



Contents

- [1 Introduction](#)
- [2 Sockets server architecture](#)
 - ◆ [2.1 What is a socket?](#)
 - ◆ [2.2 Protocol modules](#)
 - ◆ [2.3 Transport independence](#)
 - ◆ [2.4 Client-server interface](#)
- [3 The main sockets server classes](#)
- [4 Using RSocketServ](#)
 - ◆ [4.1 Establishing a session with the sockets server](#)
 - ◆ [4.2 Pre-loading a protocol module](#)
- [5 Using RSocket](#)
 - ◆ [5.1 Establishing communication between endpoints](#)
 - ◇ [5.1.1 Connected or connectionless sockets?](#)
 - ◇ [5.1.2 Streams and datagrams](#)
 - ◆ [5.2 Datagram lifecycle](#)
 - ◆ [5.3 Stream lifecycle](#)
 - ◆ [5.4 Creating a socket](#)
 - ◆ [5.5 Binding a socket](#)
 - ◆ [5.6 Listening for connections](#)
 - ◆ [5.7 Accepting connections](#)
 - ◆ [5.8 Connection process for datagrams](#)
 - ◆ [5.9 Closing a socket](#)
 - ◆ [5.10 Shutting down a socket](#)
 - ◆ [5.11 Multi-threaded sockets applications](#)
- [6 Host resolution services](#)
 - ◆ [6.1 What is host resolution?](#)
 - ◆ [6.2 Using RHostResolver](#)
 - ◆ [6.3 Domain Name Service \(DNS\)](#)
- [7 Using active objects in sockets code](#)
- [8 Code example: connecting sockets](#)
 - ◆ [8.1 Server ?listening? class definition](#)
 - ◆ [8.2 Getting ready to receive a client connection](#)
 - ◆ [8.3 Handling a connection request](#)
 - ◆ [8.4 Using the connected socket](#)
- [9 Transferring data](#)
 - ◆ [9.1 Receiving data](#)
 - ◇ [9.1.1 Using unconnected sockets](#)

- ◊ [9.1.2 Using connected sockets](#)
- ◆ [9.2 Sending data](#)
 - ◊ [9.2.1 Using unconnected sockets](#)
 - ◊ [9.2.2 Using connected sockets](#)
- [10 Summary](#)

Introduction

This article gives an introduction to the Symbian OS sockets API. It's aimed at developers who need to add communications abilities to their applications using sockets, and it provides both a theoretical overview and practical code examples.

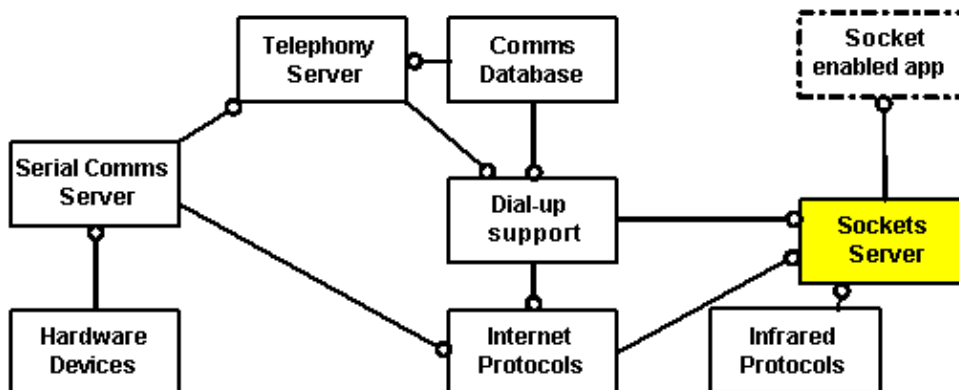
This article includes:

- An overview of the relevant components involved in socket communications.
- An overview of the sockets server architecture and the use of the two main API classes, *RSocketServ* & *RSocket*.
- A discussion of the process of creating a connection between two end-points.
- A discussion of the different modes of communicating between sockets: either as a flow of data or as a series of discrete messages.
- A practical example of how to use active objects to connect sockets.

Sockets server architecture

This article looks at adding communications capabilities to applications through the use of the sockets server. However, computer communications systems are highly complex and it should be understood that the sockets server is a reasonably high-level component in its own right. A lot of lower level software is required to allow the sockets server to achieve its role.

The diagram below shows how the sockets server fits in with the other Symbian OS communications components.



Symbian OS communications components

Firstly, the transport layer protocols need to be considered. The diagram above includes Internet and infrared protocols, and Symbian OS v6.0 also contains support for Bluetooth® wireless technology.

When we talk about using Internet protocols, there is an implicit reliance on dialing an Internet service provider. So, Symbian OS offers support for dial-up network access, which is also shown in the diagram above. In this case, the most significant system component is the telephony server.

Ultimately, we need to talk to a hardware device to send and receive data to and from our chosen network. The diagram above focuses on dial-up access to a network using Internet protocols and shows how the serial comms server fits into the communications system. The serial comms server talks, through specific device drivers, to hardware devices and the outside world.

What is a socket?

What is a socket? The classic quote from the Berkeley UNIX implementation of sockets is that "a socket is an endpoint for communication".

Which means what, exactly?

A socket represents the logical end of a communication "channel". A socket is a combination of a physical machine network address and a logical port number to which another socket somewhere can transmit data.

Because a socket is identified by a machine address and a port number, each socket is uniquely identified on a particular computer network. This allows an application to uniquely identify another location on the network with which to communicate.

Note that it is perfectly possible for a conversation to occur between two sockets on the same machine " in this case each socket will have the same machine address, but different port numbers.

The machine network address and port number combination is protocol dependent. The classic use of sockets is to communicate over a network running Internet Protocol (IP), but the sockets server supports a number of other protocols, which will be mentioned shortly.

As we'll see, we can use the same high-level sockets API irrespective of our chosen communications protocol (that is, transport layer).

Protocol modules

As has been stated, the classic use of sockets is to allow communication between logical points on a computer network running the TCP/IP suite of protocols. The most famous network running TCP/IP is, of course, the Internet.

Using_the_sockets_API

Most implementations of sockets are indeed limited to communication over networks running TCP/IP.

The Symbian OS sockets server goes much further, however, and features an architecture supporting plug-in protocol modules. This allows Symbian (and development partners) to extend the lifetime of the sockets server and socket-enabled applications.

