symbian Academy

Sockets

Sockets

This lecture

- Gives an initial overview of the function of sockets in communications
- Describes the sockets implementation on Symbian OS

Including

- The socket server architecture
- The main classes used
- The role and features of protocol module plug-ins

symbian Academy

Introducing Sockets

- Recognize correct high-level statements which define and describe a network socket
- Recognize correct statements about transport independence
- Know the difference between connected and connectionless sockets
- Differentiate between streamed and datagram communication and their relationship with connected/connectionless sockets

Introducing Sockets

Sockets

- Were formally defined by the Lincoln Laboratory MIT
- In the 1971 RFC147 paper

The University of California, Berkeley

- Introduced what would become the de facto sockets API
- The BSD Unix 4.2 release during the 1980s

During this lecture

- We will focus on the network sockets
- Note sockets may be used for a number of different technologies
- Including infrared, USB and Bluetooth

Typical Socket Properties

A socket

- Is a communication endpoint between two or more software processes

Communication is two-way:

- Either a direct peer-to-peer relation
- Where two similar processes communicate
- Or a client–server relationship
- With each of the communicating processes performs a different role
- For example: An internet browser and a web server

Typical Socket Properties

A socket

- Has three characteristics or parameters:

1. The communication domain

- The address family or format of the socket
- For example: an Internet socket (**`KAfInet`**) has an address and port number

2. The socket type

- Typically streaming connected or datagram connectionless
- These are indicated by use of either **`KSockStream`** or **`KSockDatagram`**

3. Transport protocol that is relevant to the domain

- A streaming Internet socket would typically use TCP/IP by use of **`KProtocolInetTcp`**

Typical Socket Properties

A typical TCP/IP Symbian OS socket

- Would be specified as follows:

```
iSocket.Open(iSocketServ,
             KAfInet,
             KSockStream,
             KProtocolInetTcp);
```

More practical details coming up shortly

Connection and Connectionless

A connection-oriented communication protocol

- Establishes an end-to-end connection before any data is sent
- The presence of a connection allows certain guarantees
- As it has a state that is order of delivery, data arrival and error control
- It is often referred to as "reliable".

Connection and Connectionless

A connectionless communication protocol

- Requires the destination address each time data is sent
- It is used for sending datagrams only
- As there is no pre-established end-to-end connection
- There is no of state

Whenever a packet arrives

- It is treated completely independently of the preceding one
- Thus a datagram socket cannot provide guarantees of order, duplication or delivery
- Typically referred to as "unreliable"

Blocking and Non-blocking

Sockets may operate in one of two modes

- Blocking and non-blocking

An operation on a blocking socket

- Is synchronous
- The socket does not return control to the calling programming
- Until it has completed the operation

An operation on a non-blocking socket

- Is asynchronous
- Returns control immediately
- Requires additional infrastructure to monitor the data (arrives/completes)

The OSI Reference Model and Network Sockets

To provide a context

- For how network sockets are used
- We will briefly examine the UDP/TCP/IP layers (Transport and Network)
- Of the Open Systems Interconnection (OSI) reference model

The OSI model is a 7-layer model

- Encapsulating each layer of communication functionality
- Providing the layer above it with meaningful semantics and implementation independence

The OSI Reference Model and Network Sockets

Layer	Example	Units/Representation
Application	HTTP, SMTP	
Presentation	NCP, SMB	
Session	TLS, RPC	
Transport	TCP, UDP	Segments
Network	IP	Packets
Data Link	Ethernet, 802.11 WiFi	Frames
Physical	100BASE-T, WiFi transceiver	Typically bits

The application, presentation and session layers

- For TCP/IP are dealt with as a single entity
- Providing suitable protocols and a relevant API to general applications

The OSI Reference Model and Network Sockets

The transport layer

- Sends and receives segments of data

The network layer

- Further breaks the segments up into packets on transmitting data
- Reassembling packets into segments on receiving data
- Each packet contains addressing information and relevant control information as well as the data payload

