the value of `_ptr`.

- `dynamicCast`

  This member function works exactly like a C++ `dynamic_cast`:[4] it tests whether the argument supports the specified type and, if so, returns a non-null pointer; if the target does not support the specified type, it returns null. For example:

  ```
  MyClassPtr p = ...;
  MyOtherClassPtr o = ...;

  o = MyOtherClassPtr::dynamicCast(p);
  if (o)
  {
      // o points at an instance of type MyOtherClass.
  }
  else
  {
  ```

4. The reason for not using an actual `dynamic_cast` and using a `dynamicCast` function instead is that `dynamic_cast` only operates on pointer types, but `IceUtil::Handle` is a class.

```
    // p points at something that is
    // not compatible with MyOtherClass.
}
```

Note that this example also illustrates the use of `operator bool`: when used in a boolean context, a smart pointer returns true if it is non-null and false otherwise.

- Comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`

  The comparison operators compare the value of the underlying class instance pointer, that is, they compare the value returned by `get`. In other words, `==` returns true if two smart pointers point at the same underlying class instance, and the ordering operators compare the memory addresses of the underlying class instances.

F.8 `Handle` Template Adaptors

`IceUtil` provides adaptors that support use of smart pointers with STL algorithms. Each template function returns a corresponding function object that is for use by an STL algorithm. The adaptors are defined in the header `IceUtil/Functional.h`.

Here is a list of the adaptors:

```
memFun
memFun1
voidMemFun
voidMemFun1

secondMemFun
secondMemFun1
secondVoidMemFun
secondVoidMemFun1

constMemFun
constMemFun1
constVoidMemFun
constVoidMemFun1

secondConstMemFun
secondConstMemFun1
secondConstVoidMemFun
secondConstVoidMemFun1
```

As you can see, the adaptors are in two groups. The first group operates on non-const smart pointers, whereas the second group operates on `const` smart pointers (for example, on smart pointers declared as `const MyClassPtr`).

Each group is further divided into two sub-groups. The adaptors in the first group operate on the target of a smart pointer, whereas the `second<name>` adapters operate on the second element of a pair, where that element is a smart pointer.

Each of the four sub-groups contains four adaptors:

- `memFun`

  This adaptor is used for member functions that return a value and do not accept an argument. For example:

  ```
  class MyClass : public IceUtil::Shared {
  public:
      MyClass(int i) : _i(i) {}
      int getVal() { return _i; }
  private:
      int _i;
  };

  typedef IceUtil::Handle<MyClass> MyClassPtr;

  // ...

  vector<MyClassPtr> mcp;
  mcp.push_back(new MyClass(42));
  mcp.push_back(new MyClass(99));

  transform(mcp.begin(), mcp.end(),
            ostream_iterator<int>(cout, " "),
            IceUtil::memFun(&MyClass::getVal));
  cout << endl;
  ```

  This code invokes the member function `getVal` on each instance that is pointed at by smart pointers in the vector `mcp` and prints the return value of `getVal` on `cout`, separated by spaces. The output from this code is:

  ```
  42 99
  ```

- `memFun1`

  This adaptor is used for member functions that return a value and accept a single argument. For example:

  ```
  class MyClass : public IceUtil::Shared {
  ```

```
public:
    MyClass(int i) : _i(i) {}
    int plus(int v) { return _i + v; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(2));
mcp.push_back(new MyClass(4));
mcp.push_back(new MyClass(6));

int A[3] = { 5, 7, 9 };
transform(mcp.begin(), mcp.end(), A,
          ostream_iterator<int>(cout, " "),
          IceUtil::memFun1(&MyClass::plus));
cout << endl;
```

This code invokes the member function `plus` on each instance that is pointed at by smart pointers in the vector `mcp` and prints the return value of a call to `plus` on `cout`, separated by spaces. The calls to `plus` are successively passed the values stored in the array `A`. The output from this code is:

```
7 11 15
```

- `voidMemFun`

  This adaptor is used for member functions that do not return a value and do not accept an argument. For example:

```
class MyClass : public IceUtil::Shared {
public:
    MyClass(int i) : _i(i) {}
    void print() { cout << _i << endl; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

vector<MyClassPtr> mcp;
```

```
mcp.push_back(new MyClass(2));
mcp.push_back(new MyClass(4));
mcp.push_back(new MyClass(6));

for_each(mcp.begin(), mcp.end(),
         IceUtil::voidMemFun(&MyClass::print));
```

This code invokes the member function `print` on each instance that is pointed at by smart pointers in the vector `mcp`. The output from this code is:

```
2
4
6
```

- `voidMemFun1`

  This adaptor is used for member functions that do not return a value and accept a single argument. For example:

```
class MyClass : public IceUtil::Shared {
public:
    MyClass(int i) : _i(i) {}
    void printPlus(int v) { cout << _i + v << endl; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

vector<MyClassPtr> mcp;
mcp.push_back(new MyClass(2));
mcp.push_back(new MyClass(4));
mcp.push_back(new MyClass(6));

for_each(
    mcp.begin(), mcp.end(),
    bind2nd(IceUtil::voidMemFun1(&MyClass::printPlus), 3));
```

  This code invokes the member function `printPlus` on each instance that is pointed at by smart pointers in the vector `mcp`. The output from this code is:

```
5
7
9
```

As mentioned on page 1985, the `second<name>` versions of the adaptors operate on the second element of a `std::pair<T1, T2>`, where `T2` must be a

smart pointer. Most commonly, these adaptors are used to apply an algorithm to each lookup value of a map or multi-map. Here is an example:

```
class MyClass : public IceUtil::Shared {
public:
    MyClass(int i) : _i(i) {}
    int plus(int v) { return _i + v; }
private:
    int _i;
};

typedef IceUtil::Handle<MyClass> MyClassPtr;

// ...

map<string, MyClassPtr> m;
m["two"] = new MyClass(2);
m["four"] = new MyClass(4);
m["six"] = new MyClass(6);

int A[3] = { 5, 7, 9 };
transform(
    m.begin(), m.end(), A,
    ostream_iterator<int>(cout, " "),
    IceUtil::secondMemFun1<int, string, MyClass>(&MyClass::plus));
```

This code invokes the `plus` member function on the class instance denoted by the `second` smart pointer member of each pair in the dictionary `m`. The output from this code is:

```
9 13 11
```

Note that `secondMemFun1` is a template that requires three arguments: the return type of the member function to be invoked, the key type of the dictionary, and the type of the class that is pointed at by the smart pointer.

In general, the `second<name>` adaptors require the following template arguments:

```
secondMemFun<R, K, T>
secondMemFun1<R, K, T>
secondVoidMemFun<K, T>
secondVoidMemFun<K, T>
```

where `R` is the return type of the member function, `K` is the type of the first member of the pair, and `T` is the class that contains the member function.

F.9 ScopedArray

`IceUtil::ScopedArray` is a smart pointer class similar to `Ptr` smart pointers. However, instead of managing the memory for class instances, `ScopedArray` manages memory for an array. This class is provided mainly for use with the stream API (see Section 35.2.1). However, you can use it with arrays for other purposes.

Here is the definition of the template in full:

```
template<typename T>
class ScopedArray : private IceUtil::noncopyable
{
public:
    explicit ScopedArray(T* ptr = 0)
        : _ptr(ptr) { }

    ScopedArray(const ScopedArray& other) {
        _ptr = other._ptr;
        const_cast<ScopedArray&>(other)._ptr = 0;
    }

    ~ScopedArray() {
        if (_ptr != 0)
            delete[] _ptr;
    }

    void reset(T* ptr = 0) {
        assert(ptr == 0 || ptr != _ptr);
        if (_ptr != 0)
            delete[] _ptr;
        _ptr = ptr;
    }

    T& operator[](size_t i) const {
        assert(_ptr != 0);
        assert(i >= 0);
        return _ptr[i];
    }

    T* get() const {
        return _ptr;
    }

    void swap(ScopedArray& a) {
```

```
        T* tmp = a._ptr;
        a._ptr = _ptr;
        _ptr = tmp;
    }

private:
    T* _ptr;
};
```

The class allows you to allocate an array on the heap and assign its pointer to a `ScopedArray` instance. When the instance goes out of scope, it calls `delete[]` on the array, so you do not need to deallocate the array explicitly yourself. This greatly reduces the risk of a memory leak due to an early return or uncaught exception.

F.10 `Shared` and `SimpleShared`

`IceUtil::Shared` and `IceUtil::SimpleShared` are base classes that implement the reference-counting mechanism for smart pointers (see Section 6.14.6). The two classes provide identical interfaces; the difference between `Shared` and `SimpleShared` is that `SimpleShared` is not thread-safe and, therefore, can only be used if the corresponding class instances are accessed only by a single thread. (`SimpleShared` is marginally faster than `Shared` because it avoids the locking overhead that is incurred by `Shared`.)

The interface of `Shared` looks as follows. (Because `SimpleShared` has the same interface, we do not show it separately here.)

```
class Shared {
public:
    Shared();
    Shared(const Shared&);
    virtual ~Shared();

    Shared& operator=(const Shared&);

    virtual void __incRef();
    virtual void __decRef();
    virtual int __getRef() const;
    virtual void __setNoDelete(bool);
};
```

The class maintains a reference that is initialized to zero by the constructor. `__incRef` increments the reference count and `__decRef` decrements it. If, during a call to `__decRef`, after decrementing the reference count, the reference count drops to zero, `__decRef` calls `delete this`, which causes the corresponding class instance to delete itself. The copy constructor increments the reference count of the copied instance, and the assignment operator increments the reference count of the source and decrements the reference count of the target.

The `__getRef` member function returns the value of the reference count and is useful mainly for debugging.

The `__setNoDelete` member function can be used to temporarily disable self-deletion and re-enable it again. This is useful mainly if you initialize a smart pointer with the `this` pointer of a class instance during construction—see page 241 for a detailed explanation.

To create a class that is reference-counted, you simply derive the class from `Shared` and define a smart pointer type for the class, for example:

```
class MyClass : public IceUtil::Shared {
    // ...
};

typedef IceUtil::Handle<MyClass> MyClassPtr;
```

F.11 Threads and Synchronization Primitives

The `IceUtil` namespace contains a platform-independent thread and synchronization API. This API is documented in Chapter 31.

F.12 `Time`

The `Time` class provides basic facilities for getting the current time, constructing time intervals, adding and subtracting times, and comparing times:

```
namespace IceUtil {

    typedef ... Int64;

    class Time {
    public:
        enum Clock { Realtime, Monotonic };
```

```
Time(Clock = Realtime);
static Time now();
static Time seconds(Int64);
static Time milliSeconds(Int64);
static Time microSeconds(Int64);

Int64 toSeconds() const;
Int64 toMilliSeconds() const;
Int64 toMicroSeconds() const;

double toSecondsDouble() const;
double toMilliSecondsDouble() const;
double toMicroSecondsDouble() const;

std::string toDateTime() const;
std::string toDuration() const;

Time operator-() const;

Time operator-(const Time&) const;
Time operator+(const Time&) const;

Time operator*(int) const;
Time operator*(Int64) const;
Time operator*(double) const;

double operator/(const Time&) const;
Time operator/(int) const;
Time operator/(Int64) const;
Time operator/(double) const;

Time& operator-=(const Time&);
Time& operator+=(const Time&);

Time& operator*=(int);
Time& operator*=(Int64);
Time& operator*=(double);

Time& operator/=(int);
Time& operator/=(Int64);
Time& operator/=(double);

bool operator<(const Time&) const;
bool operator<=(const Time&) const;
bool operator>(const Time&) const;
bool operator>=(const Time&) const;
```

```
        bool operator==(const Time&) const;
        bool operator!=(const Time&) const;

#ifndef _WIN32
        operator timeval() const;
#endif
    };

    std::ostream& operator<<(std::ostream&, const Time&);
}
```

The member functions behave as follows:

- `Time`

  Internally, the `Time` class stores ticks in microsecond units. For absolute time, this is the number of microseconds since the Unix epoch (00:00:00 UTC on 1 Jan. 1970). For durations, this is the number of microseconds in the duration. The default constructor initializes the tick count to zero and selects the real-time clock. Constructing `Time` with an argument of `Monotonic` selects the monotonic clock on platforms that support it; the real-time clock is used on other platforms.