As new protocols and transport layers are introduced, the sockets server can adapt to the changing environment by the addition of a new protocol module which knows how to communicate using the new ?language? or protocol.

The sockets server supplied with Symbian OS Release 5 supports TCP/IP and IrDA protocol suites. In Symbian OS version 6.0, support was added for Bluetooth® wireless technology and SMS.

Protocol modules are standard Symbian OS DLLs. They have a UID2 of KUidProtocolModule (0x1000004A), and typically have a file extension *.PRT.

As an aside, note that the sockets server also supports PLP ? or Psion Link Protocol. PLP is used to communicate between Symbian OS phones and computers running Microsoft Windows. PLP is used by Symbian Connect ? currently better known by its Psion Computers-badged name of ?PsiWin?.

The sockets server can load protocol modules in two ways:

- Most commonly, protocol modules will be loaded the first time a socket is opened using a protocol which is not currently loaded.
- Alternatively, applications are allowed to load protocol modules explicitly. This is useful because loading a protocol module can take a significant amount of time. The API to do this will be covered later in this article.

Note that one protocol module can contain multiple protocol implementations. For example, the protocol module TCPIP.PRT contains implementations of UDP, TCP, ICMP, IP & DNS. Individual protocols are mapped to their protocol modules by files with the extension .esk in \system\data\. Each protocol module has an .esk file specifying which protocols the module contains, and the index of each protocol within the module.

Transport independence

As already stated, the sockets server features a plug-in architecture allowing new protocol modules to be added to a Symbian OS phone at any time.

This architecture allows the sockets server to implement the concept of transport independence. By offering a generic core sockets API applicable to all data transport systems and allowing protocol modules to add protocol specific functionality to the sockets server, application programmers can be insulated from having to rewrite large parts of their communications subsystems.

Over time, as new protocols appear, protocol modules will be written to offer a sockets interface to these new protocols. Application programmers then just need to add code to deal with any unique properties or behaviors of that protocol to add support for the new protocol to their application. The sockets server will handle the mechanics of communicating using the new protocol, with assistance from lower level

communications components within the operating system as required.

Overall, the sockets server allows application programmers to write applications which can communicate over many protocols while maintaining a core API which limits the amount of protocol specific code an application programmer needs to write.

Client-server interface

Symbian OS features a micro-kernel, by which we mean that only the core services required to interact with the hardware and to control the machine run kernel side. A large number of system services are supplied by user mode server threads, commonly referred to as the "system servers".

The sockets server is one of these system servers, and applications interact with it through a published client API which features a number of classes, of which the four most important are:

- **RSocketServer**: This class establishes and obtains necessary resources for the connection to the sockets server. In client-server terminology, this class represents the application's session with the sockets server. All other client interface classes require a session to the server to be opened via an instance of this class.
- **RSocket**: This class represents a socket. A typical application will have several instances of RSocket operational at any time.
- **RHostResolver**: This class offers an interface for host name resolution services.
- **RNetDatabase**: This class offers an interface for network database access.

RSocket, *RHostResolver* & *RNetDatabase* all represent sub-sessions within the context of a given application's session to the sockets server - that is, an *RSocketServer* instance.

The main sockets server classes

The sockets server offers two main classes as its client side API.

- **RSocketServ**: Each application thread that wishes to use sockets requires an instance of this class to establish a connection (a session) to the sockets server.
- **RSocket**: Each application thread that wishes to use sockets also requires one or more *RSocket* objects - these are subsessions.

The next two sections introduce the key features of both the session and subsession classes (*RSocketServ* & *RSocket*) which form part of the client side API exposed by the sockets server.

Using RSocketServ

The RSocketServ class is important because it represents a client application's session with the sockets server.

However, a client application will not use this class to send or receive data or to create a connection with a remote endpoint. The class RSocket is used by client applications for these purposes and is covered shortly.

RSocketServ does allow client applications to query the sockets server to determine the number of protocols it knows about, and return information about each of these protocols.

A client application wishing to use sockets will need to have one instance of an RSocketServ object which will be used to establish a session with the sockets server. Each individual socket is represented by a separate RSocket instance. An application's RSocketServ object effectively acts as a container for the RSocket instances.

The two main methods offered by RSocketServ are Connect() and StartProtocol().

Establishing a session with the sockets server

The Connect() method is used to create the application's session with the sockets server. It takes one parameter - the number of message slots available to this session.

```
TInt Connect (TUint aMessageSlots);
```

The number of message slots limits the number of simultaneous asynchronous operations this application can ask the sockets server to perform. All synchronous operations consume one slot and asynchronous operations consume one slot while the operation is pending.

A typical socket will perform asynchronous read and write operations meaning that a typical socket will require two message slots. You do not need to specify an additional slot if the socket can also perform synchronous operations, because a slot for synchronous operations is provided by the client-server framework. You should determine the total number of sockets your application will use at any one time, from which you can deduce the total number of message slots your session with the sockets server is going to need.

If you don't specify any value, you will get the default number of message slots, KESockDefaultMessageSlots (0x08).

Pre-loading a protocol module

The sockets server will automatically load a protocol module on demand when the first socket requiring a given protocol is created. However, loading a protocol can take a significant amount of time, and RSocketServ provides a method, StartProtocol(), to pre-load protocol modules so that they are ready for use before a socket requiring that protocol is created.

If your application wishes to pre-load protocol modules, rather than allowing the sockets server to load the modules on demand, use the StartProtocol() method, which has the following prototype:

```
void StartProtocol (TUInt aFamily, TUInt aSockType,  
                  TUInt aProtocol, TRequestStatus& aStatus);
```

The StartProtocol() method takes a number of parameters: the protocol family (e.g.,KAfInet), the type of socket that will use this protocol (e.g.,KSockStream), a protocol from the chosen protocol family (e.g.,KProtocolInetTcp), and finally an asynchronous completion status. The meaning of these parameters will be explained shortly.

Note that, even though StartProtocol() is an asynchronous service, it cannot be cancelled.

Using RSocket

The class RSocket represents a single socket used by the application. Each individual socket an application wishes to create requires a separate RSocket instance. In practice, application code will spend more time using the RSocket class than the RSocketServ class.