The data link layer

- Provides synchronization
- error- and flow control of frames over the physical layer

The physical layer

- Deals with the hardware medium

The OSI Reference Model and Network Sockets

TCP is a transport-level

- Connection-oriented protocol
- Provides built-in flow control for reliable transfer of data
- Between network nodes or processes
- TCP is packaged and sent over the network-layer IP protocol

The OSI Reference Model and Network Sockets

UDP is a transport-level

- Connectionless protocol
- More lightweight than TCP

Does not:

- Any confirmation the packet has arrived at its destination
- Perform any retransmissions on errors
- Handshaking of any kind

UDP is used

- Where speed is an issue and reliability is not crucial
- For example: In network multiplayer games

The OSI Reference Model and Network Sockets

IP is the network-level protocol

- Over which both TCP and UDP are layered
- TCP and UDP packets reside within the data area of an IP packet
- IP is connectionless
- Data is transferred via packets that flow from a source to a destination

Symbian OS includes socket support

- For TCP, UDP and IP for Internet and local network communications

Note

There are other kinds of socket

- For example Bluetooth and infrared

And additional

- Address Families which are not generally supported by Symbian OS

As well as Streaming and datagram socket types there are others including:

- Raw sockets, where the header and data payload is provided
- Sequence, which ensures the correct order without using a streamed connection
- Reliably Delivered Message (RDM)

symbian Academy

The Symbian OS Sockets Architecture

- Demonstrate a basic understanding of the support for sockets on Symbian OS
- Recognize the characteristics of the **`RSocketServ`**, **`RSocket`** and **`RHostResolver`** classes
- Understand the role and purpose of PRT protocol modules

The Symbian OS Sockets Architecture

Symbian OS provides a socket framework

- Which supplies a C++ API similar to the BSD C-based socket API
- Supporting communication using the Internet protocol suite
- Allows for other types of communication including Bluetooth, USB and IR

The lower layers

- Of the communications ("comms") architecture handle any communication differences
- So that the sockets API can be used in a transport-independent way

We will now examine

- The framework and plug-ins architecture
- The main classes used for sockets programming on Symbian OS