- `now`

  This function constructs a `Time` object that is initialized to the current time of day.

- `seconds`
  `milliSeconds`
  `microSeconds`

  These functions construct `Time` objects from the argument in the specified units. For example, the following code fragment creates a time duration of one minute:

  ```
  IceUtil::Time t = IceUtil::Time::seconds(60);
  ```

- `toSeconds`
  `toMilliSeconds`
  `toMicroSeconds`

  The member functions provide explicit conversion of a duration to seconds, milliseconds, and microseconds, respectively. The return value is a 64-bit signed integer (`IceUtil::Int64`).

  ```
  IceUtil::Time t = IceUtil::Time::milliSeconds(2000);
  IceUtil::Int64 secs = t.toSeconds(); // Returns 2
  ```

- `toSecondsDouble`
  `toMilliSecondsDouble`
  `toMicroSecondsDouble`

  The member functions provide explicit conversion of a duration to seconds, milliseconds, and microseconds, respectively. The return value is of type `double`.

- `toDateTime`

  This function returns a human-readable representation of a `Time` value as a date and time.

- `toDuration`

  This function returns a human-readable representation of a `Time` value as a duration.

- `operator-`
  `operator+`
  `operator*`
  `operator/`
  `operator-=`
  `operator+=`
  `operator*=`
  `operator/=`

  These operators allow you to add, subtract, multiply, and divide times. For example:

  ```
  IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
  IceUtil::Time oneMinuteAgo = IceUtil::Time::now() - oneMinute;
  ```

  The multiplication and division operators permit you to multiply and divide a duration. Note that these operators provide overloads for `int`, `long long`, and `double`.

- The comparison operators allow you to compare times and time intervals with each other, for example:

  ```
  IceUtil::Time oneMinute  = IceUtil::Time::seconds(60);
  IceUtil::Time twoMinutes = IceUtil::Time::seconds(120);
  assert(oneMinute < twoMinutes);
  ```

- `operator timeval`

  This operator converts a `Time` object to a `struct timeval`, defined as follows:

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

The conversion is useful for API calls that require a `struct timeval` argument, such as `select`. To convert a duration into a `timeval` structure, simply assign a `Time` object to a struct timeval:

```
IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
struct timeval tv;
tv = t;
```

Note that this member function is not available under Windows.

- `std::ostream& operator<<(std::ostream&, Time&);`

  This operator prints the number of whole seconds since the epoch.

F.13 `Timer` and `TimerTask`

The `Timer` class allows you to schedule some code for once-only or repeated execution after some time interval elapses. The code to be executed resides in a class you derive from `TimerTask`:

```
class Timer;
typedef IceUtil::Handle<Timer> TimerPtr;

class TimerTask : virtual public IceUtil::Shared {
public:
    virtual ~TimerTask() { }
    virtual void runTimerTask() = 0;
};

typedef IceUtil::Handle<TimerTask> TimerTaskPtr;
```

Your derived class must override the `runTimerTask` member function; the code in this method is executed by the timer. If the code you want to run requires access to some program state, you can pass that state into the constructor of your class or, alternatively, set that state via member functions of your class before scheduling it with a timer.

The `Timer` class invokes the `runTimerTask` member function to run your code. The class has the following definition:

```
class Timer : /* ... */ {
public:
    Timer();
    Timer(int priority);

    void schedule(const TimerTaskPtr& task,
                  const IceUtil::Time& interval);

    void scheduleRepeated(const TimerTaskPtr& task,
                          const IceUtil::Time& interval);

    bool cancel(const TimerTaskPtr& task);

    void destroy();
};

typedef IceUtil::Handle<Timer> TimerPtr;
```

The constructor is overloaded to allow you specify a thread priority (see Section 31.8.1). The priority controls the priority of the thread that executes your task.

The `schedule` member function schedules an instance of your timer task for once-only execution after the specified time interval has elapsed. Your code is executed by a separate thread that is created by the `Timer` class. The function throws an `IllegalArgumentException` if you invoke it on a destroyed timer.

The `scheduleRepeated` member function runs your task repeatedly, at the specified time interval. Your code is executed by a separate thread that is created by the `Timer` class; the same thread is used every time your code runs. The function throws an `IllegalArgumentException` if you invoke it on a destroyed timer.

If your code throws an exception, the `Timer` class ignores the exception, that is, for a task that is scheduled to run repeatedly, an exception in the current execution does not cancel the next execution.

If your code takes longer to execute than the time interval you have specified for repeated execution, the second execution is delayed accordingly. For example, if you ask for repeated execution once every five seconds, and your code takes ten seconds to complete, then the second execution of your task starts five seconds after the previous execution finishes, that is, the interval specifies the wait time between successive executions.

A `TimerTask` instance that has already been scheduled with a `Timer` instance cannot be scheduled again with the same `Timer` instance until the task has completed or been canceled.

For a single `Timer` instance, the execution of all registered tasks is serialized. The wait interval applies on a per-task basis so, if you schedule task A at an interval of five seconds, and task B at an interval of ten seconds, successive runs of task A start no sooner than five seconds after the previous task A has finished, and successive runs of task B start no sooner than ten seconds after the previous task B has finished. If, at the time a task is scheduled to run, another task is still running, the new task's execution is delayed until the previous task has finished.

If you want scheduled tasks to run concurrently, you can create several `Timer` instances; tasks then execute in as many threads concurrently as there are `Timer` instances.

The `cancel` member function removes a task from a timer's schedule. In other words, it stops a task that is scheduled for repeated execution from being executed again. (For once-only tasks, `cancel` does nothing.) If you cancel a task while it is executing, `cancel` returns immediately and the currently running task is allowed to complete normally; that is, `cancel` does not wait for any currently running task to complete.

The return value is true if `cancel` removed the task from the schedule. This is the case if you invoke `cancel` on a task that is scheduled for repeated execution and this was the first time you cancelled that task; subsequent calls to `cancel` return false. Calling `cancel` on a task scheduled for once-only execution always returns false, as does calling `cancel` on a destroyed timer.

The `destroy` member function removes all tasks from the timer's schedule. If you call `destroy` from any thread other than the timer's own execution thread, it joins with the currently executing task (if any), so the function does not return until the current task has completed. If you call `destroy` from the timer's own execution thread, it instead detaches the timer's execution thread. Calling `destroy` a second time on the same `Timer` instance has no effect. Similarly, calling `cancel` on a destroyed timer has no effect.

Note that you *must* call `destroy` on a `Timer` instance before allowing it to go out of scope; failing to do so causes undefined behavior.

Calls to `schedule` or `scheduleRepeated` on a destroyed timer do nothing.

F.14 Unicode and UTF-8 Conversion Functions

The `IceUtil` namespace contains two helper functions that allow you to convert between wide strings containing Unicode characters (either 16- or 32-bit, depending on your native `wchar_t` size) and narrow strings in UTF-8 encoding:

```
enum ConversionFlags { strictConversion, lenientConversion };

std::string wstringToString(const std::wstring&,
                            ConversionFlags = lenientConversion);
std::wstring stringToWstring(const std::string&,
                             ConversionFlags = lenientConversion);
```

These functions always convert to and from UTF-8 encoding, that is, they ignore any locale setting that might specify a different encoding.

Byte sequence that are illegal, such as `0xF4908080`, result in a `UTFConversionException`. For other errors, the `ConversionFlags` parameter determines how rigorously the functions check for errors. When set to `lenientConversion` (the default), the functions tolerate isolated surrogates and irregular sequences, and substitute the UTF-32 replacement character `0x0000FFFD` for character values above `0x10FFFF`. When set to `strictConversion`, the functions do not tolerate such errors and throw a `UTFConversionException` instead:

```
enum ConversionError { partialCharacter, badEncoding };

class UTFConversionException : public Exception {
public:
    UTFConversionException(const char* file, int line,
                           ConversionError r);

    ConversionError conversionError() const;
    // ...
};
```

The `conversionError` member function returns the reason for the failure:

- `partialCharacter`

  The UTF-8 source string contains a trailing incomplete UTF-8 byte sequence.

- `badEncoding`

  The UTF-8 source string contains a byte sequence that is not a valid UTF-8 encoded character, or the Unicode source string contains a bit pattern that does not represent a valid Unicode character.

F.15 Version Information

The header file `IceUtil/Config.h` defines two macros that expand to the version of the Ice run time:

```
#define ICE_STRING_VERSION "3.3.0" // "<major>.<minor>.<patch>"
#define ICE_INT_VERSION 30300      // AABBCC, with AA=major,
                                   // BB=minor, CC=patch
```

`ICE_STRING_VERSION` is a string literal in the form *`<major>.<minor>.<patch>`*, for example, `3.3.0`. For beta releases, the version is *`<major>.<minor>`*b, for example, `3.3b`.

`INT_VERSION` is an integer literal in the form *`AABBCC`*, where *`AA`* is the major version number, *`BB`* is the minor version number, and *`CC`* is patch level, for example, `30300` for version 3.3.0. For beta releases, the patch level is set to 51 so, for example, for version 3.3b, the value is `30351`.

Appendix G
The Java Utility Library

G.1 Introduction

Ice for Java includes a number of utility APIs in the `IceUtil` package and the `Ice.Util` class. This appendix summarizes the contents of these APIs for reference.

G.2 The `IceUtil` Package

`Cache` and `Store`

The `Cache` class allows you to efficiently maintain a cache that is backed by secondary storage, such as a Berkeley DB database, without holding a lock on the entire cache while values are being loaded from the database. If you want to create evictors (see Section 32.9.4) for servants that store their state in a database, the `Cache` class can simplify your evictor implementation considerably.[1]

The `Cache` class has the following interface:

1. You may also want to examine the implementation of the Freeze `BackgroundSaveEvictor` in the source distribution; it uses `IceUtil.Cache` for its implementation.

```
package IceUtil;

public class Cache {
    public Cache(Store store);

    public Object pin(Object key);
    public Object pin(Object key, Object o);
    public Object unpin(Object key);

    public Object putIfAbsent(Object key, Object newObj);
    public Object getIfPinned(Object key);

    public void clear();
    public int size();
}
```

Internally, a `Cache` maintains a map of name–value pairs. The implementation of `Cache` takes care of maintaining the map; in particular, it ensures that concurrent lookups by callers are possible without blocking even if some of the callers are currently loading values from the backing store. In turn, this is useful for evictor implementations, such as the Freeze `BackgroundSaveEvictor`. The `Cache` class does not limit the number of entries in the cache—it is the job of the evictor implementation to limit the map size by calling `unpin` on elements of the map that it wants to evict.

The `Cache` class works in conjunction with a `Store` interface for which you must provide an implementation. The `Store` interface is trivial:

```
package IceUtil;

public interface Store {
    Object load(Object key);
}
```

You must implement the `load` method in a class that you derive from `Store`. The `Cache` implementation calls `load` when it needs to retrieve the value for the passed key from the backing store. If `load` cannot locate a record for the given key because no such record exists, it must return null. If `load` fails for some other reason, it can throw an exception derived from `java.lang.RuntimeException`, which is propagated back to the application code.

The public member functions of `Cache` behave as follows:

- `Cache(Store s);`

  The constructor initializes the cache with your implementation of the `Store` interface.