RSocket is a large class that provides many services, including:

- Connection services, as a client or a server
- Setting or querying its own address, or querying remote address
- Reading data from the socket
- Writing data to the socket
- ... and more

You must have an active RSocketServ session before opening any sockets. Also, before any of the services indicated above can be used, the socket must be opened. As part of opening a socket, the RSocket sub-session object is associated with a sockets server session ? that is, an instance of the RSocketServ class.

The following sections introduce the available RSocket methods by looking at tasks involved in writing a socket-enabled application.

Establishing communication between endpoints

We now know that we need an `RSocketServ` instance, our session to the sockets server, and a number of `RSocket` objects, each representing a different communication endpoint to our application.

How do we now establish communication between an endpoint in our application and whatever remote endpoint we wish to talk to? Well, this depends on a number of factors.

Firstly, we need to know how data is to flow between the two endpoints. Data can be transmitted from endpoint to endpoint as a flow of bytes, a stream, or as a series of discrete messages (datagrams). The difference between a stream and a datagram will be looked at shortly.

Secondly, in each end-to-end communication, one party is acting as a client and the other as a server. We need to know what role our application plays in this relationship. How we establish communication between two endpoints varies depending on our role as a client or a server.

(Note: the terms "client" and "server" here refer to the role of the application in the communication process, and should not be confused with the terms client and server when referring to the client-server architecture.)

We will now look at the difference between connected and connectionless sockets, streams and datagrams.

Connected or connectionless sockets?

When we create a socket, we specify whether we want the socket to operate in a connectionless or connected mode. This is done by passing a parameter to `RSocket::Open()` as we shall see shortly.

If we operate in a connectionless mode, we send a number of individual messages to a given address (another endpoint). This can be compared to sending postcards to a friend through the mail.

If we operate in a connected mode, we maintain a logical connection between two endpoints and send a stream of bytes during the lifetime of our connection. The address of the remote endpoint is specified once during the connection process. Thereafter, while the connection is open, we don't need to specify the address again while sending data. This can be compared to picking up the phone and dialing a friend.

As it happens, a connection is effectively a series of individual messages which the connected protocol addresses and routes for us.

Connectionless sockets can be used to broadcast messages to multiple addresses, whereas a connected socket talks exclusively to a single remote endpoint for the duration of a given connection.

Streams and datagrams

Although when we create a socket, we specify whether we want the socket to operate in a connectionless or connected mode, what we are really specifying is whether the socket should send data as datagrams or as a

stream.

Datagrams are used with connectionless sockets. A datagram is a single logical message which we need to address so the physical network knows where to deliver our message to. There is no logical connection between the sending and receiving endpoints. For this reason, you could view a datagram as a postcard. Write the message on the postcard, address it and stick it in the mail. When you send a datagram you do not know whether or not your intended recipient actually receives the message.

Streams are used with connected sockets. With a stream, we need first to establish a logical connection between the two endpoints. Having established a connection, we have a reliable, ordered flow of data between the two endpoints. Rather than sending messages we send a flow, or stream, of bytes between the two endpoints. A connected socket can detect and correct errors in the dataflow and we always know whether or not our data has been received by the endpoint.

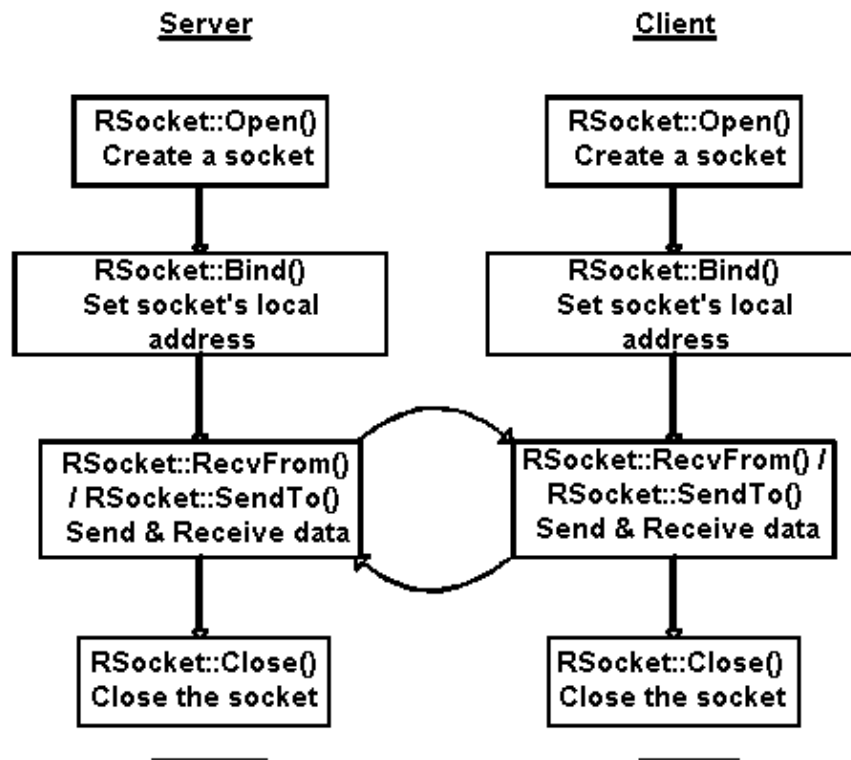
Because a connection exists between the two endpoints, there is no need to supply the receiving address every time we transmit data through a connected socket.

As part of the in-built error detection, lost connections can also be detected.

The term "logical connection" has been used several times here. This is deliberate - our reliable, ordered data flow is actually implemented as a series of datagrams, each carrying information about its position in the data flow, among other control data.

Datagram lifecycle

The diagram below shows the lifecycle of a typical data exchange using datagrams.



Lifecycle for data exchange using datagrams

Note that there is no connection, logical or physical, between the client application and the server application. Data flows between the client and the server as a series of discrete messages, or datagrams.

What these datagrams mean is completely under the control of the applications at either end of the data flow.

