Symbian OS Internals/04. Inter-thread Communication

by Andrew Rogers and Jane Sales

Be not deceived: evil communications corrupt good manners.
1 Corinthians 15:33

In the last chapter, I introduced Symbian OS threads. Now I will go on to discuss some of the mechanisms that those threads can use to communicate with one another. Symbian OS provides several such mechanisms, including shared I/O buffers, publish and subscribe, message queues and client-server. Each of the methods has its particular merits and restrictions. I will examine each of them in turn, discussing their implementation and the general class of problem each is intended to solve. Since the most widely used of these is the client-server mechanism used to communicate with system servers, I will start there.

Client-server ITC

Client-server is the original Symbian OS inter-thread communication (ITC) mechanism, having been present from the earliest implementations of EPOC32 in the Psion Series 5 right up to the latest mobile phones deployed on the secure platform of Symbian OS v9.

This mechanism allows a client to connect to a server using a unique, global name, establishing a *session* that provides the context for all further requests. Multiple sessions can be established to a server, both from within a single thread and from within multiple threads and processes. Clients queue messages in the kernel; the server then retrieves these messages and processes each in turn. When the processing of a request is finished, the server signals the client that its request is complete. The implementation of the client-server mechanism guarantees request completion, even under out-of-memory and server death conditions. The combination of a robust service request mechanism and a service provider guaranteed to be unique within the system was designed to make the client-server mechanism suitable for providing centralized and controlled access to system resources.

The managing of central resources using the client-server paradigm can be seen throughout Symbian OS, with all the major services provided by Symbian OS being implemented using this method. The file server (F32), the window server (WSERV), the telephony server (ETEL), the comms server (C32) and the socket server (ESOCK) are all examples of central resources managed by a server and accessed by a client interface to them.

However, my intention in this chapter is to give you the inner workings of client-server. It is not to teach you how to write your own server based on this framework, or to show you how to use any of the existing servers within Symbian OS. For that, please refer to a Symbian OS SDK or Symbian OS Explained (*Symbian OS Explained: Effective C++ Programming for Smartphones*, by Jo Stichbury. Symbian Press). Since this book focuses on EKA2, I will only discuss the newest version of client-server (IPCv2). I will summarize the differences from the legacy client-server framework (IPCv1) in a separate section at the end.

History

The design and architecture of the client-server system have evolved with the platform. In the original, pre-Symbian OS v6.0 implementation of client-server, a session was explicitly tied to the thread that had created it. In Symbian OS v6.0, we generalized the concept of a session from this very *client-centric* implementation by adding the concept of a *shared session* that could be used by all the threads within the process.

Though they appeared less client-centric in the user API, we implemented shared sessions within the kernel by creating objects tied to the session to represent each "share". The share managed per-thread resources for the session, including a handle to the client thread for the server. The handle from the share that corresponded to the thread that sent a given message was then passed in the message delivered to the server, so that when the server performed an operation to access a client's address space using a message handle, it was actually performed in the kernel using a handle associated with the client (in the share) rather than the message itself.

With the advent of a secure platform on EKA2, Symbian OS client-server has undergone a second evolution to an entirely *message-centric* architecture known as IPCv2. IPCv2 performs all accesses to memory in a client's address space using a handle that is explicitly associated with a message, rather than the client that sent it. This allows us to deprecate and then remove APIs that allow direct access to an arbitrary thread's address space with nothing more than a handle to that thread.

Earlier versions of EKA2 supported the legacy client-server APIs (IPCv1), using a special category of handle values, which directly referred to kernel-side message objects. The kernel passed these back as the client thread handle in legacy messages. When the kernel detected the use of such a handle by legacy code during translation of a user-side handle to a kernel-side object,

it located the corresponding kernel-side message object, extracted the client thread associated with that message, then used it to perform the requested operation. This legacy support was removed with the move to the secure Symbian OS v9.

Also, as part of the changes we made to implement a secure Symbian OS, we chose several categories of kernel-side objects to be able to be shared securely and anonymously between two separate processes. Handles to such *protected* objects may be passed between processes both by an extended *command line* API and via the client-server mechanism. We have extended the *command line* API to allow a parent to pass handles to its children, and we have extended the client-server mechanism to enable a message to be completed with a handle rather than a standard error code.