- `Object pin(Object key, Object val);`

  To add a key–value pair to the cache, your evictor can call `pin`. The return value is null if the key and value were added; otherwise, if the map already contains an entry with the given key, the entry is unchanged and `pin` returns the original value for that key.

  This version of `pin` does *not* call `load` to retrieve the entry from backing store if it is not yet in the cache. This is useful when you add a newly-created object to the cache.

- `Object pin(Object key);`

  This version of `pin` returns the value stored in the cache for the given key if the cache already contains an entry for that key. If no entry with the given key is in the cache, `pin` calls `load` to retrieve the corresponding value (if any) from the backing store. `pin` returns the value returned by `load`, that is, the value if `load` could retrieve it, null if `load` could not retrieve it, or any exception thrown by `load`.

- `Object unpin(Object key);`

  `unpin` removes the entry for the given key from the cache. If the cache contained an entry for the key, the return value is the value for that key; otherwise, the return value is null.

- `Object putIfAbsent(Object key, Object val);`

  This function adds a key–value pair to the cache. If the cache already contains an entry for the given key, `putIfAbsent` returns the original value for that key. If no entry with the given key is in the cache, `putIfAbsent` calls `load` to retrieve the corresponding entry (if any) from the backing store and returns the value returned by `load`.

  If the cache does not contain an entry for the given key and `load` does not retrieve a value for the key, the method adds the new entry and returns null.

- `Object getIfPinned(Object key);`

  This function returns the value stored for the given key. If an entry for the given key is in the map, the function returns the corresponding value; otherwise, the function returns null. `getIfPinned` does *not* call `load`.

- `void clear();`

  This function removes all entries in the map.

- `int size();`

  This function returns the number of entries in the map.

G.3 The Ice.Util Class

Communicator Initialization Methods

`Ice.Util` provides a number of overloaded `initialize` methods that create a communicator. See Section 30.7 for details on these methods.

Identity Conversion

`Ice.Util` contains two methods to convert object identities of type `Ice.Identity` to and from strings. These methods are described in Section 32.5.2.

Property Creation Methods

`Ice.Util` provides a number of overloaded `createProperties` methods that create property sets. See Section 30.9.2 for details on these methods.

Proxy Comparison Methods

Two methods, `proxyIdentityCompare` and `proxyIdentityAndFacetCompare`, allow you to compare object identities that are stored in proxies (either ignoring the facet or taking the facet into account). See Section 10.11.5 for more details.

Stream Creation

Two methods, `createInputStream` and `createOutputStream` create streams for use with dynamic invocation. See Section 35.2.2 for more detail.

UUID Generation

`Ice.Util` contains a method `generateUUID` with the following signature:

```
static synchronized String generateUUID();
```

The function returns a unique identifier, such as

```
c0:a8:4:3:6e868177:119eabd625b:-8000
```

Version Information

The `stringVersion` and `intVersion` methods return the version of the Ice run time:

```
public static String stringVersion();
public static int intVersion();
```

The `stringVersion` method returns the Ice version in the form *<major>.<minor>.<patch>*, for example, `3.3.0`. For beta releases, the version is *<major>.<minor>*b, for example, `3.3b`.

The `intVersion` method contains the Ice version in the form *AABBCC*, where *AA* is the major version number, *BB* is the minor version number, and *CC* is patch level, for example, `30300` for version 3.3.0. For beta releases, the patch level is set to 51 so, for example, for version 3.3b, the value is `30351`.

Appendix H
Windows Services

H.1 Introduction

A Windows service is a program that runs in the background and typically does not require user intervention. Similar to a daemon on Unix platforms, a Windows service is usually launched automatically when the operating system starts and runs until the system shuts down.

Ice includes a class named `Service` that simplifies the task of writing an Ice-based Windows service in C++ (see Section 8.3.2). Writing the service is only the first step, however, as it is also critically important that the service be installed and configured correctly for successful operation. Service installation and configuration is outside the scope of the `Service` class because these tasks are generally not performed by the service itself but rather as part of a larger administrative effort to deploy an application. Furthermore, there are security implications to consider when a service is able to install itself: such a service typically needs administrative rights to perform the installation, but does not necessarily need those rights while it is running. A better strategy is to grant administrative rights to a separate installer program and not to the service itself.

H.2 Installing a Windows Service

The installation of a Windows service varies in complexity with the needs of the application, but usually involves the following activities:

- Selecting the user account in which the service will run.
- Registering the service and establishing its activation mode and dependencies.
- Creating one or more file system directories to contain executables, libraries, and supporting files or databases.
- Configuring those directories with appropriate permissions so that they are accessible to the user account selected for the service.
- Creating keys in the Windows registry.
- Configuring the Windows Event Log so that the service can report status and error messages.

There are many ways to perform these tasks. For example, an administrator can execute them manually, as we discuss beginning on page 2014. Another option is to write a script or program tailored to the needs of your application. Finally, you can build an installer using a developer tool such as InstallShield.

Selecting a User Account

Before installing a service, you should give careful consideration to the user account that will run the service. Unless your service has special requirements, we recommend that you use the built-in account that Windows provides specifically for this purpose. On Windows XP and Windows Server 2003, the fully-qualified name for this account is `NT Authority\LocalService`; in an English locale, its name is displayed as `Local Service`. On Windows Vista, the account name is simply `Local Service`.

H.3 The Ice Service Installer

Ice provides the command-line tool **`iceserviceinstall`** to assist you in installing and uninstalling the following Windows services:

- IceGrid registry
- IceGrid node
- Glacier2 router

Ice includes other programs that can also be run as Windows services, such as the IceBox and IcePatch2 servers. Typically it is not necessary to install these programs as Windows services because they can be launched by an IceGrid node service. However, if you wish to run an IceBox or IcePatch2 server as a Windows service without the use of IceGrid, you must install the service manually (see page 2014).

In this section, we describe how to use the Ice service installer and discuss its actions and prerequisites.

Usage

`iceserviceinstall` supports the following options and arguments:

```
iceserviceinstall [options] service config-file [property ...]

Options:
-h, --help           Show this message.
-n, --nopause        Do not call pause after displaying a message.
-v, --version        Display the Ice version.
-u, --uninstall      Uninstall the Windows service.
```

The ***`service`*** and ***`config-file`*** arguments are required during installation and uninstallation.

The ***`service`*** argument selects the type of service you are installing; use one of the following values:

- `icegridregistry`
- `icegridnode`
- `glacier2router`

Note that the Ice service installer currently does not support the installation of an IceGrid node with a collocated registry, therefore you must install the registry and node separately.

The ***`config-file`*** argument specifies the name of an Ice configuration file. See page 2010 for additional information.

When installing a service, properties can be defined on the command line using the ***`--name=value`*** syntax, or they can be defined in the configuration file. The supported properties are described on page 2011.

Security Considerations

None of the Ice services require privileges beyond a normal user account. In the case of the IceGrid node service in particular, we do not recommend running it in a user account with elevated privileges because the service is responsible for launching server executables, and those servers would inherit the node's access rights. See page 2008 for more information selecting a user account.

Configuration File

The Ice service installer requires that you specify the path name of the Ice configuration file for the service being installed or uninstalled. The tool needs this path name for several reasons:

- During installation, it verifies that the configuration file has sufficient access rights.
- It configures a newly-installed service to load the configuration file using its absolute path name, therefore you must decide in advance where the file will be located.
- It reads the configuration file and examines certain service-specific properties. For example, prior to installing an IceGrid registry service, the tool verifies that the directory specified by the property `IceGrid.Registry.Data` has sufficient access rights.
- The tool supports its own configuration parameters that may also be defined as properties in this file (see page 2011).

You may still modify a service's configuration file after installation, but you should uninstall and reinstall the service if you change any of the properties that influence the service installer's actions. The table below describes the service properties that affect the installer:

Property	Service	Description
`IceGrid.InstanceName`	IceGrid Registry	Value appears in the service name; also included in the default display name if one is not defined.
`IceGrid.Node.Data`	IceGrid Node	Directory is created if necessary; access rights are verified.

Property	Service	Description
`IceGrid.Node.Name`	IceGrid Node	Value appears in the service name; also included in the default display name if one is not defined.
`IceGrid.Registry.Data`	IceGrid Registry	Directory is created if necessary; access rights are verified.
`Ice.Default.Locator`	IceGrid Node, Glacier2 Router	The IceGrid instance name is derived from the identity in this proxy.
`Ice.EventLog.Source`	All	Specifies the name of an event log source for the service.

The steps performed by the tool during an installation are described in detail on page 2012.

Sample Configuration Files

Ice includes sample configuration files for the IceGrid and Glacier2 services in the `config` subdirectory of your Ice installation. We recommend that you review the comments and settings in these files to familiarize yourself with a typical configuration of each service.

You can modify a configuration file to suit your needs or copy one to use as a starting point for your own configuration.

Installer Properties

The Ice service installer uses a set of optional properties that customize the installation process. These properties can be defined in the service's configuration file as discussed in the previous section, or they can be defined on the command line using the familiar ***--name=value*** syntax:

```
iceserviceinstall --DependOnRegistry=1 ...
```

The installer's properties are listed below:

- `AutoStart=num`

  If *num* is a value greater than zero, the service is configured to start automatically at system start up. Set this property to zero to configure the service to start on demand instead. If not specified, the default value is `1`.

- `DependOnRegistry=num`

  If *num* is a value greater than zero, the service is configured to depend on the IceGrid registry, meaning Windows will start the registry prior to starting this service. Enabling this feature also requires that the property `Ice.Default.Locator` be defined in *`config-file`*. If not specified, the default value is zero.

- `Description=value`

  A brief description of the service. If not specified, a general description is used.

- `DisplayName=name`

  The friendly name that identifies the service to the user. If not specified, **`iceserviceinstall`** composes a default display name.

- `EventLog=name`

  The name of the event log used by the service. If not specified, the default value is `Application`.

- `ImagePath=path`

  The path name of the service executable. If not specified, **`iceserviceinstall`** assumes the service executable resides in the same directory as itself and fails if the executable is not found.

- `ObjectName=name`

  Specifies the account used to run the service. If not specified, the default value is `NT Authority\LocalService`.

- `Password=value`

  The password required by the account specified in `ObjectName`.

Installation Process

The Ice service installer performs a number of steps to install a service. As discussed on page 2010, you must specify the path name of the service's configuration file because the service installer uses certain properties during the installation process. The actions taken by the service installer are described below:

- Obtain the service's *instance name* from the configuration file. The instance name is specified by the property `IceGrid.InstanceName` or `Glacier2.InstanceName`. If an instance name is not specified, the default value is `IceGrid` or `Glacier2`, respectively. If the service being installed depends on the IceGrid registry, the IceGrid instance name is derived from the value of the `Ice.Default.Locator` property.
- For an IceGrid node, obtain the node's name from the property `IceGrid.Node.Name`. This property must be defined when installing a node.
- Compose the service name from the service type, instance name, and node name (for an IceGrid node). For example, the default service name for an IceGrid registry is `icegridregistry.IceGrid`. Note that the service name is not the same as the display name.
- Resolve the user account specified by `ObjectName`.
- For an IceGrid registry, create the data directory specified by the property `IceGrid.Registry.Data` and ensure that the user account specified by `ObjectName` has read/write access to the directory.
- For an IceGrid node, create the data directory specified by the property `IceGrid.Node.Data` and ensure that the user account specified by `ObjectName` has read/write access to the directory.
- For an IceGrid node, ensure that the user account specified by `ObjectName` has read access to the following registry key:

  `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib`

  This key allows the node to access CPU utilization statistics (see page 2024).
- Ensure that the user account specified by `ObjectName` has read/write access to the configuration file.
- Create a new Windows event log by adding the registry key specified by `EventLog`.
- Add an event log source under `EventLog` for the source name specified by `Ice.EventLog.Source` (see Appendix D). If this property is not defined, the service name is used as the source name.
- Install the service, including command line arguments that specify the service name (**`--service name`**) and the absolute path name of the configuration file (**`--Ice.Config=`*`config-file`***).

