# The Internet Communications Engine

# Object-oriented Middleware

- Used by the computing industry since  the mid-nineties
- The middleware platform takes care of the majority of networking chores
  - Marshaling and unmarshaling
  - Mapping logical object addresses to physical transport endpoints
  - Changing the representation of data according to the native machine architecture
  - Automatically starting servers on demand.

# Object-oriented Middleware

- Binary protocols
  - DCOM - a Microsoft-only solution, superceded by .NET
    - Scales badly to large numbers (hundreds of thousands or millions) of objects, largely due to the overhead of its distributed garbage collection mechanism.
  - CORBA - available from a variety of vendors, standardized by OMG
    - Some problems with interoperability
    - Excessive complexity
- XML based, standardized by w3c
  - SOAP (Simple Object Access Protocol)
    - Very serious performance penalties on applications, both in terms of network bandwidth and CPU overhead
    - Lack of higher-level abstractions
  - REST (Representational State Transfer)
    - Data access

# The Internet Communications Engine

- Developed by ZeroC, Inc.
- The main design goals of Ice:
  - 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.
  - 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.

# The Ice Architecture

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

# Ice Objects

- 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.
- 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.

# Ice Objects

- An Ice object has a distinguished interface, known as its main interface.
    - 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.

# 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:
    - Locates the Ice object
    - Activates the Ice object’s server if it is not running
    - Activates the Ice object within the server
    - Transmits any in-parameters to the Ice object
    - Waits for the operation to complete
    - Returns any out-parameters and the return value to the client
      - or throws an exception in case of an error

# Servants

- An Ice object is a conceptual entity that has a type, identity, and addressing information.
- Client requests ultimately must end up with a concrete server-side processing entity that can provide the behavior for an operation invocation.
- 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.
    - It 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.

# 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
    - Inform the client with an appropriate exception that it cannot deliver the request
    - Under no circumstances is a request delivered twice

# Different Ways of Method Invocation

- Synchronous Method Invocation
- Asynchronous Method Invocation
- Oneway Method Invocation
- Batched Oneway Method Invocation
- Datagram Invocations
- Batched Datagram Invocations