These *protected* objects can provide a mechanism for secure data transfer between processes. In particular, this new class of objects includes server sessions, so that a session to a server can be shared between two separate processes. This means that in EKA2 a new method of sharing a session has now become possible: global (inter-process) session sharing. An interesting range of features can be designed using this new paradigm. For example, secure sharing of protected files between processes can be accomplished using a shared file server session.

Design challenges

The design of the kernel-side client-server architecture that I will describe in more detail below is inherently a difficult matter. Even in the most basic of configurations, with only a single client thread communicating with a single-threaded server, there are several things that can happen concurrently:

- A client may send a message (asynchronously)
- A client may close a session
- Either the client or the server thread may die, resulting in kernel-side object cleanup being invoked
- A server thread may complete a message (in a multi-threaded server, possibly a different thread than the one owning the handle to the kernel-side server object)
- The server may close the handle to the kernel-side server object.

All of these actions require the state of kernel-side objects to be updated and, since these actions may execute concurrently and preempt one another, they need to be carefully synchronized to avoid corruption of the kernel data structures. To make this task as simple and efficient as possible, it was important that our design minimized the complexity of these objects, their states and the transitions needed to perform client-server communication.

Design simplicity was a key priority on EKA2. This was because, if the required synchronization (using the system lock) is not performed in a constant-order manner, unbounded delays can occur and there is no possibility of Symbian OS offering hard real-time guarantees. Also, care needs to be taken to allow the server itself to respond to messages it has received within bounded time limits, so that servers which offer real-time guarantees themselves can be created. To this end, we added support in IPCv2 for asynchronous session creation and destruction, allowing the server's `RunL()` duration to be small and bounded. In this way we have provided a platform on which, for example, guaranteed multimedia services can be offered.

However, the design of client-server on EKA2 is further complicated due to the need to provide a secure IPC architecture. This has meant adding checks that operations performed on a client's address space are valid and providing a mechanism whereby the identity of a server may be securely verified. It has also meant adding the ability for sessions to be shared so that requirements for a secure platform such as secure file sharing can be implemented using client-server.

I'll now describe both the new user-side architecture of IPCv2 and the kernel architecture that supports it.

User-side architecture - server

The key component in the user-side architecture is the server object itself. This is simply a standard active object (For a comprehensive explanation of the use of active objects in Symbian OS C++, consult *Symbian OS Explained* by Jo Stichbury, Symbian Press), queuing requests on an asynchronous service API provided by the kernel to retrieve messages sent to the server by its clients. On the completion of such a request (receiving a message), the server active object's RunL() usually dispatches the message to a session object which has been created to manage the connection between the server and its client.

Session objects manage the server-side resources associated with a session and are created by the server when a connect message is received from the client. In this case, the server's `RunL()` calls the virtual `NewSessionL()` which returns an instance of a `CSession2` derived class. You would implement the server active object by deriving from `CServer2` and the session object by deriving from `CSession2`. The server-side architecture is shown in Figure 4.1 and `CServer2` is implemented as follows:

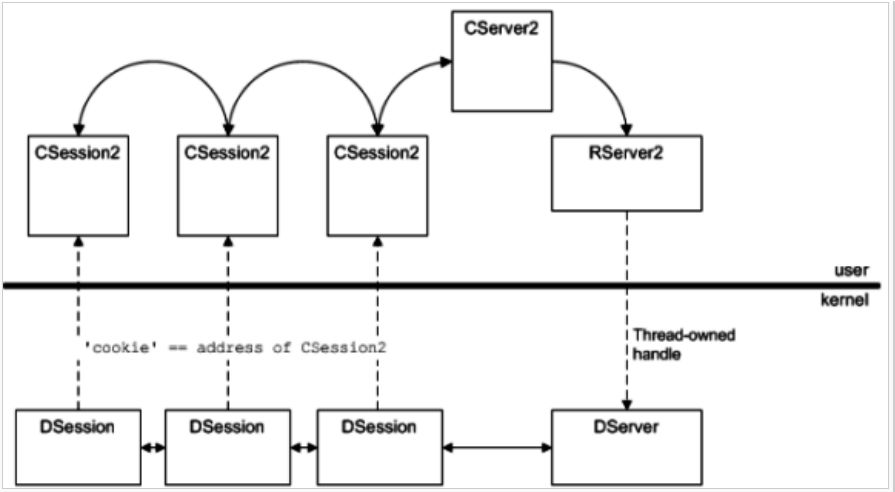


Figure 4.1 *Server-side architecture*

```
class CServer2 : public CActive
  {
public:
  enum TServerType
  {
    EUnsharableSessions = EIpcSession_Unsharable,
    ESharableSessions = EIpcSession_Sharable,
    EGlobalSharableSessions = EIpcSession_GlobalSharable,
  };

public:
  IMPORT_C virtual ~CServer2() =0;
  IMPORT_C TInt Start(const TDesC& aName);
  IMPORT_C void StartL(const TDesC& aName);
  IMPORT_C void ReStart();
  inline RServer2 Server() const { return iServer; }

protected:
  inline const RMessage2& Message() const;
  IMPORT_C CServer2(TInt aPriority, TServerType aType=EUnsharableSessions);
  IMPORT_C void DoCancel();
  IMPORT_C void RunL();
  IMPORT_C TInt RunError(TInt aError);
  IMPORT_C virtual void DoConnect(const RMessage2& aMessage);

private:
  IMPORT_C virtual CSession2* NewSessionL(const TVersion& aVersion, const RMessage2&
aMessage) const =0;

private:
  TInt iSessionType;
  RServer2 iServer;
  RMessage2 iMessage;
  TDblQue<CSession2> iSessionQ;

protected:
  TDblQueIter<CSession2> iSessionIter;
  };
```

The server keeps track of the current message it is processing (iMessage) and manages a queue of the session objects so they

can be cleaned up during the destruction of the server (iSessionQ).

Upon reception of a (session) connect message, the server calls `NewSessionL()` to create a new `CSession2`-derived session object and then appends the new session object to the end of the server queue. Finally, the server uses the connect message to pass a *cookie* (in the form of the address of the newly-created session object) to the kernel, which the kernel can then use to identify the particular session a given message is associated with.

The server implementation can over-ride `DoConnect()` in order to provide asynchronous session creation. It works in a similar way to the method I described in the previous paragraph, except that the memory allocation for the new session is performed in a separate thread and the connect message is completed asynchronously, rather than after a (potentially long) synchronous wait in `DoConnect()`. The separate thread will simply consist of a base call to `DoConnect()` as there is no public API to perform the actions of DoConnect() separately, such as setting the session's "cookie".

Your `CServer2`-derived server active object uses an instance of the private class `RServer2` - which holds a handle to the equivalent kernel-side server object, `DServer` - to receive messages from the kernel. The handle to the kernel-side kernel object, contained in the `iServer` member, remains open throughout the lifetime of the server object and is Closed in its destructor.

The server active object receives messages from the kernel by queuing a request on the `RServer2`. This is done automatically for you in `Start()` after the kernel-side server object has been successfully created. A request is also automatically requeued after processing a message in the server's `RunL()`. The request to receive another message from the kernel is canceled in the server object's DoCancel() and the server active object cancels itself in its destructor, as with any other active object.

The kernel-side (DServer) object is created via a call to `Start()` on your CServer2-derived object and the level of session sharability the server will support, as enumerated in `CServer2::TServerType`, is determined at this time. You specify the level of sharability a given session actually has when a client calls `RSessionBase::CreateSession()`. The sharability of a new session is opaque to the server itself, which will only know that a new session has been created. Therefore it is the kernel's responsibility to police sharability - both when creating new sessions and also when opening handles to existing ones.

CSession2, from which you derive session objects used to manage per-session resources in your server, is shown below:

```
class CSession2 : public CBase
  {
  friend class CServer2;
public:
  IMPORT_C virtual ~CSession2() =0;

private:
  IMPORT_C virtual void CreateL(); // Default method, does nothing

public:
  inline const CServer2* Server() const;
  IMPORT_C void ResourceCountMarkStart();
  IMPORT_C void ResourceCountMarkEnd(const RMessage2& aMessage);
  IMPORT_C virtual TInt CountResources();
  virtual void ServiceL(const RMessage2& aMessage) =0;
  IMPORT_C virtual void ServiceError(const RMessage2& aMessage, TInt aError);

protected:
  IMPORT_C CSession2();
  IMPORT_C virtual void Disconnect(const RMessage2& aMessage);

private:
  TInt iResourceCountMark;
  TDblQueLink iLink;
  const CServer2* iServer;
  };
```

The heart of the session object is the `ServiceL()` method. On reception of a message, the server uses the cookie the kernel returns to it as a pointer to a session object and then passes the message to that session by calling `ServiceL()`. The session

class will then perform appropriate actions to fulfil the request contained in the message and at some future point the message will be completed to signal the completion of the request to the client.