The Ice service installer currently does *not* perform these tasks:

- modify access rights for the service's executable or its DLL dependencies
- verify that the user account specified by `ObjectName` has the right to "Log on as a service"

Uninstallation

When uninstalling an existing service, the Ice service installer first ensures that the service is stopped, then proceeds to remove the service. The service's event log source is removed and, if the service is not using the `Application` log, the event log registry key is also removed.

H.4 Manual Installation

This section describes how to manually install and configure an Ice service using the IcePatch2 service as a case study. For the purposes of this discussion, we assume that Ice is installed in the directory `C:\Ice`. We also assume that you have administrative access to your system, which is required by many of the installation steps discussed in this section.

Selecting a User Account

The IcePatch2 service can run in a regular user account, therefore we will follow the recommendation on page 2008 and use the `Local Service` account.

Preparing a Directory

The service needs a directory in which to store the files that it distributes to clients. A common mistake is assuming that a service will be able to access a file or directory that you created using your current account, which is likely to cause the service to fail in a way that is difficult to diagnose. To prevent such failures, we will ensure that the directory has the necessary permissions for the service to access it while running in the `Local Service` account.

Selecting a Directory

The directory tree for our IcePatch2 service is shown below:

```
C:\Documents and Settings\
  LocalService\
    Local Settings\
      Application Data\
        ZeroC\
          icepatch2\
            data\
```

Note that this tree applies to Windows XP and Windows Server 2003, and is locale dependent. On Windows Vista, we would use the following tree instead:

```
C:\Windows\
  ServiceProfiles\
    LocalService\
      AppData\
        Local\
          ZeroC\
            icepatch2\
              data\
```

For this example, we will use the Windows XP directory tree.

Creating the Directory

Since `Local Service` is a built-in account, its user directory should already exist and have the proper access rights.[1] If the directory does not exist, we can create it in a command window with the following steps:

```
cd \Documents and Settings
mkdir LocalService
```

At this point we could create the rest of the directory hierarchy. However, a newly-created directory inherits the privileges of its enclosing directory, and we have not yet modified the privileges of the `LocalService` directory to grant access to the `Local Service` account. At present, the privileges of the `LocalService` directory are inherited from `Documents and Settings` and require modification. In general, it is better to establish the necessary access rights on the parent directory prior to creating any subdirectories, so we will modify the `LocalService` directory first.

1. If you open `C:\Documents and Settings` in Windows Explorer, the `LocalService` directory may not be visible until you modify your folder options to show protected files and folders.

On all Windows systems, we can use the command-line utility `cacls`. The following command does what we need:

```
cacls LocalService /G "Local Service":F Administrators:F
```

By omitting the `/E` option to `cacls`, we have replaced all of the prior access rights on the directory with the rights given in this command. As a result, the `Local Service` account and anyone in the Administrators group are granted full access to the directory, while all others are forbidden. (We grant full access to the Administrators group because presumably someone other than the `Local Service` account will need to manage the subdirectory, create the configuration file, an so on). You can verify the directory's current privilege settings by running `cacls` without options:

```
cacls LocalService
```

Now we can create the remaining subdirectories, and they will automatically inherit the access rights established for the `LocalService` directory:

```
cd LocalService
mkdir "Local Settings\Application Data\ZeroC\icepatch2\data"
```

If you want to further restrict access to files or subdirectories, you can modify them as necessary using the `cacls` utility. Note however that certain actions may cause a file to revert back to the access rights of its enclosing directory. For example, modifying a file using a text editor is often the equivalent of erasing the file and recreating it, which discards any access rights you may have previously set for the file.

On some versions of Windows XP, and on Windows Server 2003 and Windows Vista, you can manage privilege settings interactively using Windows Explorer. For example, right click on the `LocalService` directory, select Properties, and select the Security tab. Next select Advanced and Edit, uncheck “Include inheritable permissions from this object’s parent,” and select Copy. Remove all permission entries, then add entries for Local Service and the Administrators group and grant Full Control to each.

Populating the Directory

Now you can copy the files that will be distributed to clients into the `data` subdirectory. The new files should inherit the access rights of their enclosing directory. For the sake of discussion, let us copy some Slice files from the Ice distribution into the data directory:

```
cd "Local Settings\Application Data\ZeroC\icepatch2\data"
copy \Ice\slice\Ice\*.ice
```

Next we need to run `icepatch2calc` to prepare the directory for use by the IcePatch2 service:

```
icepatch2calc .
```

Configuration File

IcePatch2 requires a minimal set of configuration properties. We could specify them on the service's command line, but if we later want to modify those properties we would have to reinstall the service. Defining the properties in a file simplifies the task of modifying the service's configuration.

Our IcePatch2 configuration is quite simple:

```
IcePatch2.Directory=C:\Documents and Settings\LocalService\
Local Settings\Application Data\ZeroC\icepatch2\data
IcePatch2.Endpoints=tcp -p 10000
```

The `IcePatch2.Directory` property specifies the location of the server's data directory, which we created in the previous section.

We will save our configuration properties into the following file:

```
C:\Ice\config\icepatch2.cfg
```

We must also ensure that the service has permission to access its configuration file. The Ice run time never modifies a configuration file, therefore read access is sufficient. The configuration file likely already has the necessary access rights, which we can verify using the `cacls` utility that we described in the previous section:

```
cacls C:\Ice\config\icepatch2.cfg
```

Creating the Service

We will use Microsoft's Service Control (`sc`) utility[2] in a command window to create the service. Our first `sc` command does the majority of the work (the command is formatted for readability but must be typed on a single line):

2. See `http://support.microsoft.com/kb/251192` for more information about the `SC` utility.

```
sc create icepatch2
  binPath= "C:\Ice\bin\icepatch2server.exe
    --Ice.Config=C:\Ice\config\icepatch2.cfg --service icepatch2"
  DisplayName= "IcePatch2 Server" start= auto
  obj= "NT Authority\LocalService" password= ""
```

There are several important aspects of this command:

- The service name is `icepatch2`. You can use whatever name you like, as long as it does not conflict with an existing service. Note however that this name is used in other contexts, such as in the `--service` option discussed below, therefore you must use it consistently.
- Following the service are several options. Note that all of the option names end with an equals sign and are separated from their arguments with at least one space.
- The `binPath=` option is required. We supply the full path name of the IcePatch2 server executable, as well as command-line arguments that define the location of the configuration file and the name of the service, all enclosed in quotes.
- The `DisplayName=` option sets a friendly name for the service.
- The `start=` option configures the start up behavior for the service. We used the argument `auto` to indicate the service should be started automatically when Windows boots.
- The `obj=` option selects the user account in which this service runs. As explained on page 2008, the `Local Service` account is appropriate for most services.
- The `password=` option supplies the password associated with the user account indicated by `obj=`. The `Local Service` account has an empty password.

The `sc` utility should report success if it was able to create the service as specified. You can verify that the new service was created with this command:

```
sc qc icepatch2
```

Alternatively, you can start the Services administrative control panel and inspect the properties of the IcePatch2 service.

If you start the control panel, you will notice that the entry for IcePatch2 does not have a description. To add a description for the service, use the following command:

```
sc description icepatch2 "IcePatch2 file server"
```

After refreshing the list of services, you should see the new description.

Creating the Event Log

By default, programs such as the IcePatch2 service that utilize the `Service` class (see Section 8.3.2) log messages to the `Application` event log. Below we describe how to prepare the Windows registry for the service's default behavior, and we also show how to use a custom event log instead. We make use of Microsoft's Registry (`reg`) utility to modify the registry, although you could also use the interactive `regedit` tool. As always, caution is recommended whenever you modify the registry.

Using the Application Log

We must configure an event log source for events to display properly. The first step is to create a registry key with the name of the source. Since the `Service` class uses the service name as the source name by default, we add the key `icepatch2` as shown below:

```
reg add HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
Application\icepatch2
```

Inside this key we must add a value that specifies the location of the Ice run time DLL:

```
reg add HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
Application\icepatch2 /v EventMessageFile /t REG_EXPAND_SZ
/d C:\Ice\bin\ice34.dll
```

We will also add a value indicating the types of events that the source supports:

```
reg add HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
Application\icepatch2 /v TypesSupported /t REG_DWORD /d 7
```

The value `7` corresponds to the combination of the following event types:

- `EVENTLOG_ERROR_TYPE`
- `EVENTLOG_WARNING_TYPE`
- `EVENTLOG_INFORMATION_TYPE`

You can verify that the registry values have been defined correctly using the following command:

```
reg query HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
Application\icepatch2
```

Our configuration of the event log is now complete.

Changing the Source Name

Using the configuration described in the previous section, events logged by the IcePatch2 service are recorded in the event log using the source name `icepatch2`. If you prefer to use a source name that differs from the service name, you can replace `icepatch2` in the registry commands with the name of your choosing, but you must also add a matching definition for the property `Ice.EventLog.Source` to the service's configuration file.

For example, to use the source name `Ice File Patching Service`, you would add the registry key as shown below:

```
reg add "HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
Application\Ice File Patching Service"
```

The commands to add the `EventMessageFile` and `TypesSupported` values must be modified in a similar fashion. Finally, add the following configuration property to `icepatch2.cfg`:

```
Ice.EventLog.Source=Ice File Patching Service
```

Using a Custom Log

You may decide that you want your services to record messages into an application-specific log instead of the `Application` log that is shared by other unrelated services. As an example, let us create a log named `MyApp`:

```
reg add "HKLM\SYSTEM\CurrentControlSet\Services\EventLog\MyApp"
```

Next we add a subkey for the IcePatch2 service. As described in the previous section, we will use a friendlier source name:

```
reg add "HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
MyApp\Ice File Patching Service"
```

Now we can define values for `EventMessageFile` and `TypesSupported`:

```
reg add "HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
MyApp\Ice File Patching Service" /v EventMessageFile
/t REG_EXPAND_SZ /d C:\Ice\bin\ice34.dll

reg add "HKLM\SYSTEM\CurrentControlSet\Services\EventLog\
MyApp\Ice File Patching Service" /v TypesSupported /t REG_DWORD /d 7
```

Finally, we define `Ice.EventLog.Source` in the IcePatch2 service's configuration file:

```
Ice.EventLog.Source=Ice File Patching Service
```

Note that you must restart the Event Viewer control panel after adding the `MyApp` registry key in order to see the new log.

Registry Caching

The first time a service logs an event, Windows' Event Log service caches the registry entries associated with the service's source. If you wish to modify a service's event log configuration, such as changing the service to use a custom log instead of the `Application` log, you should perform the following steps:

1. Stop the service.
2. Remove the unwanted event log registry key.
3. Add the new event log registry key(s).
4. Restart the system (or at least the Event Log service).
5. Start the service and verify that the log entries appear in the intended log.

After following these steps, open a log entry and ensure that it displays properly. If it does not, for example if the event properties indicate that the description of an event cannot be found, the problem is likely due to a misconfigured event source. Verify that the value of `EventMessageFile` refers to the correct location of the Ice run time DLL, and that the service is defining `Ice.EventLog.Source` in its configuration file (if necessary).

Starting the Service

We are at last ready to start the service. In a command window, you can use the `sc` utility:

```
sc start icepatch2
```

The program usually responds with status information indicating that the start request is pending. You can query the service's status to verify that it started successfully:

```
sc query icepatch2
```

The service should now be in the running state. If it is not in this state, open the Event Viewer control panel and inspect the relevant log for more information that should help you to locate the problem. Even if the service started successfully, you may still want to use the Event Viewer to confirm that the service is using the log you expected.