Framework and Protocol Module Plug-ins

The Symbian OS comms system provides

- Support for dynamically loaded protocol modules

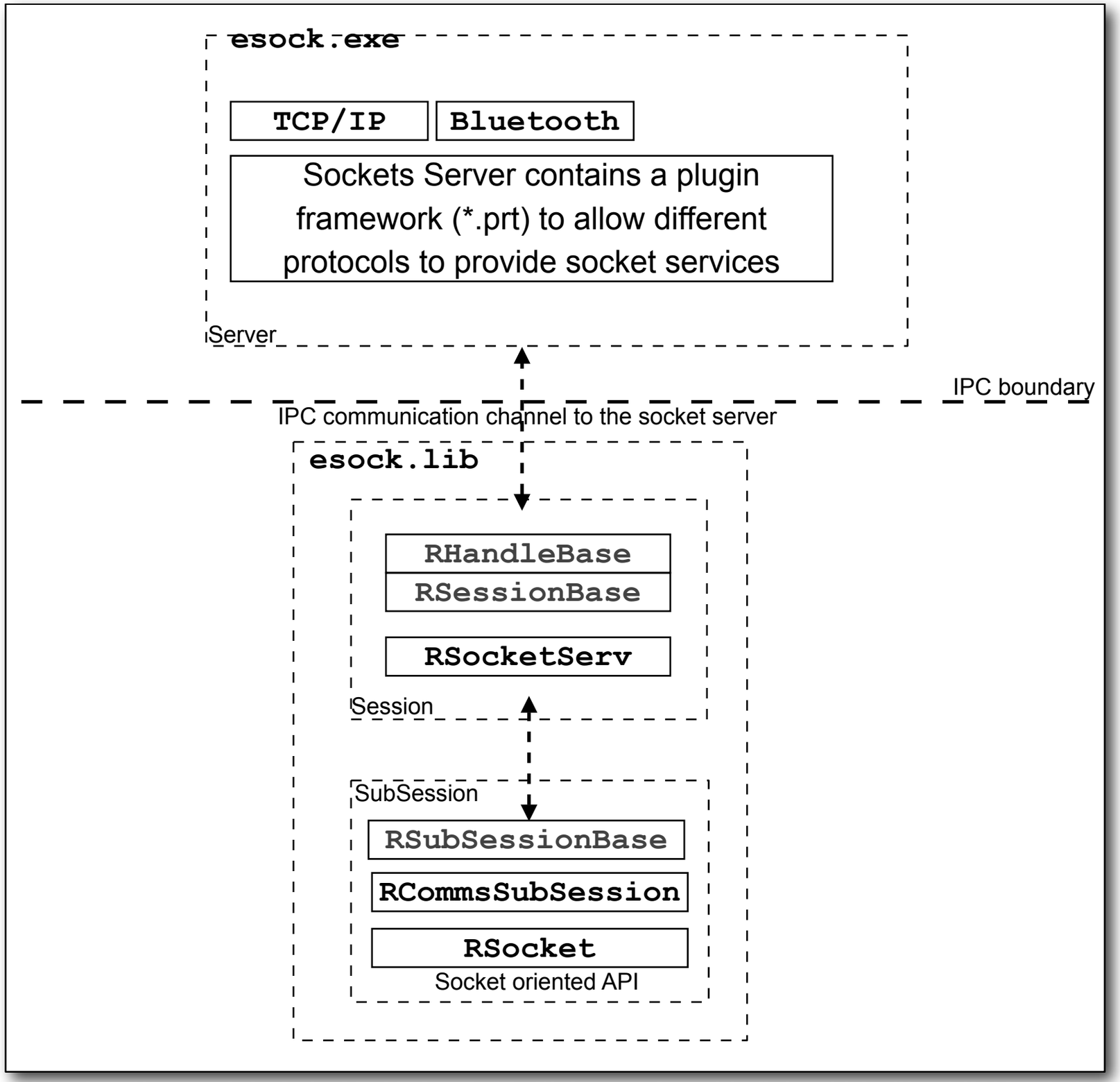
The use of the socket API means

- That application code written for use with one protocol
- Can be modified simply to use a different protocol

The Symbian OS sockets server provides

- Communications between addressable endpoints (sockets)
- Called **`ESOCK.EXE`**

Overview of ESOCK Client-Server Relationship



Framework and Protocol Module Plug-ins

Supports a number of different protocols

- Bluetooth (L2CAP and RFCOMM)
- USB
- IR (IrDA, IrTinyTP and IrMUX)

The protocols are supplied by plug-in DLLs

- Called protocol modules (**`PRT`**s)
- Which the server loads and unloads as required
- One protocol module can contain multiple protocols

For example the **`TCPIP.PRT`** protocol module

- Contains UDP, TCP, ICMP, IP, and DNS
- UDP and TCP are accessible via sockets
- To transfer data over IP

Framework and Protocol Module Plug-ins

The Symbian OS sockets APIs

- Are published in **`es_sock.h`**
- Calling code uses the client-side implementation classes provided by **`esock.dll`**

The socket server's client-side APIs

- Make asynchronous calls to the server
- Which coordinates client access to the socket services
- Protocol modules provide support for the particular networking protocol requested