The virtual method Disconnect() is used to allow the session to implement asynchronous session deletion in a similar manner to that described for asynchronous session creation using `CServer2::DoConnect()`.

The server can access the kernel-side message objects using `RMessagePtr2`, which encapsulates both the handle to the message object and descriptor APIs for accessing the client's memory space using the message. A small number of APIs to allow manipulation of the client thread are available to allow the server to enforce certain behavior of its client, for example, enforcing that the client passes certain parameters by panicking it if it presents a message containing invalid values. Finally, `RMessagePtr2` also presents APIs that allow the security attributes of the client to be examined and checked against predetermined security policies. The security of these APIs is ensured as the kernel verifies that the thread using them has a valid handle to a kernel message object.

The client constructs a message for a particular session by specifying a *function number* to identify the operation that is being requested and optionally up to four message parameters. These parameters may be either plain integers, pointers or may be descriptors that the server will then be able to use to access the client's address space. Using templated argument types and an overloaded function that maps argument types to a bitfield, the class `TIpcArgs` (which is used to marshallmessage arguments in IPCv2) generates a bit mask at compile time which describes the types of its arguments.

This bit mask is stored in the kernel-side message object. This allows the kernel to check whether operations requested by the server using the message are correct - for example, checking the source and target descriptors are either both 8-bit or both 16-bit descriptors. It also allows the kernel to check that the requested operation is permitted by the client, for example by checking that when the server requests to write to a client descriptor, the client descriptor is `TDes`-derived (modifiable) rather than `TDesC`-derived (constant).

You should beware that though you can still pass pointers in IPCv2, there are no longer any APIs to directly access memory in another thread's address space using an arbitrary pointer and a handle to the thread as this is inherently insecure. The ability to pass a pointer between a client and server is therefore only of any value when the client and server are within the same process. In this case, the use of a pointer is obviously not limited to pointing to a descriptor, but may also be used to point to an arbitrary data structure containing information to be shared between the client and server:

```
class RMessagePtr2
  {
public:
  inline RMessagePtr2();
  inline TBool IsNull();
  inline TInt Handle();
  #ifndef __KERNEL_MODE__
    inline TBool ClientDataCaging();
    IMPORT_C void Complete(TInt aReason);
    IMPORT_C void Complete(RHandleBase aHandle);
    IMPORT_C TInt GetDesLength(TInt aParam);
    IMPORT_C TInt GetDesLengthL(TInt aParam);
    IMPORT_C TInt GetDesMaxLength(TInt aParam);
    IMPORT_C TInt GetDesMaxLengthL(TInt aParam);
    IMPORT_C void ReadL(TInt aParam,TDes8& aDes,TInt aOffset=0);
    IMPORT_C void ReadL(TInt aParam,TDes16 &aDes,TInt aOffset=0);
    IMPORT_C void WriteL(TInt aParam,const TDesC8& aDes,TInt aOffset=0);
    IMPORT_C void WriteL(TInt aParam,const TDesC16& aDes,TInt aOffset=0);
    IMPORT_C TInt Read(TInt aParam,TDes8& aDes,TInt aOffset=0);
    IMPORT_C TInt Read(TInt aParam,TDes16 &aDes,TInt aOffset=0);
    IMPORT_C TInt Write(TInt aParam,const TDesC8& aDes,TInt aOffset=0);
    IMPORT_C TInt Write(TInt aParam,const TDesC16& aDes,TInt aOffset=0);
    IMPORT_C void Panic(const TDesC& aCategory,TInt aReason);
    IMPORT_C void Kill(TInt aReason);
    IMPORT_C void Terminate(TInt aReason);
    IMPORT_C TInt SetProcessPriority(TProcessPriority aPriority);
    inline void SetProcessPriorityL(TProcessPriority aPriority);
```

```
    IMPORT_C TInt Client(RThread& aClient, TOwnerType aOwnerType=EOwnerProcess);
    inline void ClientL(RThread& aClient, TOwnerType aOwnerType=EOwnerProcess);
    IMPORT_C TUint ClientProcessFlags();
    IMPORT_C TSecureId SecureId();
    IMPORT_C TVendorId VendorId();
    inline TBool HasCapability(TCapability aCapability, const char* aDiagnostic=0);
    inline void HasCapabilityL(TCapability aCapability,
    const char* aDiagnosticMessage=0);
    inline TBool HasCapability(TCapability aCapability1, TCapability aCapability2, const
char* aDiagnostic=0);
    inline void HasCapabilityL(TCapability aCapability1, TCapability aCapability2, const
char* aDiagnosticMessage=0);
    inline TUid Identity() const { return SecureId(); }
  #endif

protected:
  TInt iHandle;
  };

inline TBool operator==(RMessagePtr2 aLeft, RMessagePtr2 aRight);
inline TBool operator!=(RMessagePtr2 aLeft, RMessagePtr2 aRight);
```