Test the Service

Ice includes a graphical IcePatch2 client in the `demo/IcePatch2/MFC` directory of the Ice distribution. Once you have built the client, you can use it to test that the service is working properly.

H.5 Troubleshooting

Missing Libraries

One failure that commonly occurs when starting a Windows service is caused by missing DLLs, which usually results in an error window stating a particular DLL cannot be found. Fixing this problem can often be a trial-and-error process because the DLL mentioned in the error may depend on other DLLs that are also missing. It is important to understand that a Windows service is launched by the operating system and can be configured to execute as a different user, which means the service's environment (most importantly its `PATH`) may not match yours and therefore extra steps are necessary to ensure that the service can locate its required DLLs[3].

The simplest approach is to copy all of the necessary DLLs to the directory containing the service executable. If this solution is undesirable, another option is to modify the system `PATH` to include the directory or directories containing the required DLLs. (Note that modifying the system `PATH` requires restarting the system.) Finally, you can copy the necessary DLLs to `\WINDOWS\system32`, although we do not recommend this approach[4].

Assuming that DLL issues are resolved, a Windows service can fail to start for a number of other reasons, including

- invalid command-line arguments or configuration properties

3. The command-line utility `dumpbin` can be used to discover the dependencies of an executable or DLL.
4. Copying DLLs to `\WINDOWS\system32` often results in subtle problems later when trying to develop using newer versions of the DLLs. Inevitably you will forget about the DLLs in `\WINDOWS\system32` and struggle to determine why your application is misbehaving or failing to start.

- inability to access necessary resources such as file systems and databases, because either the resources do not exist or the service does not have sufficient access rights to them
- networking issues, such as attempting to open a port that is already in use, or DNS lookup failures

Failures encountered by the Ice run time prior to initialization of the communicator are reported to the Windows event log if no other logger implementation is defined, so that should be the first place you look. Typically you will find an entry in the `System` event log resembling the following message:

```
The IcePatch2 service terminated with service-specific error 1.
```

Error code `1` corresponds to `EXIT_FAILURE`, the value used by the `Service` class to indicate a failure during startup. Additional diagnostic messages may be available in the `Application` event log. See page 289 for more information on configuring a logger for a Windows service.

As we mentioned earlier, insufficient access rights can also prevent a Windows service from starting successfully. By default, a Windows service is configured to run under a local system account, in which case the service may not be able to access resources owned by other users. It may be necessary for you to configure a service to run under a different account, which you can do using the Services control panel. See page 2008 for more information on selecting a user account for your service, and page 2014 for instructions on configuring the access rights of files and directories.

Windows Firewall

Your choice of user account determines whether you receive any notification when the Windows Firewall blocks the ports that are used by your service. For example, if you use `Local Service` as recommended on page 2008, you will not see any Windows Security Alert dialog (see this Microsoft article for details).

If you are not prompted to unblock your service, you will need to manually add an exception in Windows Firewall. For example, follow the steps below to unblock the ports of a Glacier2 router service:

1. Open the Windows Firewall Settings panel and navigate to the Exceptions panel.
2. Select “Add program...”
3. Select “Browse,” navigate to the Glacier2 router executable, and click “OK.”

Note that adding an exception for a program unblocks all ports on which the program listens. Review the endpoint configurations of your services carefully to ensure that no unnecessary ports are opened.

For services listening on one or a few fixed ports, you could also create port exceptions in your Windows Firewall. Please refer to the Windows Firewall documentation for details.

IceGrid Node Performance Monitoring

The IceGrid node uses Windows' `Perflib` facility to obtain statistics about the CPU utilization of its host (see Section 38.10). On Vista-derived operating systems, the IceGrid node may log the following warning message:

```
warning: Unable to lookup the performance counter name
```

This message is an indication that the node does not have sufficient privileges to access a key in the Windows registry:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib
```

As part of its installation procedure, the **`iceserviceinstall`** utility described in Section H.2 modifies the permissions of this registry key to grant read access to the node's designated user account. If you are trying to change the node's user account, we recommend using the **`iceserviceinstall`** utility to uninstall and reinstall the node. If you wish to modify the permissions of this registry key manually, follow these steps:

1. Start **`regedit`** and navigate to the `Perflib` key.
2. Right click on `Perflib` and select `Permissions`.
3. If the desired user account is not already present, click `Add` to add the user account. Enter `LOCAL SERVICE` if you wish to run the node in the Local Service account, otherwise enter the name of the user account. Press `OK`.
4. Check the `Read` box in the `Allow` column to grant read access to the registry key and press `OK` to apply the changes.

Another way to grant the node's user account with the necessary access rights is to add it to the `Performance Monitor Users` group.

Appendix I
The .NET Utility Library

I.1 Introduction

Ice for .Net includes a number of utility APIs in the `Ice.Util` class. This appendix summarizes the contents of these APIs for reference.

I.2 Communicator Initialization Methods

`Ice.Util` provides a number of overloaded `initialize` methods that create a communicator. See Section 30.7 for details on these methods.

I.3 Identity Conversion

`Ice.Util` contains two methods to convert object identities of type `Ice.Identity` to and from strings. These methods are described in Section 32.5.2.

I.4 Property Creation Methods

`Ice.Util` provides a number of overloaded `createProperties` methods that create property sets. See Section 30.9.2 for details on these methods.

I.5 Proxy Comparison Methods

Two methods, `proxyIdentityCompare` and `proxyIdentityAndFacetCompare`, allow you to compare object identities that are stored in proxies (either ignoring the facet or taking the facet into account). See Section 14.10.4 for more details.

I.6 Stream Creation

Two methods, `createInputStream` and `createOutputStream` create streams for use with dynamic invocation. See Section 35.2.3 for more detail.

I.7 UUID Generation

`Ice.Util` contains a method `generateUUID` with the following signature:

```
static string generateUUID();
```

The function returns a universally-unique identifier, such as

```
02b066f5-c762-431c-8dd3-9b1941355e41
```

Each invocation returns a new identifier that differs from all previous ones.[1]

I.8 Version Information

The `stringVersion` and `intVersion` methods return the version of the Ice run time:

1. Or, rather, differs from all previous ones for the next few decades.

```
public static string stringVersion();
public static int intVersion();
```

The `stringVersion` method returns the Ice version in the form *<major>.<minor>.<patch>*, for example, `3.3.0`. For beta releases, the version is *<major>.<minor>*b, for example, `3.3`b.

The `intVersion` method contains the Ice version in the form *AABBCC*, where *AA* is the major version number, *BB* is the minor version number, and *CC* is patch level, for example, `30300` for version 3.3.0. For beta releases, the patch level is set to 51 so, for example, for version 3.3b, the value is `30351`.

Appendix J
Binary Distributions

J.1 Introduction

ZeroC supplies binary distributions for the combinations of platforms and compilers that Ice supports. The packaging of each distribution varies with the platform:

- a Microsoft Installer (MSI) for Windows
- a collection of RPMs for Red Hat Enterprise Linux and SuSE Linux Enterprise Server
- a compressed TAR file (*tarball*) for other Unix platforms

In general, the binary distributions are intended for the developer, and not for the end users of the developer's application. In other words, ZeroC expects you, as an Ice developer, to bundle the necessary Ice run time components into your own installation package. This chapter discusses ZeroC's binary distributions and provides guidelines for distributing your applications.

J.2 Developer Kits

The Windows installer and the Unix tarballs are developer kits that provide everything necessary for you to develop applications with Ice, including header files,

pre-compiled shared libraries, Slice compilers, and executables for services such as IceGrid. The Windows installer also includes the source code for sample programs[1] as well as debug versions of the Ice run time libraries.

The RPM distributions are different than the monolithic distributions of other platforms. Packaged using standard RPM conventions, these distributions include separate run time and developer kit RPMs for each language mapping. For example, the `ice-java` RPM contains the Ice for Java run time (`Ice.jar`), while the `ice-java-devel` RPM contains development tools such as the Slice-to-Java compiler. Additional RPMs are provided for third-party dependencies that are not included in the Red Hat distribution, or whose versions are out of date, such as Berkeley DB.

Regardless of the platform, we do not recommend using Ice developer kits as a way of installing the Ice run time components required by your application because the majority of what is installed by a developer kit is irrelevant to end user applications. The RPM distribution, however, is packaged in such a way that you may find it convenient to simply redistribute select Ice run time RPMs along with your application.

J.3 Guidelines

This section provides some guidance for developers that are planning to distribute an Ice-based application. We can start by listing items that typically should *not* be included in your binary distribution:

- Slice compilers
- Slice files (unless you are using a scripting language, as discussed below)
- Executables and libraries for Ice services and tools that your application does not use
- For C++ programs on Windows:
  - DLLs built in debug mode (such as `ice34d.dll`)
  - program database (`PDB`) files
  - header files
  - import library (`LIB`) files

1. Sample programs for Unix platforms are provided as a separate source tarball.

Each of the language mappings is discussed in its own subsection below. In the following discussion, we use the term *library* to refer to a shared library or DLL as appropriate for the platform.

C++

The `Ice` library contains the implementation of the core Ice run time. Supplemental libraries provide the stubs and skeletons for the Ice services, as well as utility functions used by Ice, its services, and Ice applications:

- `Glacier2`
- `IceBox`
- `IceGrid`
- `IcePatch2`
- `IceSSL`
- `IceStorm`

The `IceUtil` library is a dependency of the `Ice` library and therefore must be distributed with any Ice application. The `IceXML` library is required by certain Ice services.

Your distribution needs to include only those libraries that your application uses. If your application implements an IceBox service, you must also distribute the IceBox server executable (`icebox`).

Discovering Dependencies

On Windows, you can use the **`dumpbin`** utility in a command window to display the dependencies of a DLL or executable. For example, here is the output for the **`glacier2router`** executable:

```
> dumpbin /dependents glacier2router.exe
ice34.dll
iceutil34.dll
LIBEAY32.dll
glacier234.dll
icessl34.dll
MSVCP90.dll
MSVCR90.dll
KERNEL32.dll
```

We can deduce from the names of the Microsoft Visual C++ run time DLLs that this Ice installation was compiled with Visual Studio 2008. Note that each of these

DLLs has its own dependencies, which can be inspected using additional **dumpbin** commands. However, tracking down the dependencies recursively through each DLL can quickly become tedious, therefore you should consider using the Dependency Walker[2] graphical utility instead.

On Unix, the **ldd** utility displays the dependencies of shared libraries and executables.

Qt Libraries

The IceGrid and IceStorm services use a pluggable architecture for their persistent storage needs. These services use Freeze as their default persistent store, but it is also possible to use SQL databases instead. To implement the SQL persistence solution, Ice uses libraries from the Qt framework (see `http://qt.nokia.com`). Ice has no other dependency on the Qt libraries, therefore you only need to include the Qt libraries in your own distribution if you use IceGrid (see Section 38.16) or IceStorm (see Section 44.12.6) with a SQL database. In this case, you would need to bundle the following libraries:

- `IceGridSqlDB` (for IceGrid)
- `IceStormSqlDB` (for IceStorm)
- `QtCore` and `QtSql`

.NET

The `Ice` assembly contains the implementation of the core Ice run time. Supplemental assemblies provide the stubs and skeletons for the Ice services:

- `Glacier2`
- `IceBox`
- `IceGrid`
- `IcePatch2`
- `IceSSL`
- `IceStorm`

Your distribution needs to include only those assemblies that your application uses. If your application implements an IceBox service, you must also distribute the IceBox server executable (`iceboxnet.exe`).

2. See `http://www.dependencywalker.com`.

On Mono, the file `Ice.dll.config` provides a mapping for the Bzip2 DLL. If your application does not use Ice's protocol compression feature, you do not need to distribute this file. Otherwise, you should include the file and verify that its mapping is appropriate for your target platform.

Java

The Ice for Java run time (`Ice.jar`) contains the following components:

- implementations of Ice, IceSSL, and the IceBox server
- stubs and skeletons for all of the Ice services

If your application uses Freeze, you must also distribute `Freeze.jar` along with the Berkeley DB run time libraries and JAR file.

For assistance with packaging your Java application, consider using a utility such as ProGuard.[3]

Python and Ruby

The Ice run time for a Python or Ruby application consists of the following components:

- the library for the scripting language extension: `IcePy` or `IceRuby`
- the libraries required by the extension: `Ice`, `IceUtil`, and `Slice`
- the source code generated from the Slice files in the Ice distribution

In addition, your distribution should include the source code generated for your own Slice files, or the Slice files themselves if your application loads them dynamically.

PHP

The Ice run time for a PHP application consists of the following components:

- the library for the scripting language extension: `IcePHP` or `php_ice`
- the libraries required by the extension: `Ice`, `IceUtil`, and `Slice`
- the source code generated from the Slice files in the Ice distribution

3. See http://proguard.sourceforge.net.

In addition, your distribution should include the source code generated for your own Slice files.

Appendix K
Deprecated AMI Mapping

K.1 Chapter Overview

NOTE: As of version 3.4, Ice provides a new API for asynchronous method invocation (AMI). The mapping described in this appendix is the old AMI mapping, which is deprecated and will be removed in a future version of Ice.

This chapter describes the deprecated asynchronous method invocation API. Section K.2 gives a brief overview of the capabilities and demonstrates how to modify Slice definitions to enable asynchronous support in language mappings. Section K.3 provides an introduction to the types and concepts used in the AMI mapping. Finally, Section K.4 presents the API for C++, Java, C#, and Python.

K.2 Introduction

Modern middleware technologies attempt to ease the programmer's transition to distributed application development by making remote invocations as easy to use as traditional method calls: a method is invoked on an object and, when the method completes, the results are returned or an exception is raised. Of course, in a distributed system the object's implementation may reside on another host, and

consequently there are some semantic differences that the programmer must be aware of, such as the overhead of remote invocations and the possibility of network-related errors. Despite those issues, the programmer's experience with object-oriented programming is still relevant, and this *synchronous* programming model, in which the calling thread is blocked until the operation returns, is familiar and easily understood.

Ice is inherently an asynchronous middleware platform that simulates synchronous behavior for the benefit of applications (and their programmers). When an Ice application makes a synchronous twoway invocation on a proxy for a remote object, the operation's in parameters are marshaled into a message that is written to a transport, and the calling thread is blocked in order to simulate a synchronous method call. Meanwhile, the Ice run time operates in the background, processing messages until the desired reply is received and the calling thread can be unblocked to unmarshal the results.

There are many cases, however, in which the blocking nature of synchronous programming is too restrictive. For example, the application may have useful work it can do while it awaits the response to a remote invocation; using a synchronous invocation in this case forces the application to either postpone the work until the response is received, or perform this work in a separate thread. When neither of these alternatives are acceptable, the asynchronous facilities provided with Ice are an effective solution for improving performance and scalability, or simplifying complex application tasks.

Asynchronous Method Invocation

Asynchronous Method Invocation (*AMI*) is the term used to describe the client-side support for the asynchronous programming model. AMI supports both oneway and twoway requests, but unlike their synchronous counterparts, AMI requests never block the calling thread. When a client issues an AMI request, the Ice run time hands the message off to the local transport buffer or, if the buffer is currently full, queues the request for later delivery. The application can then continue its activities and, in the case of a twoway invocation, is notified when the reply eventually arrives. Notification occurs via a callback to an application-supplied programming-language object[1].

1. Polling for a response is not supported by the Ice run time, but it can be implemented easily by the application if desired.

AMI is described in detail in Section K.3.

Controlling Code Generation using Metadata

A programmer indicates a desire to use an asynchronous model (AMI, AMD, or both) by annotating Slice definitions with metadata (Section 4.17). The programmer can specify this metadata at two levels: for an interface or class, or for an individual operation. If specified for an interface or class, then asynchronous support is generated for all of its operations. Alternatively, if asynchronous support is needed only for certain operations, then the generated code can be minimized by specifying the metadata only for those operations that require it.

Synchronous invocation methods are always generated in a proxy; specifying AMI metadata merely adds asynchronous invocation methods. In contrast, specifying AMD metadata causes the synchronous dispatch methods to be *replaced* with their asynchronous counterparts. This semantic difference between AMI and AMD is ultimately practical: it is beneficial to provide a client with synchronous and asynchronous versions of an invocation method, but doing the equivalent in a server would require the programmer to implement both versions of the dispatch method, which has no tangible benefits and several potential pitfalls.

Consider the following Slice definitions:

```
["ami"] interface I {
  bool isValid();
  float computeRate();
};

interface J {
  ["amd"]        void startProcess();
  ["ami", "amd"] int endProcess();
};
```

In this example, all proxy methods of interface `I` are generated with support for synchronous and asynchronous invocations. In interface `J`, the `startProcess` operation uses asynchronous dispatch, and the `endProcess` operation supports asynchronous invocation and dispatch.

Specifying metadata at the operation level, rather than at the interface or class level, not only minimizes the amount of generated code, but more importantly, it minimizes complexity. Although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations for which it provides a particular advantage, while using the simpler synchronous model for the rest.

Transparency

The use of an asynchronous model does not affect what is sent "on the wire." Specifically, the invocation model used by a client is transparent to the server, and the dispatch model used by a server is transparent to the client. Therefore, a server has no way to distinguish a client's synchronous invocation from an asynchronous invocation, and a client has no way to distinguish a server's synchronous reply from an asynchronous reply.

K.3 Using AMI

In this section, we describe the Ice implementation of AMI and how to use it. We begin by discussing a way to (partially) simulate AMI using oneway invocations. This is not a technique that we recommend, but it is an informative exercise that highlights the benefits of AMI and illustrates how it works. Next, we explain the AMI mapping and illustrate its use with examples.

Simulating AMI using Oneways

As we discussed at the beginning of the chapter, synchronous invocations are not appropriate for certain types of applications. For example, an application with a graphical user interface typically must avoid blocking the window system's event dispatch thread because blocking makes the application unresponsive to user commands. In this situation, making a synchronous remote invocation is asking for trouble.

The application could attempt to avoid this situation using oneway invocations (see Section 32.14), which by definition cannot return a value or have any `out` parameters. Since the Ice run time does not expect a reply, the invocation blocks only as long as it takes to establish a connection (if necessary), marshal the request, and copy the message into the local transport buffer. However, these network activities may still block. Furthermore, the use of oneway invocations may require unacceptable changes to the interface definitions. For example, a twoway invocation that returns results or raises user exceptions must be converted into at least two operations: one for the client to invoke with oneway semantics that contains only in parameters, and one (or more) for the server to invoke to notify the client of the results.

To illustrate these changes, suppose that we have the following Slice definition:

```
interface I {
  int op(string s, out long l);
};
```

In its current form, the operation `op` is not suitable for a oneway invocation because it has an `out` parameter and a non-`void` return type. In order to accommodate a oneway invocation of `op`, we can change the Slice definitions as shown below:

```
interface ICallback {
  void opResults(int result, long l);
};

interface I {
  void op(ICallback* cb, string s);
};
```

We made several modifications to the original definition:

- We added interface `ICallback`, containing an operation `opResults` whose arguments represent the results of the original twoway operation. The server invokes this operation to notify the client of the completion of the operation.
- We modified `I::op` to be compliant with oneway semantics: it now has a `void` return type, and takes only in parameters.
- We added a parameter to `I::op` that allows the client to supply a proxy for its callback object.

As you can see, we have made significant changes to our interface definitions to accommodate the implementation requirements of the client. One ramification of these changes is that the client must now also be a server, because it must create an instance of `ICallback` and register it with an object adapter in order to receive notifications of completed operations.

A more severe ramification, however, is the impact these changes have on the type system, and therefore on the server. Whether a client invokes an operation synchronously or asynchronously should be irrelevant to the server; this is an artifact of behavior that should have no impact on the type system. By changing the type system as shown above, we have tightly coupled the server to the client, and eliminated the ability for `op` to be invoked synchronously.

To make matters even worse, consider what would happen if `op` could raise user exceptions. In this case, `ICallback` would have to be expanded with additional operations that allow the server to notify the client of the occurrence of each exception. Since exceptions cannot be used as parameter or member types in

Slice, this quickly becomes a difficult endeavor, and the results are likely to be equally difficult to use.

At this point, you will hopefully agree that this technique is flawed in many ways, so why do we bother describing it in such detail? The reason is that the Ice implementation of AMI uses a strategy similar to the one described above, with several important differences:

1. No changes to the type system are required in order to use AMI. The on-the-wire representation of the data is identical, therefore synchronous and asynchronous clients and servers can coexist in the same system, using the same operations.
2. The AMI solution accommodates exceptions in a reasonable way.
3. Using AMI does not require the client to also be a server.
4. Ice guarantees that AMI requests never block the calling thread.

Overview

AMI operations have the same semantics in all of the language mappings that support asynchronous invocations. This section provides a language-independent introduction to the AMI model.

Proxy Method

Annotating a Slice operation with the AMI metadata tag does not prevent an application from invoking that operation using the traditional synchronous model. Rather, the presence of the metadata extends the proxy with an asynchronous version of the operation, so that invocations can be made using either model.

The asynchronous operation never blocks the calling thread. If the message cannot be accepted into the local transport buffer without blocking, the Ice run time queues the request and immediately returns control to the calling thread.

The parameters of the asynchronous operation are modified similar to the example from page 2038: the first argument is a callback object (described below), followed by any `in` parameters in the order of declaration. The operation's return value and `out` parameters, if any, are passed to the callback object when the response is received.

The asynchronous operation only raises `CommunicatorDestroyedException` directly; all other exceptions are reported to the callback object. See page 2057 for more information on error handling.

Finally, the return value of the asynchronous operation is a boolean that indicates whether the Ice run time was able to send the request synchronously; that is, whether the entire message was immediately accepted by the local transport buffer. An application can use this value to implement flow control (see page 2050).

Callback Object

The asynchronous operation requires the application to supply a callback object as the first argument. This object is an instance of an application-defined class; in strongly-typed languages this class must inherit from a superclass generated by the Slice compiler. In contrast to the example on page 2038, the callback object is a purely local object that is invoked by the Ice run time in the client, and not by the remote server.

The Ice run time always invokes methods of the callback object from a thread in an Ice thread pool, and never from the thread that is invoking the asynchronous operation. Exceptions raised by a callback object are ignored but may cause the Ice run time to log a warning message (see the description of `Ice.Warn.AMICallback` in Appendix D).

The callback class must define two methods:

- `ice_response`

  The Ice run time invokes `ice_response` to supply the results of a successful twoway invocation; this method is not invoked for oneway invocations. The arguments to `ice_response` consist of the return value (if the operation returns a non-`void` type) followed by any `out` parameters in the order of declaration.

- `ice_exception`

  This method is called if an error occurs during the invocation. As explained on page 2057, the only exception that can be raised to the thread invoking the asynchronous operation is `CommunicatorDestroyedException`; all other errors, including user exceptions, are passed to the callback object via its `ice_exception` method. In the case of a oneway invocation, `ice_exception` is only invoked if an error occurs before the request is sent.

For an asynchronous invocation, the Ice run time calls `ice_response` or `ice_exception`, but not both. It is possible for one of these methods to be called before control returns to the thread that is invoking the operation.

A callback object may optionally define a third method:

- `ice_sent`

  The `ice_sent` method is invoked when the entire message has been passed to the local transport buffer. The Ice run time does not invoke `ice_sent` if the asynchronous operation returned true to indicate that the message was sent synchronously. An application must make no assumptions about the order of invocations on a callback object; `ice_sent` can be called before, after, or concurrently with `ice_response` or `ice_exception`. Refer to page 2050 for more information about the purpose of this method.

K.4 Language Mappings

The AMI language mappings are described in separate subsections below.

C++ Mapping

The C++ mapping emits the following code for each AMI operation:

1. An abstract callback class whose name is formed using the pattern `AMI_`*`class_op`*. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Two methods must be defined by the subclass:

   ```
   void ice_response(<params>);
   void ice_exception(const Ice::Exception &);
   ```

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method returns a boolean indicating whether the request was sent synchronously. The first parameter is a smart pointer to an instance of the callback class described above. The remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