Framework and Protocol Module Plug-ins

In addition to the following

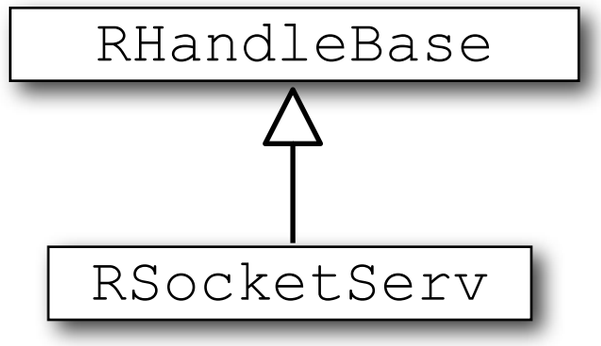
- Connecting to sockets
- Hostname resolution
- Reading and writing data

The sockets framework also supports

- Querying for protocol information
- Allowing calling code to determine
- Which sockets protocols are supported on a phone

RSocketServ

RSocketServ



- Is the client-side implementation class
- Which communicates with the socket server
- Analogous to the **RFs** client-side class for file server communication

RSocketServ

For code to make requests to the socket server

- An object of **RSocketServ** must be instantiated
- A session with the socket server established by calling **Connect()**

The prime use for instances of **RSocketServ**

- Is to establish subsession communications
- For **RSocket** and **RHostResolver**

RSocketServ

Any of the **RSocket** objects

- Which are opened using the session are automatically closed
- When the session is terminated through a call to **RSocketServ::Close()**
- Cannot be re-used

However

- It is recommended to call **Close()** on each subsession object
- Before closing the session

RSocketServ can be used

- To pre-load a **PRT** by calling the asynchronous function
- **RSocketServ::StartProtocol()**

RSocketServ

However

- Client programs do not normally need to call `RSocketServ::StartProtocol()`
- As loading a protocol is managed automatically by the sockets server
- When a socket of that protocol is opened

Some applications

- May need to ensure that an open socket call will not take a significant amount of time
- So make use of `StartProtocol()`
- For example applications using IrCOMM

RSocketServ

RSocketServ::GetProtocolInfo()

- Can be used to acquire a comprehensive description of a protocol's capabilities and properties
- Returned in a **TProtocolDesc** object

RSocketServ::NumProtocols()

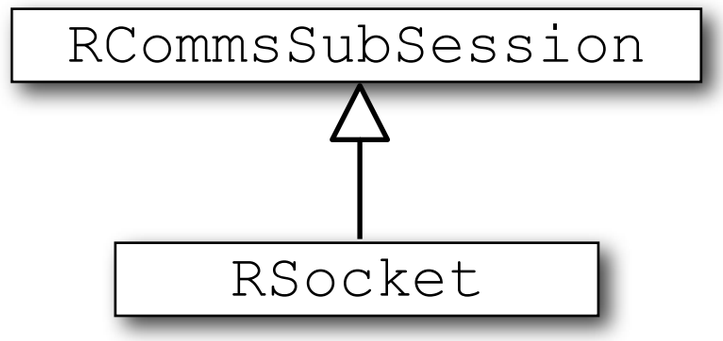
- Can be used to acquire the number of protocols the socket server is currently aware of

RSocketServ

- Is not used to send/receive data or establish connections
- The **RSocket** subsessions provide APIs to invoke these functions

RSocket

RSocket



- Is the endpoint for all socket-based communications
- The class is a subsession of **RSocketServ**
- Each object instantiated represents a single socket

RSocket

The methods of **RSocket** correspond

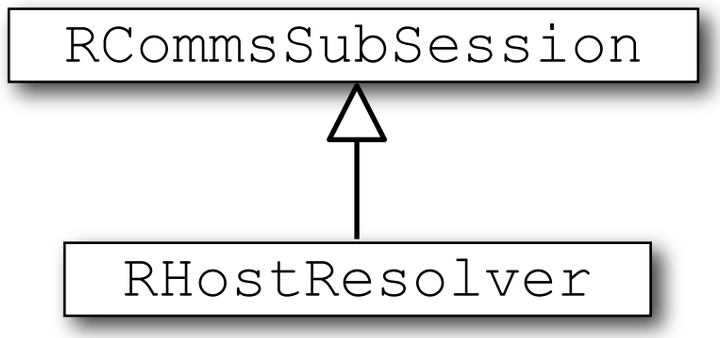
- For the most part with the BSD network API functions