RMessage2 expands the API provided by `RMessagePtr2` by bringing a copy of the message arguments and the cookie (CSession2 pointer) stored in the kernel over into user space, allowing access to both. `RMessage2` also contains a number of spare words that do not correspond to data stored in the kernel. One of these, `iFlags`, is used to provide the *Authorised()* APIs which are used by a utility server base class, `CPolicyServer`, which allows simple implementation of a server that validates given security policies upon session creation and reception of each message. For full details of `CPolicyServer` and its use, refer to a recent Symbian OS SDK.

```
class RMessage2 : public RMessagePtr2
  {
  friend class CServer2;

public:
  enum TsessionMessages
    {
    EConnect=-1,
    EDisConnect=-2
    };

public:
  inline RMessage2();
  #ifndef __KERNEL_MODE__
    IMPORT_C explicit RMessage2(const RMessagePtr2& aPtr);
    void SetAuthorised() const;
    void ClearAuthorised() const;
    TBool Authorised() const;
  #endif
  inline TInt Function() const;
  inline TInt Int0() const;
  inline TInt Int1() const;
  inline TInt Int2() const;
  inline TInt Int3() const;
  inline const TAny* Ptr0() const;
  inline const TAny* Ptr1() const;
```

```
  inline const TAny* Ptr2() const;
  inline const TAny* Ptr3() const;
  inline CSession2* Session() const;

protected:
  TInt iFunction;
  TInt iArgs[KMaxMessageArguments];

private:
  TInt iSpare1;

protected:
  const TAny* iSessionPtr;

private:
  mutable TInt iFlags; // Currently only used for *Authorised above
  TInt iSpare3; // Reserved for future use
  friend class RMessage;
  };
```

User-side architecture - client

The client-side architecture is shown in Figure 4.2.