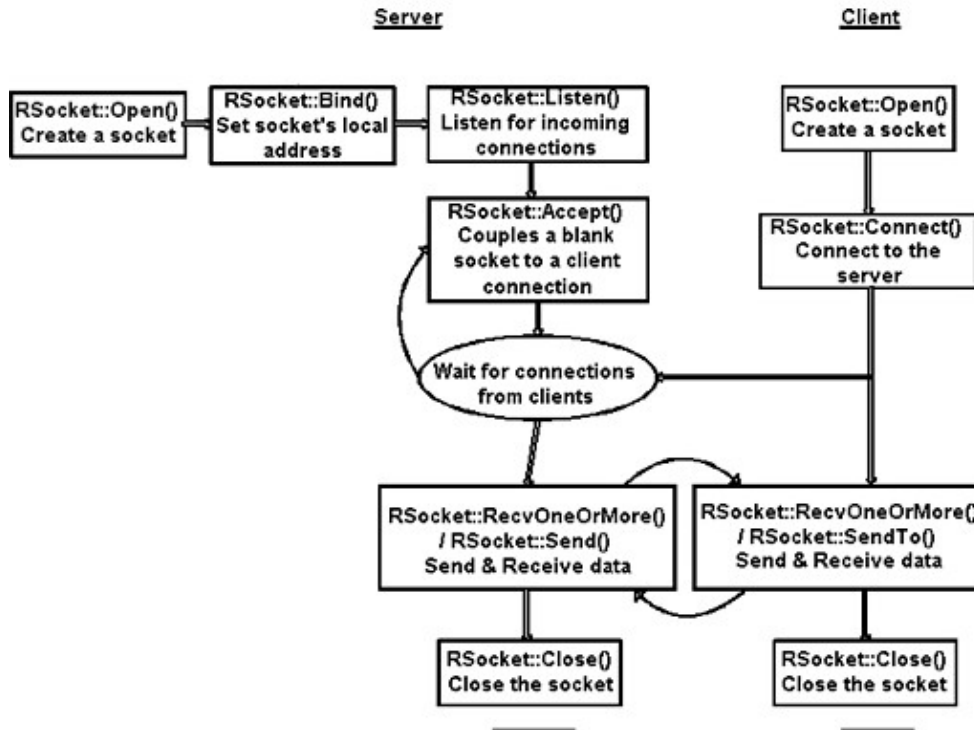
The datagram protocol offered by the TCP/IP protocol suite is called UDP (User Datagram Protocol, defined in RFC768).

It is normal for an application that uses datagrams to add an ?application layer protocol? over the raw message transfer offered by UDP. The application layer protocol can be as complex as the application requires, but it is common to add acknowledgment and logical flow control services.

Stream lifecycle

The diagram below shows the lifecycle of a typical data exchange using streams.

Using_the_sockets_API



Lifecycle for data exchange using streams

In this case, a logical connection is established between the client application and the server application.

The client will use one socket for the connection. The server will actually use a number of sockets.

The first of the server sockets listens for incoming connections from new clients. When a connection request is received, a new socket will be created to handle the communication with the new client. The listening socket continues to listen for further client connection requests.

Having created the connection, data flows between the two endpoints as a continuous stream of bytes.

The stream protocol offered by the TCP/IP protocol suite is called TCP (Transmission Control Protocol, defined in RFC793).

TCP adds flow control and error correction to the data exchange process and frees the application from having to worry about ?proof of delivery? and detecting broken connections.

However, because data flows as a continuous flow of bytes, any record boundaries will be lost when using a stream. The applications at either end will have to replace record boundaries, if appropriate.

Creating a socket

Whatever your application is trying to do with sockets, you always need to create a socket before it can be used. The method `RSocket::Open()` is used to create a socket.

Using_the_sockets_API

```
TInt RSocket::Open (RSocketServ& aServer, TInt addrFamily,  
    TInt sockType, TInt protocol);
```

As already discussed, sockets are sessions within a sockets server client session. Therefore, when an application calls `RSocket::Open()` it needs to associate the new socket with an owning server session. All variations of the `Open()` method take an `RSocketServ&` as their first parameter.

Next, in the prototype shown above, we need to specify the address family our socket will use, e.g., the Internet address family, specified by the constant identifier `KAfInet (0x0800)`. The following table shows the available protocol families and their associated identifiers:

```
TCP/IP KAfInet = 0x0800  
IrDA KIrdaAddrFamily=0x100  
Bluetooth® wireless technology KBTAddrFamily = 0x101  
PLP KFamilyPlp = 273  
SMS KSMSAddrFamily=0x010  
WAP KWAPSMSAddrFamily=0x011
```

As already discussed, sockets can operate in either a connected or connectionless mode ? or, more accurately, they can communicate using either streams or datagrams. The `sockType` parameter is used to indicate the type of socket we want to create. Two values can be used here: `KSockDatagram` or `KSockStream`, as appropriate.

Finally, we can specify which protocol we wish our socket to use. The protocol module `TCPIP.PRT` implements a number of protocols, such as UDP & TCP for example, which can be manually specified by use of the following identifiers: `KProtocolInetUdp` or `KProtocolInetTcp`. However, it is common to leave the selection of the protocol to the sockets server. This is done by specifying the protocol as `KUndefinedProtocol`.

`KUndefinedProtocol` asks the sockets server to create a socket of the specified type, using the natural choice of protocol for the selected socket type. The most common (socket type, protocol) pairs are: `KSockDatagram` with `KProtocolInetUdp` and `KSockStream` with `KProtocolInetTcp`.

Binding a socket

The `Bind()` method sets the address of a socket.

```
TInt RSocket::Bind (TSockAddr& anAddr);
```

If the socket your application creates is going to receive data, you must set its local address so that data can be routed correctly to your socket as opposed to any other socket.

Using_the_sockets_API

Taking TCP/IP sockets as our example, Bind() must be called on any datagram socket, client or server, which will receive data, and on the ?listening? stream socket.

(Note: A client stream socket which connects to a server does not need to call Bind(). The local address of a client stream socket will be set as part of the connection process.)

The Bind() method takes one parameter ? a reference to a TSocketAddr instance. In the case of TCP/IP sockets, a reference to a TInetAddr instance is passed which allows a socket address to be specified in two parts:

- the address of a networked machine
- the logical port number on that networked machine

Multiple sockets on the same machine will have the same machine address and different port numbers.

Wildcard addresses and port numbers are allowed. A common Internet address to bind a TCP/IP socket to is: (KInetAddrAny, <specific port number>). In this case, we want to set the address of our socket to our chosen port number. The use of the wildcard address KInetAddrAny indicates that we will receive data sent to our chosen port number at our machine?s address.