The client socket interface allows:

- Socket opening
- Active connecting
- Data read from and write to a protocol
- Passive connections (through the listen/accept model)

RHostResolver

RHostResolver



- Is used for hostname resolution
- Providing a generic interface to protocol-specific host-resolution services

RHostResolver

For example

- IP addresses are hard for people to remember
- So domain names, are used instead
- It is much easier to remember www.yahoo.com
- Than it is to remember 216.109.118.77!

The hostname resolution service

- Used in the internet is the Domain Name System (DNS)
- DNS translates human-readable domain names to IP addresses

RHostResolver

The **RHostResolver** class

- Provides methods for getting an IP address given a domain name
- And a domain name given an IP address

Thus **RHostResolver::GetByName()**

- Converts the server name to an IP address

RHostResolver

For Bluetooth and infrared

- The resolution interface can be used to discover
- Which other devices are available to communicate using those protocols

Queries made through **RHostResolver** objects

- Are packaged in **TNameEntry** descriptors
- Which hold **TNameRecord** objects
- Containing the host name and address

RHostResolver

RHostResolver

- Also provides functions to allow
- Getting and setting the hostname of the local device

Since the interface is generic

- Implementation is provided by each protocol module individually
- Not all protocol modules provide all services offered by **RHostResolver**

RHostResolver

If the protocol does not support a given operation

- Functions return **KErrNotSupported**

RHostResolver is a subsession

- Of an active socket server session
- A host resolver subsession is opened for a specific protocol by passing an appropriate identifier

symbian Academy

Using Symbian OS Sockets

- Recognize correct patterns for opening and configuring connected and connectionless sockets
- Know which RSocket API methods should be used for connected and unconnected sockets to send and receive data
- Know the characteristics of the synchronous and asynchronous methods for closing an RSocket subsession

Initialization and Socket Opening

Before an **`RSocket`** subsession can be opened

- A session with the socket server must be created
- Through a call to **`RSocketServ::Connect()`**

Once a server session is acquired

- A socket can be opened by calling one of the overloads of **`RSocket::Open()`**
- Each of which takes the connected socket server session as a parameter

Other parameters that can be specified include

- The protocol type
- The socket type (stream-interface or datagram)
- The address family of the socket

Configuring and Connecting Sockets

Once a socket is open

- It is configured differently
- Depending on whether the socket is connectionless or connection-oriented

Connectionless sockets

- Are configured with a local address
- Use special versions of I/O calls
- That include the remote socket address

The `RSocket::Bind()` method

- Assigns the local address to a connectionless socket

Configuring and Connecting Sockets

Before it is possible to send or receive data

- A call to **`RSocket::Open()`** followed by **`RSocket::Bind()`** must occur

Once the connectionless socket is bound to a local address

- It is ready to send or receive data
- Using **`SendTo()`** or **`RecvFrom()`**
- Each specifying the remote address

Configuring and Connecting Sockets

Connected sockets

- Are configured with the address of the remote socket
- So that they remain tied together for the duration of the connection

A client socket makes a call

- To the asynchronous **`RSocket::Connect()`** method
- Passing in the address to connect to the remote socket
- Waiting for the remote side to complete the connection

If a socket is unbound

- That is if **`Bind()`** has not yet been called on it
- It will automatically have a local address assigned to it

Configuring and Connecting Sockets

A server socket

- Must use two sockets

One bound to a local address

- By a call to **`Bind()`**
- Is used to listen for an incoming connection from a client
- Using **`RSocket::Listen()`**

A second blank socket

- Is passed to **`RSocket::Accept()`**
- To accept a connection once it has been detected