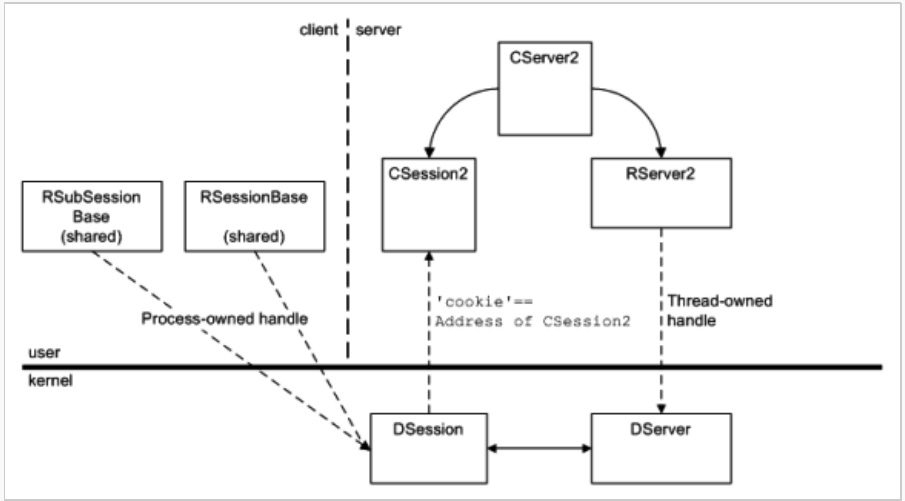


Figure 4.2 *Client-side architecture*

The remainder of the user-side support for client-server consists of two client base classes, one for sessions and another for sub-sessions. The session class is `RSessionBase`, as follows:

```
class RSessionBase : public RHandleBase
  {
  friend class RSubSessionBase;

public:
  enum TAttachMode
    {
    EExplicitAttach,
    EAutoAttach
    };

public:
  inline TInt ShareAuto() { return DoShare(EAutoAttach); }
  inline TInt ShareProtected()
```

```
  { return DoShare(EAutoAttach|KCreateProtectedObject); }
  IMPORT_C TInt Open(RMessagePtr2 aMessage,TInt aParam, TOwnerType aType=EOwnerProcess);
  IMPORT_C TInt Open(RMessagePtr2 aMessage,TInt aParam,
  const TSecurityPolicy& aServerPolicy, TOwnerType aType=EOwnerProcess);
  IMPORT_C TInt Open(TInt aArgumentIndex, TOwnerType aType=EOwnerProcess);

protected:
  IMPORT_C TInt Open(TInt aArgumentIndex, const TSecurityPolicy& aServerPolicy,
TOwnerType aType=EOwnerProcess);
  inline TInt CreateSession(const TDesC& aServer, const TVersion& aVersion);
  IMPORT_C TInt CreateSession(const TDesC& aServer, const TVersion& aVersion,TInt
aAsyncMessageSlots);
  IMPORT_C TInt CreateSession(const TDesC& aServer, const TVersion& aVersion,
    TInt aAsyncMessageSlots, TIpcSessionType aType, const TSecurityPolicy* aPolicy=0,
    TRequestStatus* aStatus=0);
  inline TInt CreateSession(RServer2 aServer,const TVersion& aVersion);
  IMPORT_C TInt CreateSession(RServer2 aServer, const TVersion& aVersion, TInt
aAsyncMessageSlots);
  IMPORT_C TInt CreateSession(RServer2 aServer, const TVersion& aVersion, TInt
aAsyncMessageSlots,
    TIpcSessionType aType, const TSecurityPolicy* aPolicy=0, TRequestStatus* aStatus=0);
  inline TInt Send(TInt aFunction,const TIpcArgs& aArgs) const;
  inline void SendReceive(TInt aFunction,const TIpcArgs& aArgs, TRequestStatus& aStatus)
const;
  inline TInt SendReceive(TInt aFunction,const TIpcArgs& aArgs) const;
  inline TInt Send(TInt aFunction) const;
  inline void SendReceive(TInt aFunction,TRequestStatus& aStatus) const;
  inline TInt SendReceive(TInt aFunction) const;

private:
  TInt SendAsync(TInt aFunction,const TIpcArgs* aArgs, TRequestStatus* aStatus) const;
  TInt SendSync(TInt aFunction,const TIpcArgs* aArgs) const;
  IMPORT_C TInt DoShare(TInt aAttachMode);
  TInt DoConnect(const TVersion &aVersion,TRequestStatus* aStatus);
};
```

Sub-sessions are simply a lightweight wrapper over the functionality of session objects, as already described above. They are useful because it is often the case that clients wish to use multiple instances of an API that would otherwise be associated with the session, for example a client might wish to have multiple instances of the file API from the file server. Sessions are relatively heavyweight in terms of the kernel overhead associated with them, so rather than insist that a new session be created to support such paradigms, we provide a simple mechanism for multiplexing multiple *sub-sessions* over a single session in the `RSubSessionBase` class.

You enable sub-session creation by specifying a specific function that the server will use to create resources required to manage the sub-session. As well as this, the sub-session creation function generates a *sub-session cookie* that `RSubSessionBase` then stores, which is automatically passed as the fourth argument of any future messages to that session. (This leaves only three parameters for use by the sub-session requests.) When a session object receives a request that it recognizes as being for a sub-session, it uses the cookie in the fourth argument to identify the sub-session and then processes the message accordingly. You should note that the sub-session cookie is only shared by the client and server and is opaque to the kernel, which sees it as any other message parameter. For example, requests to the `RFile` API appear to the kernel as identical to requests to the file server session API, `RFs`, of which it is a sub-session.