Note: KInetAddrAny is defined as the Internet address 0.0.0.0

Listening for connections

The method RSocket::Listen() is used to prepare a socket to listen for incoming client connection requests. It sets up a queue to hold incoming connection requests.

```
TInt RSocket::Listen (TUint qSize);
```

Despite the name, this function does NOT actually listen for the incoming connection.

When clients send a connection request to the server, each request will occupy a place in the holding queue. Should the queue become full, further connection requests will be rejected. Under normal circumstances a queue size of one is sufficient.

The method RSocket::Accept() actually waits for an incoming connection. We will look at the Accept() method next.

Note: This mechanism for processing incoming connections may be slightly different to other sockets implementations you may have used in the past. In other socket implementations, the Listen() method will wait for an incoming connection request.

Accepting connections

The `Accept()` method is used to pair a blank socket with a client connection request.

```
void RSocket::Accept (RSocket& aBlankSocket,
                    TRequestStatus& aStatus);
```

As has already been discussed, when writing a server using streams at least two sockets will be needed.

The first socket is responsible for fielding incoming client requests. It is this socket on which we call the `Listen()` and `Accept()` methods.

`Accept()` waits asynchronously for an incoming client connection request. The request will be placed in a slot in the queue created by `Listen()`, and `Accept()` will extract the request from the queue and, if all is well, pair the request with a blank socket.

We have already seen how to create a socket for a given addressing family and protocol. One of the variations of `RSocket::Open()` simply creates a blank socket ? one which is not associated with a particular protocol. Blank sockets cannot be used for data transfer until they have been paired with a client by the `Accept()` method.

It follows, therefore, that before calling the `Accept()` method, the application must create a blank socket in advance, ready for use when a client connects.

The listening socket can be re-used and can create as many connections as your server is prepared to support. In other words, you can call the `Accept()` method repeatedly on the same socket, provided there is not already an outstanding call to `Accept()`. However, each call to `Accept()` will require a new blank socket.

On completion of the `Accept()` request, the slot in the listening queue occupied by the connection request will be released. Hence, in most applications, a listening queue size of 1 is sufficient.

Connection process for datagrams

Because datagram sockets operate in an unconnected mode, there is no connection process for datagrams. All you need to do is to create and bind a socket, and you can then send and receive data.

Closing a socket

`RSocket::Close()` should be called on any socket created using `RSocket::Open()` to ensure that all resources associated with the socket are released.

Using_the_sockets_API

```
void RSocket::Close (void);
```

Additionally, if the socket is connected when Close() is called, the socket will be disconnected from the remote end point. This operation may take a prolonged time, depending on the requirements of the protocol. Note that the Close() method is synchronous.

If an asynchronous disconnection is required, RSocket::Shutdown() should be used. This method is discussed in the next section.

Note that if the parent session (RSocketServ instance) is closed, then all sockets (subsessions) associated with that session will be forcibly closed. Any further requests on such sockets will result in the application panicking.

Shutting down a socket

If you need to asynchronously shutdown a socket, rather than allow the socket to be closed synchronously, you should use RSocket::Shutdown().

```
void RSocket::Shutdown (TShutdown aHow, TRequestStatus& aStatus);
```

The first parameter to this method allows the programmer to specify the mode of disconnection. TShutdown is an enumeration, which can take one of four values: ENormal, EStopInput, EStopOutput or EImmediate. All variants will complete when the socket is disconnected.

ENormal allows the disconnection process to run to its protocol defined natural end. EImmediate does the minimum amount of work required to disconnect the socket.

Additionally, it is also possible to partially disconnect a socket. EStopInput prevents the socket receiving incoming data while EStopOutput prevents the socket transmitting any data.

Note that even using ENormal, the socket is still open ? it is still a subsession within the application?s RSocketServ session instance. If you have finished with the socket you should additionally call RSocket::Close() to free any resources associated with the socket.

Also note that there is no way to cancel a socket shutdown request once it has started.

Multi-threaded sockets applications

When writing sockets applications on other operating systems, one thread is sometimes used to create and connect sockets as required, and on connection, control of the socket is passed to another thread to perform any data transfer.

In versions of Symbian OS prior to v6.0, this type of multi-threaded design was not possible due to the design of the client-server architecture. Any client side server session or subsession was relative to the creating thread and could not be passed to another thread. This also applied to sockets, because sockets are subsessions.

As from v6.0, however, you can pass sockets between threads. Symbian OS v6.0 contains an enhanced client-server architecture which supports process-relative client resource ownership, and the sockets server has been enhanced to provide shareable session functionality. To make use of this functionality, you have to call `RSessionBase::Share()` on your sockets server session. You can then create sockets that can be used by multiple threads.

It is worth noting, however, that it is usually not necessary to implement multi-threaded applications, due to the efficient design of Symbian OS's event handling system and the use of active objects, which we will look at later in this article.

Host resolution services

What is host resolution?

In a network of computing devices, individual machines on the network sometimes use different address formats depending on who, or what, the address is presented to.

For example, your Internet email may be stored on a server that has a human readable name, such as `pop3.freemove.net`. This address, while being reasonably sensible to a person, is not much use to the network itself.

When your mail application attempts to download any waiting email, it will ask for the human readable server name to be translated to a numeric network address. Having obtained the numeric address, a connection can be established. Using the TCP/IP protocol suite, this address translation is performed using the Domain Name Service, or DNS.

This address translation serves two purposes. Firstly, it allows users of the network (the Internet in this case) to refer to network resources using a reasonably straightforward and memorable address. You may perhaps regularly use `212.134.93.203` or `204.71.202.160`. But you probably don't use these numeric addresses to access these services. You probably recognize these better as `www.symbian.com` and `www.yahoo.com` respectively.

Additionally, this separation of the address used by a user of a network resource from its physical network address allows the physical address to be changed as necessary. It also allows global service providers, such as Microsoft's Hotmail service, to provide local servers around the world, which offers users better

performance than having every single Hotmail user log on to a Seattle (or wherever) based server.

Using RHostResolver

As part of its client side API, the sockets server offers the class RHostResolver which provides a generic interface to protocol specific host resolution services. If we are talking about the TCP/IP protocol suite, then the RHostResolver class acts as an interface to the Domain Name Service, or DNS.

Each different protocol that offers a host resolution service will provide the implementation of that service as part of the relevant protocol module. The same RHostResolver client interface is used across all protocols that provide such a service.