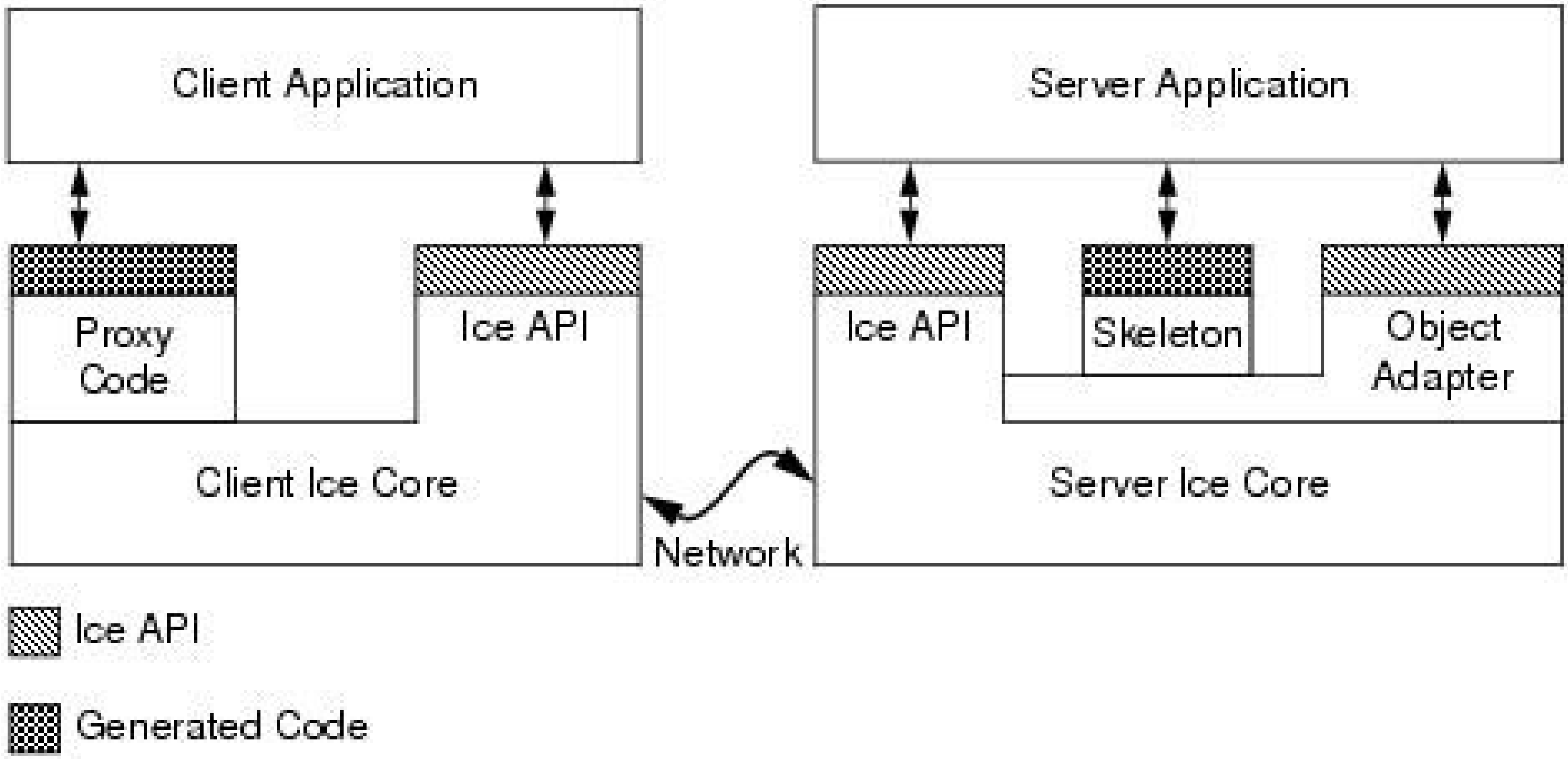
# Slice

- Specification Language for Ice
- 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.

# Language Mappings

- The rules that govern how each Slice construct is translated into a specific programming language are known as language mappings.
  - For the C++ mapping a Slice sequence appears as an STL vector
  - For the Java mapping 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.

# Client and Server Structure



# The Ice Protocol

- Ice provides an RPC protocol that can use either TCP/IP or UDP as an underlying transport.
  - 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.

# The Ice Protocol

- Ice supports compression on the wire: by setting a configuration parameter, you can arrange for all network traffic to be compressed to conserve bandwidth.
- The Ice protocol is suitable for building highly-efficient event forwarding mechanisms as it permits forwarding of a message without knowledge of the details of the information inside a message.
- 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.

# Ice Services

- Freeze
  - Built-in object persistence service.
- IceGrid
  - Ice location service that resolves the symbolic information in an indirect proxy to a protocol – address pair for indirect binding.
- IceBox
  - A simple application server that can orchestrate the starting and stopping of a number of application components
- IceStorm
  - A publish–subscribe service that decouples clients and servers.
- IcePatch2
  - A software patching service.
- Glacier2
  - The Ice firewall traversal service.

# Architectural Benefits of Ice

- Object-oriented semantics
- Support for synchronous and asynchronous messaging
- Support for multiple interfaces
- Machine independence
- Language independence
- Implementation independence
- Operating system independence
- Threading support
- Transport independence
- Location and server transparency
- Security
- Built-in persistence
- Source code availability

# A Hello World Application

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

```
$ slice2cpp Printer.ice
```

- The **`slice2cpp`** compiler produces two C++ source files from this definition, **`Printer.h`** and **`Printer.cpp`**.
  - **`Printer.h`** - 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 source code for our Printer interface.
    - The generated source contains type-specific run-time support for both clients and servers.

# The Server

```cpp
#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;
}
```

```cpp
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;
}
```

```
$ g++ -I. Printer.cpp Server.cpp -lIce
```

# The Client

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

using namespace std;
using namespace Demo;

int
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;
}
```

```
$ g++ -I. Printer.cpp Client.cpp -lIce
```