```
class AMI_I_foo : public ... {
public:
    virtual void ice_response(Ice::Int, Ice::Long) = 0;
    virtual void ice_exception(const Ice::Exception&) = 0;
};
typedef IceUtil::Handle<AMI_I_foo> AMI_I_fooPtr;
```

The proxy method for asynchronous invocation of operation `foo` is generated as follows:

```
bool foo_async(const AMI_I_fooPtr&, Ice::Short);
```

The overview on page 2040 describes proxy methods and callback objects in greater detail.

Java Mapping

The Java mapping emits the following code for each AMI operation:

1. An abstract callback class whose name is formed using the pattern `AMI_`*`class_op`*. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Three methods must be defined by the subclass:

   ```
   public void ice_response(<params>);
   public void ice_exception(Ice.LocalException ex);
   public void ice_exception(Ice.UserException ex);
   ```

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method returns a boolean indicating whether the request was sent synchronously. The first parameter is a reference to an instance of the callback class described above. The remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

```
public abstract class AMI_I_foo extends ... {
    public abstract void ice_response(int __ret, long l);
    public abstract void ice_exception(Ice.LocalException ex);
    public abstract void ice_exception(Ice.UserException ex);
}
```

The proxy methods for asynchronous invocation of operation `foo` are generated as follows:

```
public boolean foo_async(AMI_I_foo __cb, short s);
public boolean foo_async(AMI_I_foo __cb, short s,
                         java.util.Map<String, String> __ctx);
```

As usual, the version of the operation without a context parameter forwards an empty context to the version with a context parameter.

The overview on page 2040 describes proxy methods and callback objects in greater detail.

C# Mapping

The C# mapping emits the following code for each AMI operation:

1. An abstract callback class whose name is formed using the pattern `AMI_`*`class_op`*. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Two methods must be defined by the subclass:

   ```
   public abstract void ice_response(<params>);
   public abstract void ice_exception(Ice.Exception ex);
   ```

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method returns a boolean indicating whether the request was sent synchronously. The first parameter is a reference to an instance of the callback class described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

```
public abstract class AMI_I_foo : ...
{
    public abstract void ice_response(int __ret, long l);
    public abstract void ice_exception(Ice.Exception ex);
}
```

The proxy method for asynchronous invocation of operation `foo` is generated as follows:

```
bool foo_async(AMI_I_foo __cb, short s);
bool foo_async(AMI_I_foo __cb, short s,
               Dictionary<string, string> __ctx);
```

As usual, the version of the operation without a context parameter forwards an empty context to the version with a context parameter.

The overview on page 2040 describes proxy methods and callback objects in greater detail.

Python Mapping

For each AMI operation, the Python mapping emits an additional proxy method having the mapped name of the operation with the suffix `_async`. This method returns a boolean indicating whether the request was sent synchronously. The first parameter is a reference to a callback object; the remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

Unlike the mappings for strongly-typed languages, the Python mapping does not generate a callback class for asynchronous operations. In fact, the callback object's type is irrelevant; the Ice run time simply requires that it define the `ice_response` and `ice_exception` methods:

```
def ice_response(self, <params>)
def ice_exception(self, ex)
```

For example, suppose we have defined the following operation:

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

The method signatures required for the callback object of operation `foo` are shown below:

```
class ...
    #
    # Operation signatures:
    #
    # def ice_response(self, _result, l)
    # def ice_exception(self, ex)
```

The proxy method for asynchronous invocation of operation `foo` is generated as follows:

```
def foo_async(self, __cb, s)
```

The overview on page 2040 describes proxy methods and callback objects in greater detail.

Example

To demonstrate the use of AMI in Ice, let us define the Slice interface for a simple computational engine:

```
module Demo {
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {};

    interface Model {
        ["ami"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    };
};
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way. In the sections below, we present C++, Java, and C# clients that invoke `interpolate` using AMI.

C++ Client

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

```
class AMI_Model_interpolateI : public Demo::AMI_Model_interpolate
{
public:
    virtual void ice_response(const Demo::Grid& result)
```

```
    {
        cout << "received the grid" << endl;
        // ... postprocessing ...
    }

    virtual void ice_exception(const Ice::Exception& ex)
    {
        try {
            ex.ice_throw();
        } catch (const Demo::RangeError& e) {
            cerr << "interpolate failed: range error" << endl;
        } catch (const Ice::LocalException& e) {
            cerr << "interpolate failed: " << e << endl;
        }
    }
};
```

The implementation of `ice_response` reports a successful result, and `ice_exception` displays a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

```
Demo::ModelPrx model = ...;
AMI_Model_interpolatePtr cb = new AMI_Model_interpolateI;
Demo::Grid grid;
initializeGrid(grid);
model->interpolate_async(cb, grid, 0.5);
```

After obtaining a proxy for a `Model` object, the client instantiates a callback object, initializes a grid and invokes the asynchronous version of `interpolate`. When the Ice run time receives the response to this request, it invokes the callback object supplied by the client.

Java Client

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

```
class AMI_Model_interpolateI extends Demo.AMI_Model_interpolate {
    public void ice_response(float[][] result)
    {
        System.out.println("received the grid");
        // ... postprocessing ...
    }

    public void ice_exception(Ice.UserException ex)
    {
```

```
        assert(ex instanceof Demo.RangeError);
        System.err.println("interpolate failed: range error");
    }

    public void ice_exception(Ice.LocalException ex)
    {
        System.err.println("interpolate failed: " + ex);
    }
}
```

The implementation of `ice_response` reports a successful result, and the `ice_exception` methods display a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

```
Demo.ModelPrx model = ...;
AMI_Model_interpolate cb = new AMI_Model_interpolateI();
float[][] grid = ...;
initializeGrid(grid);
model.interpolate_async(cb, grid, 0.5);
```

After obtaining a proxy for a `Model` object, the client instantiates a callback object, initializes a grid and invokes the asynchronous version of `interpolate`. When the Ice run time receives the response to this request, it invokes the callback object supplied by the client.

C# Client

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

```
using System;

class AMI_Model_interpolateI : Demo.AMI_Model_interpolate {
    public override void ice_response(float[][] result)
    {
        Console.WriteLine("received the grid");
        // ... postprocessing ...
    }

    public override void ice_exception(Ice.Exception ex)
    {
        Console.Error.WriteLine("interpolate failed: " + ex);
    }
}
```

The implementation of `ice_response` reports a successful result, and the `ice_exception` method displays a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

```
Demo.ModelPrx model = ...;
AMI_Model_interpolate cb = new AMI_Model_interpolateI();
float[][] grid = ...;
initializeGrid(grid);
model.interpolate_async(cb, grid, 0.5);
```

Python Client

We must first define our callback implementation class:

```
class AMI_Model_interpolateI(object):
    def ice_response(self, result):
        print "received the grid"
        # ... postprocessing ...

  def ice_exception(self, ex):
      try:
          raise ex
      except Demo.RangeError, e:
          print "interpolate failed: range error"
      except Ice.LocalException, e:
          print "interpolate failed: " + str(e)
```

The implementation of `ice_response` reports a successful result, and the `ice_exception` method displays a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

```
model = ...
cb = AMI_Model_interpolateI()
grid = ...
initializeGrid(grid)
model.interpolate_async(cb, grid, 0.5)
```

Concurrency Issues

Support for asynchronous invocations in Ice is enabled by the client thread pool (see Section 32.10), whose threads are primarily responsible for processing reply messages. It is important to understand the concurrency issues associated with asynchronous invocations:

- A callback object must not be used for multiple simultaneous invocations. An application that needs to aggregate information from multiple replies can create a separate object to which the callback objects delegate.
- Calls to the callback object are always made by threads from an Ice thread pool, therefore synchronization may be necessary if the application might interact with the callback object at the same time as the reply arrives. Furthermore, since the Ice run time never invokes callback methods from the client's calling thread, the client can safely make AMI invocations while holding a lock without risk of a deadlock.
- The number of threads in the client thread pool determines the maximum number of simultaneous callbacks possible for asynchronous invocations. The default size of the client thread pool is one, meaning invocations on callback objects are serialized. If the size of the thread pool is increased, the application may require synchronization, and replies can be dispatched out of order. The client thread pool can also be configured to serialize messages received over a connection so that AMI replies from a connection are dispatched in the order they are received (see Section 32.10.4).
- AMI invocations do not use collocation optimization (see page 2057). As a result, AMI invocations are always sent "over the wire" and thus are dispatched by the server thread pool.

Flow Control

The Ice run time queues asynchronous requests when necessary to avoid blocking the calling thread, but places no upper limit on the number of queued requests or the amount of memory they can consume. To prevent unbounded memory utilization, Ice provides the infrastructure necessary for an application to implement its own flow-control logic.

The components were introduced on page 2040:

- The return value of the asynchronous proxy method
- The `ice_sent` method in the AMI callback object

The return value of the proxy method determines whether the request was queued. If the proxy method returns true, no flow control is necessary because the request was accepted by the local transport buffer and therefore the Ice run time did not need to queue it. In this situation, the Ice run time does not invoke the `ice_sent` method on the callback object; the return value of the proxy method is sufficient notification that the request was sent.

If the proxy method returns false, the Ice run time has queued the request. Now the application must decide how to proceed with subsequent invocations:

- The application can be structured so that at most one request is queued. For example, the next invocation can be initiated when the `ice_sent` method is called for the previous invocation.
- A more sophisticated solution is to establish a maximum allowable number of queued requests and maintain a counter (with appropriate synchronization) to regulate the flow of invocations.

Naturally, the requirements of the application must dictate an implementation strategy.

Implementing `ice_sent` in C++

To indicate its interest in receiving `ice_sent` invocations, an AMI callback object must also derive from the C++ class `Ice::AMISentCallback`:

```
namespace Ice {
    class AMISentCallback {
    public:
        virtual ~AMISentCallback();
        virtual void ice_sent() = 0;
    };
}
```

We can modify the example on page 2046 to include an `ice_sent` callback as shown below:

```
class AMI_Model_interpolateI :
    public Demo::AMI_Model_interpolate,
    public Ice::AMISentCallback
{
public:
    // ...

    virtual void ice_sent()
    {
        cout << "request sent successfully" << endl;
    }
};
```

Implementing `ice_sent` in Java

To indicate its interest in receiving `ice_sent` invocations, an AMI callback object must also implement the Java interface `Ice.AMISentCallback`:

```
package Ice;

public interface AMISentCallback {
    void ice_sent();
}
```

We can modify the example on page 2047 to include an `ice_sent` callback as shown below:

```
class AMI_Model_interpolateI
    extends Demo.AMI_Model_interpolate
    implements Ice.AMISentCallback {
    // ...

    public void ice_sent()
    {
        System.out.println("request sent successfully");
    }
}
```

Implementing `ice_sent` in C#

To indicate its interest in receiving `ice_sent` invocations, an AMI callback object must also implement the C# interface `Ice.AMISentCallback`:

```
namespace Ice {
    public interface AMISentCallback
    {
        void ice_sent();
    }
}
```

We can modify the example on page 2048 to include an `ice_sent` callback as shown below:

```
class AMI_Model_interpolateI :
    Demo.AMI_Model_interpolate,
    Ice.AMISentCallback {
    // ...

    public void ice_sent()
    {
        Console.Out.WriteLine("request sent successfully");
    }
}
```

```
class AMI_Model_interpolateI :
    Demo.AMI_Model_interpolate,
    Ice.AMISentCallback {
    // ...

    public void ice_sent()
    {
        Console.Out.WriteLine("request sent successfully");
    }
}
```

Implementing `ice_sent` in Python

To indicate its interest in receiving `ice_sent` invocations, an AMI callback object need only define the `ice_sent` method.