The RHostResolver interface offers the following facilities to client applications:

- The ability to translate a numeric network address to a more human friendly textual representation
- The ability to translate a human readable address to the equivalent numeric network address
- Methods to get and set the host name of your local device

Like RSocket itself, the RHostResolver class is derived from RSubSessionBase. Therefore, in order to use the RHostResolver class, a client application must first be connected to the sockets server through the RSocketServ class.

The RHostResolver class provides a number of methods for host resolution services. Each of these methods is offered in two versions ? synchronous and asynchronous.

Note that this is a generic interface and not all protocols will support all host resolution services. Some protocols may not support any host resolution services.

If a client application attempts to call an RHostResolver method which is unsupported on a given protocol, the error value KErrNotSupported will be returned.

Before you can call any host resolution services, you need to open an instance of the RHostResolver class. As already discussed, the host resolver class is a subsession within a sockets server client session. Therefore, when an application calls RHostResolver::Open() it needs to associate the new instance with an owning server session.

```
TInt Open(RSocketServ& aSocketServer, TUint anAddrFamily,  
          TUint aProtocol);
```

Next, in the prototype shown above, we need to specify the address family our host resolver will use, in the same way that we specified an address family as a parameter to RSocket::Open().

Finally, we must specify the protocol which provides a host resolution service. If there is a natural choice within an addressing family, then KUndefinedProtocol may be specified here.

What is host resolution?

Using_the_sockets_API

Other methods offered by RHostResolver are as follows:

```
TInt GetByName(const TDesC& aName, TNameEntry& aResult);
TInt GetByAddress(const TSockAddr& anAddr, TNameEntry& aResult);
TInt GetHostName(TDes& aName);
TInt SetHostName(const TDesC& aName); // sync only
TInt Next(TNameEntry& aResult);
```

Most of the methods listed above should be quite intuitive. Next(), however, may require some explanation: for some protocols, GetByName() and GetByAddress() may find more than one answer, for example if aliases are allowed. If so, calling Next() returns the next result.

Domain Name Service (DNS)

Domain Name Service, or DNS, is the host resolution service provided by TCP/IP.

A typical DNS query involves three steps:

- A client application, running on one networked device, sends a resolution request to another networked machine ? a DNS server.
- The DNS server takes the supplied address and looks it up in a large table of addresses to translate it to a different address format.
- The DNS then returns the alternative address to the client.

Note that DNS can be used to translate from textual address formats, such as www.symbian.com, to numeric address formats, such as 212.134.93.203, or from numeric address, such as 204.71.202.160, to human friendly textual addresses, such as www.yahoo.com.

Internet service providers provide a number of DNS servers (usually more than one) for their customers to use. Without these machines, using the Internet would be a lot harder for normal users. Without DNS, we would all have to remember the 32-bit numeric addresses of our favorite web sites, and emailing people would be a lot less convenient than it is!

The important point to note here is that it is not the client device which actually performs the address translation. Another networked machine is used to do the translation for us. It should therefore come as no surprise that the numeric address of your ISP's DNS server is a key part of the configuration of an Internet connection established over TCP/IP.

Using active objects in sockets code

Networked communication systems are heavily asynchronous systems. Leaving our discussion of sockets behind for a while, consider the use of a phone.

When a friend calls our phone, it will signal the incoming call. When our phone rings, we answer it and engage in a conversation with the person at the other end.

While waiting for a call, we are free to get on with other tasks in our lives. Similarly if your friend transmits some data to you, in the form of a hard question, you may take some time to respond. While you consider your response your friend can be doing or thinking something else.

Use of a conventional telephone is a very good example of an asynchronous communication system.

When we use sockets to channel data between computers, we see a similar asynchronous model.

Examples of asynchronous events in a sockets-enabled application include the following:

- Connection, disconnection and acceptance of incoming connection requests
- Receiving data (asynchronous because we do not know when or how much data will be sent to us)
- Transmitting data (asynchronous because, at the application level, we do not know anything about the time our underlying hardware needs to physically transmit our data)
- Other, less obviously asynchronous activities, such as loading protocol modules

We need to be able to handle these asynchronous events in our applications, and in Symbian OS we do this using Active Objects (AOs).

Active Objects:

- Make life very easy for application developers
- Allow non-preemptive multitasking applications to be written using a single thread
- Therefore allow Symbian OS to offer efficient multitasking solutions without the need for multithreading

In Symbian OS, all threads are effectively event handlers, with a single active scheduler per thread cooperating with one or more active objects, responding to events from external sources.

A given active object can only handle one event source at a time. In fact, it is normal for active objects to be designed to handle one type of event exclusively.

The example code shown later in this article uses a number of active objects for different purposes. Both the client and server use at least three active objects. One is needed to handle the connection mechanics, one for receiving data and one more for transmitting data.

We will now look at an example showing the use of active objects to handle the connection of a client to a server using stream sockets.

Code example: connecting sockets

This section shows some example code illustrating the use of active objects for connecting sockets. The application from which the code has been taken comprises a ?server?, which listens for incoming connection requests, and a ?client?, which sends connection requests to the server.

Server ?listening? class definition

The code below is an extract from the full class definition for the server?s ?listening? class.

```
class CModel : public CActive {
public:
    void StartEngineL(void);
private:
    void RunL(void);
    void DoCancel (void);
private:
    RSocketServ iSession;
    RSocket     iListen, iSocket;
    CRx*       iRxAO; // Receiver active object
    CTx*       iTxAO; // Transmitter active object
};
```

Note that among the data members are two sockets, one for listening for connections, and one to handle data transfer with a connected client.

There are two other active objects included in this class, iRxAO and iTxAO. These active objects will handle received and transmitted data respectively while the client is connected to the server.

(As written, this class will only accept one client connection. Feel free to modify the class to support multiple clients!)

We will now look at how the connection process is implemented.

Getting ready to receive a client connection

The first thing our server needs to do before it can service incoming client connection requests is create two sockets, as follows:

```
// Need to use two sockets - one to listen for
```

Using_the_sockets_API

```
// an incoming connection.
err = iListen.Open(iSession, KAfInet,
                  KSockStream, KUndefinedProtocol);
User::LeaveIfError(err);
// The second (blank) socket is required to
// build the connection & transfer data.
err = iSocket.Open(iSession);
User::LeaveIfError(err);
```

One socket, `iListen`, is required to act as the "listener" to field the incoming connection requests from clients. `iListen` is associated with a stream protocol "TCP" in this case because we are using the Internet addressing family.