Configuring and Connecting Sockets

The initial `Listen()` method

- Is synchronous
- Sets up the bound socket with a queue
- Used for collecting connection requests

The `Accept()` method

- Is asynchronous
- Waiting for incoming connections

When the asynchronous connection completes

- The blank socket is given the handle of the new socket
- It may then be used to transfer data

Configuring and Connecting Sockets

It is also possible to call **`Connect()`**

- On a connectionless socket

The **`Bind()`** call can be omitted

- **`Send()`**, **`Write()`** and **`Recv()`** can be used instead

This allows

- Software initially written to use connection-oriented protocols
- To be quickly ported to use a connectionless one

RSocket::RecvFrom()

Reading and writing with connectionless sockets

- Requires the remote address to be specified in the I/O request

RSocket::RecvFrom() method should be used

- To read from a connectionless socket
- Rather than **Read()**, **Recv()** or **RecvOneOrMore()** methods
- Which should only be used with connected sockets

RSocket::RecvOneOrMore()

RSocket::RecvOneOrMore()

- Is an asynchronous request
- Completes when any data is available from the connection

The receive buffer

- Is specified as an 8-bit descriptor
- The received data is added to this buffer
- The size of the descriptor is updated
- To match the number of bytes received

Reading from Sockets

For Stream-interfaced sockets

- Such as TCP, **`RSocket::Recv()`** will not complete
- Until the entire descriptor is filled with data
- Specified by the maximum length of the receive descriptor

Unlike **`RecvOneOrMore()`**

- Which completes when any amount of data is received
- Unless it is known how much data will be received

`Recv()` should not be used for TCP

- Or other stream-interfaced protocols

RSocket::RecvFrom()

RSocket::RecvFrom()

- Receives UDP data and supplies:
- The data received
- The address of the endpoint that sent the data

All reading methods

- Are asynchronous
- And supply a buffer to receive the data

Where there is a cancellation method

- The state of a socket after a read has been cancelled
- Is defined by the characteristics of the protocol

RSocket::SendTo()

RSocket::SendTo()

- Is used to write to a connectionless socket

Which has these parameters

- The address to which to send the datagram
- A buffer of data
- Flags to control the transfer

Blocking behavior is controlled

- By passing protocol-specific flags to the call
- Indicating whether to wait for a receipt

RSocket::Write()

RSocket::Write()

- Is used to write to connected sockets
- Does not need address information for the remote endpoint
- It is an asynchronous method
- Taking a descriptor parameter
- Passing the entire buffer to the remote socket

RSocket::Send()

RSocket::Send()

- Can also be used for connected sockets
- As an alternative to **RSocket::Write()**

Various **Send()** overloads

- Allow for control over the amount of data sent from the buffer
- And for passing flags to the underlying protocol module
- To configure the I/O

All implementations provided

- For reading from and writing to connected sockets are asynchronous
- Each has a corresponding cancellation method

Closing Sockets

When a socket is closed

- The protocol layers in the connection may also shut down
- A socket may have pending connection requests
- Or buffers with data ready to be retrieves

`RSocket::CancelAll()`

- Called to close the socket gracefully
- It cancels any outstanding asynchronous operations waiting on data or connections

RSocket::Close()

RSocket::Close()

- Used to close the socket synchronously
- Causing the operating system to release any resources dedicated to it
- This is the typical way to close a connectionless socket
- It is recommended when there are no pending operations or buffered data waiting.

Closing Sockets

For connection-oriented protocols

- It is usual to disconnect before closing the socket

`RSocket::Shutdown()`

- Is asynchronous
- Takes a flag to indicate how to close the socket
- Used to disconnect the connection

For example

- To drain the socket
- By waiting for both output and input buffers to empty
- Or to stop input but drain the output

Socket shutdown cannot be cancelled once it has been called

symbian Academy

Sockets

- ✓ Introducing Sockets
- ✓ The Symbian OS Sockets Architecture
- ✓ Using Symbian OS Sockets