We can modify the example on page 2049 to include an `ice_sent` callback as shown below:

```
class AMI_Model_interpolateI(object):
    # ...

  def ice_sent(self):
      print "request sent successfully"
```

Flushing Batch Requests

Applications that send batched requests (see Section 32.16) can either flush a batch explicitly or allow the Ice run time to flush automatically. The proxy method `ice_flushBatchRequests` performs an immediate flush using the synchronous invocation model and may block the calling thread until the entire message can be sent. Ice also provides an asynchronous version of this method for applications that wish to flush batch requests without the risk of blocking.

The proxy method `ice_flushBatchRequests_async` initiates an asynchronous flush. Its only argument is a callback object; this object must define an `ice_exception` method for receiving a notification if an error occurs before the message is sent.

If the application is interested in flow control (see page 2050), the return value of `ice_flushBatchRequests_async` is a boolean indicating whether the message was sent synchronously. Furthermore, the callback object can define an `ice_sent` method that is invoked when an asynchronous flush completes.

C++ Mapping

The base proxy class `ObjectPrx` defines the asynchronous flush operation as shown below:

```
namespace Ice {
    class ObjectPrx : ... {
    public:
        // ...
        bool ice_flushBatchRequests_async(
            const Ice::AMI_Object_ice_flushBatchRequestsPtr& cb)
    };
}
```

The argument is a smart pointer for an object that implements the following class:

```
namespace Ice {
    class AMI_Object_ice_flushBatchRequests : ... {
    public:
        virtual void ice_exception(const Ice::Exception& ex) = 0;
    };
}
```

As an example, the class below demonstrates how to define a callback class that also receives a notification when the asynchronous flush completes:

```
class MyFlushCallbackI :
    public Ice::AMI_Object_ice_flushBatchRequests,
    public Ice::AMISentCallback
{
public:
    virtual void ice_exception(const Ice::Exception& ex);
    virtual void ice_sent();
};
```

Java Mapping

The base proxy class `ObjectPrx` defines the asynchronous flush operation as shown below:

```
package Ice;

public class ObjectPrx ... {
    // ...
    boolean ice_flushBatchRequests_async(
        AMI_Object_ice_flushBatchRequests cb);
}
```

The argument is a reference for an object that implements the following class:

```
package Ice;

public abstract class AMI_Object_ice_flushBatchRequests ...
{
    public abstract void ice_exception(LocalException ex);
}
```

As an example, the class below demonstrates how to define a callback class that also receives a notification when the asynchronous flush completes:

```
class MyFlushCallbackI
    extends Ice.AMI_Object_ice_flushBatchRequests
    implements Ice.AMISentCallback
{
    public void ice_exception(Ice.LocalException ex) { ... }
    public void ice_sent() { ... }
}
```

C# Mapping

The base proxy class `ObjectPrx` defines the asynchronous flush operation as shown below:

```
namespace Ice {
    public class ObjectPrx : ... {
        // ...
        bool ice_flushBatchRequests_async(
            AMI_Object_ice_flushBatchRequests cb);
    }
}
```

The argument is a reference for an object that implements the following class:

```
namespace Ice {
    public abstract class AMI_Object_ice_flushBatchRequests ... {
        public abstract void ice_exception(Ice.Exception ex);
    }
}
```

As an example, the class below demonstrates how to define a callback class that also receives a notification when the asynchronous flush completes:

```
class MyFlushCallbackI : Ice.AMI_Object_ice_flushBatchRequests,
                         Ice.AMISentCallback
{
    public override void
```

```
    ice_exception(Ice.LocalException ex) { ... }

    public void ice_sent() { ... }
}
```

Python Mapping

The base proxy class defines the asynchronous flush operation as shown below:

```
def ice_flushBatchRequests_async(self, cb)
```

The `cb` argument represents a callback object that must implement an `ice_exception` method. As an example, the class below demonstrates how to define a callback class that also receives a notification when the asynchronous flush completes:

```
class MyFlushCallbackI(object):
    def ice_exception(self, ex):
        # handle an exception

    def ice_sent(self):
        # flush has completed
```

Timeouts

Timeouts for asynchronous invocations behave like those for synchronous invocations: an `Ice::TimeoutException` is raised if the response is not received within the given time period. In the case of an asynchronous invocation, however, the exception is reported to the `ice_exception` method of the invocation's callback object. For example, we can handle this exception in C++ as shown below:

```
class AMI_Model_interpolateI : public Demo::AMI_Model_interpolate
{
public:
    // ...

    virtual void ice_exception(const Ice::Exception& ex)
    {
        try {
            ex.ice_throw();
        } catch (const Demo::RangeError& e) {
            cerr << "interpolate failed: range error" << endl;
        } catch (const Ice::TimeoutException&) {
            cerr << "interpolate failed: timeout" << endl;
        } catch (const Ice::LocalException& e) {
```

```
            cerr << "interpolate failed: " << e << endl;
        }
    }
};
```

Error Handling

It is important to remember that *all* errors encountered by an AMI invocation (except `CommunicatorDestroyedException`) are reported back via the `ice_exception` callback, even if the error condition is encountered "on the way out", when the operation is invoked. The reason for this is consistency: if an invocation, such as `foo_async` could throw exceptions, you would have to handle exceptions in two places in your code: at the point of call for exceptions that are encountered "on the way out", and in `ice_exception` for error conditions that are detected after the call is initiated.

Where this matters is if you want to send off a number of AMI calls, each of which depends on the preceding call to have succeeded. For example:

```
p1->foo_async(cb1);
p2->bar_async(cb2);
```

If `bar` depends for its correct working on the successful completion of `foo`, this code will not work because the `bar` invocation will be sent regardless of whether `foo` failed or not.

In such cases, where you need to be sure that one call is dispatched only if a preceding call succeeds, you must instead invoke `bar` from within `foo`'s `ice_response` implementation, instead of from the main-line code.

Limitations

AMI invocations cannot be sent using collocated optimization. If you attempt to invoke an AMI operation using a proxy that is configured to use collocation optimization, the Ice run time will raise `CollocationOptimizationException` if the servant happens to be collocated; the request is sent normally if the servant is not collocated. Section 32.21 provides more information about this optimization and describes how to disable it when necessary.

K.5 Summary

Synchronous remote invocations are a natural extension of local method calls that leverage the programmer's experience with object-oriented programming and ease the learning curve for programmers who are new to distributed application development. However, the blocking nature of synchronous invocations makes some application tasks more difficult, or even impossible, therefore Ice provides a straightforward interface to its asynchronous facilities.

Using asynchronous method invocation, a calling thread is able to invoke an operation and regain control immediately, without blocking while the operation is in progress. When the results are received, the Ice run time notifies the application via a callback.

Bibliography

References

1. Booch, G., et al. 1998. *Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.
2. Gamma, E., et al. 1994. *Design Patterns.* Reading, MA: Addison-Wesley.
3. Grimes, R. 1998. *Professional DCOM Programming*. Chicago, IL: Wrox Press.
4. Henning, M., and S. Vinoski. 1999. *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley.
5. Housley, R., and T. Polk. 2001. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*. Hoboken, NJ: Wiley.
6. Institute of Electrical and Electronics Engineers. 1985. *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*. Piscataway, NJ: Institute of Electrical and Electronic Engineers.
7. Jain, P., et al. 1997. "Dynamically Configuring Communication Services with the Service Configurator Pattern." *C++ Report 9* (6). http://www.cs.wustl.edu/~schmidt/PDF/Svc-Conf.pdf.
8. Kleiman, S., et al. 1995. *Programming With Threads*. Englewood, NJ: Prentice Hall.

9. Lakos, J. 1996. *Large-Scale C++ Software Design*. Reading, MA: Addison-Wesley.

10. McKusick, M. K., et al. 1996. *The Design and Implementation of the 4.4BSD Operating System.* Reading, MA: Addison-Wesley.

11. Microsoft. 2002. *.NET Infrastructure & Services*. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/net/Default.asp. Bellevue, WA: Microsoft.

12. Mitchell, J. G., et al. 1979. *Mesa Language Manual*. CSL-793. Palo Alto, CA: Xerox PARC.

13. Object Management Group. 2001. *Unified Modeling Language Specification*. http://www.omg.org/technology/documents/formal/uml.htm. Framingham, MA: Object Management Group.

14. The Open Group. 1997. *DCE 1.1: Remote Procedure Call*. Technical Standard C706. http://www.opengroup.org/publications/catalog/c706.htm. Cambridge, MA: The Open Group.

15. Prescod, P. 2002. *Some Thoughts About SOAP Versus REST on Security.* http://www.prescod.net/rest/security.html.

16. Red Hat, Inc. 2003. *The bzip2 and libbzip2 Home Page.* http://sources.redhat.com/bzip2. Raleigh, NC: Red Hat, Inc.

17. Schmidt, D. C. et al. 2000. "Leader/Followers: A Design Pattern for Efficient Multi-Threaded Event Demultiplexing and Dispatching". In *Proceedings of the 7th Pattern Languages of Programs Conference*, WUCS-00-29, Seattle, WA: University of Washington. http://www.cs.wustl.edu/~schmidt/PDF/lf.pdf.

18. Sleepycat Software, Inc. 2003. *Berkeley DB Technical Articles*. http://www.sleepycat.com/company/technical.shtml. Lincoln, MA: Sleepycat Software, Inc.

19. Snader, J. C. 2000. *Effective TCP/IP Programming*. Reading, MA: Addison-Wesley.

20. Stroustrup, B. 1997. *The C++ Programming Language*. Reading, MA: Addison-Wesley.

21. Sutter, H. 1999. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Reading, MA: Addison-Wesley.

22. Sutter, H. 2002. "A Pragmatic Look at Exception Specifications." *C/C++ Users Journal* 20 (7): 59–64. http://www.gotw.ca/publications/mill22.htm.

23. Unicode Consortium, ed. 2000. *The Unicode Standard, Version 3.0*. Reading, MA: Addison-Wesley. http://www.unicode.org/unicode/uni2book/u2.html.
24. Viega, J., et al. 2002. *Network Security with OpenSSL*. Sebastopol, CA: O'Reilly.
25. World Wide Web Consortium, 1999. *HTML 4.01 Specification*. http://www.w3.org/TR/html401. Boston, MA: World Wide Web Consortium.
26. World Wide Web Consortium. 2002. *SOAP Version 1.2 Specification*. http://www.w3.org/2000/xp/Group/#soap12. Boston, MA: World Wide Web Consortium.
27. World Wide Web Consortium. 2002. *Web Services Activity.* http://www.w3.org/2002/ws. Boston, MA: World Wide Web Consortium.
28. Garcia-Molina, H. 1982. "Elections in a Distributed Computing System." *IEEE Transactions on Computers* 31 (1): 48-59. http://doi.ieeecomputersociety.org/10.1109/TC.1982.1675885