The other socket, `iSocket`, is the blank socket that will be paired with a successful client connection request. This socket will then be used during any subsequent data transfer with the client.

The listening socket is then free to continue listening for additional client connections, if required.

Note the use of two different variants of `RSocket::Open()`.

The first of our two sockets, `iListen`, is going to receive incoming connection requests from clients. It needs a local address so that the connection data can be routed to it.

To set the local address, we need to bind an address to the socket:

```
// Bind the listening socket to the required
// port.
TInetAddr anyAddrOnPort(KInetAddrAny, KTestPort);
iListen.Bind(anyAddrOnPort);
```

In this case, we are not worried about the network address of the socket "we are happy to use the address of the host machine. However, we do need to set the port we wish to be bound to.

Clients can then connect to our server by sending connection requests to the Internet address of our computer and the port number `KTestPort` "which is simply a #defined value in the application. Note that we must publish this port number to potential clients, otherwise they won't be able to connect to our server.

Note that because our sockets have been opened using the Internet addressing family, the `TSockAddr` subclass we pass to `Bind()` is a `TInetAddr` instance.

The `TInetAddr` class stores an IP address as a `TUint32` in its buffer after the generic data defined by `TSockAddr`. The protocol family field provided by the base class is always set to `KAfInet` (TCP/IP).

Having created the necessary sockets, and bound our listening socket, we are almost ready to start responding to any incoming client connection requests.

Using_the_sockets_API

We need to create a queue for incoming connection requests, which we do by calling `RSocket::Listen()` ? note that we are asking for a queue of length 1. When we see how the connection is handled, it will become clear that this queue length is sufficient.

```
void CModel::StartEngineL (void)
{
    ?
    // Listen for incoming connections...
    iListen.Listen(1);
    // and accept an incoming connection.
    // On connection, subsequent data transfer will
    // occur using the socket iSocket
    iListen.Accept(iSocket, iStatus);
    SetActive();
    ...
}
```

Finally, we call the asynchronous function `RSocket::Accept()` to be ready to receive client connection requests.

Recall that the `CModel` class was derived from `CActive` ? i.e., it is an active object. It follows, therefore, that when a client does connect to our server, `CModel::RunL()` should be called.

This is indeed what happens and we will look at that next.

Handling a connection request

When a client connection request is received, the outstanding `RSocket::Accept()` request is completed and the `RunL()` method of the containing active object ? in this case our `CModel` class ? is run.

```
void CModel::RunL(void)
{
    if (iStatus==KErrNone)
    {
        // Connection has been established
        NotifyEvent(EEEventConnected);
        // Now need to start the receiver AO.
        iRxAO->RxL(iSocketType);
    }
    else // error condition
        ...
}
```

Assuming all went well, our completion status will be `KErrNone`. In the code above, we pass notification of a

Using_the_sockets_API

successful connection up to our user interface and then we start the active object we are using to handle incoming data on the connected socket, iSocket.

Because this is an asynchronous system, now that we are connected, the client may start transmitting data to our socket at any time. We need to be in a position to handle incoming data as quickly as possible.

Note that we do not start the active object we reserved to handle data transmission from the connected socket. Data will only be sent to the client when the application, or user, needs to send data to the client.

Using the connected socket

Recall that our class CModel contained a member variable iRxAO of class CRx.

Class CRx is itself derived from CActive ? i.e., CRx is also an active object.

The CRx member function RxL(), shown below, issues an asynchronous request to receive data from our connected client.

```
void CRx::RxL ( ) //class CRx derived from CActive
{
    // Issue read request
    iSocket->RecvOneOrMore(iDataBuffer, 0, iStatus, iRecvLen);
    SetActive();
}
```

The function RecvOneOrMore() will be discussed shortly, along with the other functions to read and write data from/to sockets.

On completion of the incoming data request, CRx::RunL() will be called to handle the event and process the newly arrived data.

Recall that our class CModel contained a member variable iTxAO of class CTx.

Class CTx is itself derived from CActive ? i.e., CTx is also an active object.

The CTx member function TxL(), shown below, issues an asynchronous request to transmit data to our connected client.

```
void CTx::TxL (TDesC& aData)
{
    if (!IsActive())
    {
        // Take a copy of the data to be sent.
        iDataBuffer = aData;
    }
}
```

Using_the_sockets_API

```
// Issue write request
iSocket->Send(iDataBuffer, 0, iStatus);
SetActive();
}
}
```

The function Send() will be discussed shortly, along with the other functions to read and write data from/to sockets.

On completion of the data transmit request, CTx::RunL() will be called to handle the event and typically report back to the application the result of the transmission attempt.

Transferring data

We are now going to find out how we actually transfer data between two devices on a network through a socket.

As we will see, there are a number of differences between stream and datagram sockets in this area too.

Regardless of whether we are using streams or datagrams, individual data items may vary their route between two endpoints depending on numerous network conditions over which the applications at the endpoints have no control. This is particularly true, possibly even obvious for datagrams, but because datagrams underlie the implementation of a stream, it is also true for streams.

Receiving data

Using unconnected sockets

The following functions to receive incoming data through an unconnected socket are provided by RSocket.

```
void RecvFrom(TDes8& aDesc, TSockAddr& anAddr, TUint flags,
             TRequestStatus& aStatus);
void RecvFrom(TDes8& aDesc, TSockAddr& anAddr, TUint flags,
             TRequestStatus& aStatus, TSockXfrLength& aLen);
```

If your application uses unconnected sockets, the method RSocket::RecvFrom() should be used to read incoming data from a remote endpoint.

The first parameter to this method is a reference to a descriptor, into which will be written any received data.

Using_the_sockets_API

The calling application will be signaled when a full datagram has been received. The amount of data returned is given by the length of the descriptor. If the received datagram is larger than the descriptor, the incoming data will be truncated.

The second parameter will contain the address of the remote endpoint from which data was received. This address should be in a format which is valid for the protocol on which the socket was opened ? for example, a `TInetAddr` if using a TCP/IP protocol.

Note that two versions of the function are provided, both of which behave in the same way. The second prototype shown above allows the amount of data received to be found independently of the target descriptor.

Also note that a single socket may have only one receive operation pending at any given time.

The methods given above should only be used with unconnected (datagram) sockets.

Using connected sockets

The following functions to receive incoming data through a connected socket are also provided by `RSocket`.

```
void Recv(TDes8& aDesc, TUint flags, TRequestStatus& aStatus);
void Recv(TDes8& aDesc, TUint flags,
          TRequestStatus& aStatus, TSockXfrLength& aLen);
void RecvOneOrMore(TDes8& aDesc, TUint flags,
                  TRequestStatus& aStatus, TSockXfrLength& aLen);
```

If your application uses connected sockets, the methods shown above should be used to read incoming data from a remote endpoint.

Again, the first parameter to these methods is a descriptor into which the received data will be written.

The `Recv()` methods will complete when the target descriptor is full, or when the connection breaks. The amount of data read is given by the length of the descriptor ? except when the connection breaks, when no data is returned.

The second of the `Recv()` methods allows an application to determine the amount of data received through the `TSockXfrLength` reference parameter, without querying the target descriptor.

The final method, `RecvOneOrMore()`, differs from `Recv()` in that `RecvOneOrMore()` will complete as soon as any data arrives. In other words, the amount of data read by a call to `RecvOneOrMore()` will be in the range 1..n bytes, where n is the length of the target descriptor. Again, the `RecvOneOrMore()` method will also complete immediately if the connection is broken. No data will be returned in this case.

While the data flow through a connected socket may not be physically continuous, the flow is viewed as logically continuous. Applications using connected sockets must therefore take responsibility for recognizing any record boundaries in the data, if appropriate.

Using_the_sockets_API

Note that because we are using a connected socket, we do not need to specify the address from which we want to receive data. When using a connected socket you can only receive data from the remote endpoint to which you are connected.

The methods we have looked so far in this section have offered a lot of flexibility which is sometimes of no interest to application programmers.

Specifically, the methods seen so far have all had a `TUint aFlags` parameter, which has so far been ignored. The purpose of this parameter is to allow an application to select protocol specific attributes to affect the way in which the protocol handles the data receipt.

The `Read()` method shown below removes the ability to set any flags and passes the default value of 0 down to the underlying data receiving methods. Additionally, the `TSockXfrLength` parameter is removed. In this case, you can only access the amount of data received through the length of the target descriptor.

```
void Read(TDes8& aDesc, TRequestStatus& aStatus);
```

With the two exceptions described above, the behavior of the `Read()` method is identical to the `Recv()` method.

Note that this method is only suitable for use with a connected socket.

Sending data

Using unconnected sockets

The following functions to send data through an unconnected socket are provided by `RSocket`.

```
void SendTo(const TDesC8& aDesc, TSockAddr& anAddr, TUint flags,  
            TRequestStatus& aStatus);  
void SendTo(const TDesC8& aDesc, TSockAddr& anAddr, TUint flags,  
            TRequestStatus& aStatus, TSockXfrLength& aLen);
```

If your application uses unconnected sockets, the method `RSocket::SendTo()` should be used to transmit data to a remote endpoint.

The first parameter to this method is a reference to a descriptor which contains the data you wish to send to the remote endpoint. The amount of data to be transmitted is given by the length of the descriptor.

Using_the_sockets_API

The calling application will be signaled when the data has been sent. If you use the version taking a `TSockXfrLength` reference parameter, this will be filled in with the amount of data sent prior to completion.

The second parameter contains the address of the remote endpoint to which you want to send the data to. This address should be in a format which is valid for the protocol on which the socket was opened ? for example, a `TInetAddr` if using a TCP/IP protocol.

The third parameter, `TUint` flags, is a bitmask of protocol specific flags which are passed down to the appropriate protocol module.

Note that a single socket may have only one transmit operation pending at any given time.

The methods given above should only be used with unconnected (datagram) sockets.

Using connected sockets

The following functions to transmit data through a connected socket are also provided by `RSocket`.

```
void Send(const TDesC8& aDesc, TUint someFlags,
          TRequestStatus& aStatus);
void Send(const TDesC8& aDesc, TUint someFlags,
          TRequestStatus& aStatus, TSockXfrLength& aLen);
```

If your application uses connected sockets, the methods shown above should be used to transmit data to a remote endpoint.

Again, the first parameter to these methods is a descriptor containing the data you wish to send to the remote end point. The amount of data to be sent is given by the length of the descriptor.

The `Send()` methods will complete when the source descriptor has been sent, or when the connection breaks.

The second of the `Recv()` methods allows an application to determine the amount of data transmitted through the `TSockXfrLength` reference parameter, without querying the source descriptor.

Both methods feature a `TUint someFlags` parameter which can contain a bitmask of protocol specific flags passed down to the relevant protocol module.

As with the `SendTo()` method described earlier, the `TSockXfrLength` reference parameter, shown in the second method above, is filled in with the amount of data transmitted prior to completion of the asynchronous request.

Note that because we are using a connected socket, we do not need to specify the address to which we want to receive data. When using a connected socket you can only transmit data to the remote endpoint to which you are connected.

Using_the_sockets_API

The methods we have looked so far in this section have offered a lot of flexibility which is sometimes of no interest to application programmers.

The Write() method shown below removes the ability to set any flags and passes the default value of 0 down to the underlying data transmission methods. Additionally, the TSockXfrLength parameter is removed. In this case, you can only access the amount of data transmitted through the length of the source descriptor.

```
void Write(const TDesC8& aDesc, TRequestStatus& aStatus);
```

With the two exceptions described above, the behavior of the Write() method is identical to the Send() method.

Note that this method is only suitable for use with a connected socket.

Summary

This article has provided an introduction to the Symbian OS sockets server and how to use it to add communications abilities to applications.

The sockets server provides a near-standard sockets API, providing most of this API through two main classes, RSocketServ and RSocket. RSocketServ is a session to the sockets server, while RSocket is a sub-session to the sockets server. Via these two classes you can implement both unconnected and connected sockets. Host resolution services are also available via the RHostResolver class.

The design of the sockets server is based around protocol modules ? plug-in modules that implement the protocol-specific aspects of socket communication. This design will allow the sockets server to be extended to support future protocols. Protocols supported by v6.0 include TCP/IP, IrDA, SMS and Bluetooth® wireless technology.

Courtesy: This article had been taken from www.symbian.com articles section, originally published by **Gavin Meiklejohn** on 2005. Presently the link is not available on the net so thought of sharing this valuable article here.