From the following declaration of `RSubSessionBase`, it can be seen that it is simply a wrapper around the `RSessionBase` API implemented using a private `RSessionBase` member and a copy of the sub-session cookie used to identify the sub-session to the server:

```
class RSubSessionBase
```

```
  {
public:
  inline TInt SubSessionHandle() const;

protected:
  inline RSubSessionBase();
  IMPORT_C const RSessionBase Session() const;
  inline TInt CreateSubSession(const RSessionBase& aSession, TInt aFunction,const
TIpcArgs& aArgs);
  inline TInt CreateSubSession(const RSessionBase& aSession, TInt aFunction);
  IMPORT_C TInt CreateAutoCloseSubSession(RSessionBase& aSession, TInt aFunction,const
TIpcArgs& aArgs);
  IMPORT_C void CloseSubSession(TInt aFunction);
  inline TInt Send(TInt aFunction,const TIpcArgs& aArgs) const;
  inline void SendReceive(TInt aFunction,const TIpcArgs& aArgs, TRequestStatus& aStatus)
const;
  inline TInt SendReceive(TInt aFunction,const TIpcArgs& aArgs) const;
  inline TInt Send(TInt aFunction) const;
  inline void SendReceive(TInt aFunction,TRequestStatus& aStatus) const;
  inline TInt SendReceive(TInt aFunction) const;

private:
  RSessionBase iSession;
  TInt iSubSessionHandle;
  };
```

Note that because a thread blocks on the synchronous API, only one synchronous server message may be sent by a thread at a time. This allows a significant optimization in the allocation of kernel-side memory used to hold messages.

The session and sub-session creation functions have new overloads in IPCv2 that allow the session to be created asynchronously, so the server cannot maliciously block the client whilst connecting to it.

The other methods of note in `RSessionBase` are the ShareXxx() methods. Previously, in EKA1, a session could only be created in a non-shared state. If sharing was required, then the client had to explicitly call `Share()` (not present in IPCv2) or `ShareAuto()` to create a process-relative session handle. A similar method has been added in EKA2 to explicitly create a process-relative handle that is also *protected* and may be passed to another process - `ShareProtected()`.

However, creating a duplicate handle is an expensive operation, so we have now provided a new overload of `RSessionBase::Create-Session()` which allows the sharability of the session to be determined from the creation of the session, thereby avoiding the need to perform the expensive handle duplication operation. This then is the preferred and recommended way of creating a shared session on EKA2. Before a client in a separate thread or process can use a given session, the session must either be created with the required level of sharability or a call to a `ShareXxx()` method must be made before the separate thread or process sends any messages to the session, or a panic will occur in the client.

If the return value from a client call to `RSessionBase::Create-Session()` indicates that the server is not running, then the client DLL may wish to launch the server process itself and then make another attempt to create a session with the server. When doing this, the client code needs to be able to detect whether and when the server has started up successfully. To do this in the past, the client had to pass a structure containing its thread ID and a pointer to a request status object in its address space to the server via the server's command line. The server would then use this structure to signal successful startup to the client using `RThread::RequestComplete()`. The introduction of thread and process rendezvous, as described in Section 3.3.6, removes the need for this mechanism and simplifies server startup code considerably.

Kernel-side architecture

Server - queue management

The kernel-side architecture reflects the structure of the user-side client-server architecture it is designed to support. The first kernel-side object I'll discuss is the server object, corresponding to the `RServer2` handle held by the server process. The kernel-side server object has two purposes:

- To ensure the uniqueness of the user-mode server within the system
- To provide a FIFO queue of messages to be delivered to the server thread. A client may deliver messages to the server at any time, but the server thread only receives the messages one at a time as it sequentially requests them from the kernel.

The first requirement is easy to achieve: during server creation the kernel adds the server object being added to a global object container for servers. As I will show in the next chapter, object containers mandate the uniqueness of names within them. Also, another check is performed at server creation: the server may only specify a name beginning with *!* if the server has the ProtServ capability. This allows the client to be certain that servers with names beginning with *!* have not been spoofed by some other malicious code.

To fulfill the second requirement, server objects use the state machine shown in Figure 4.3 to manage their FIFO queue of messages.

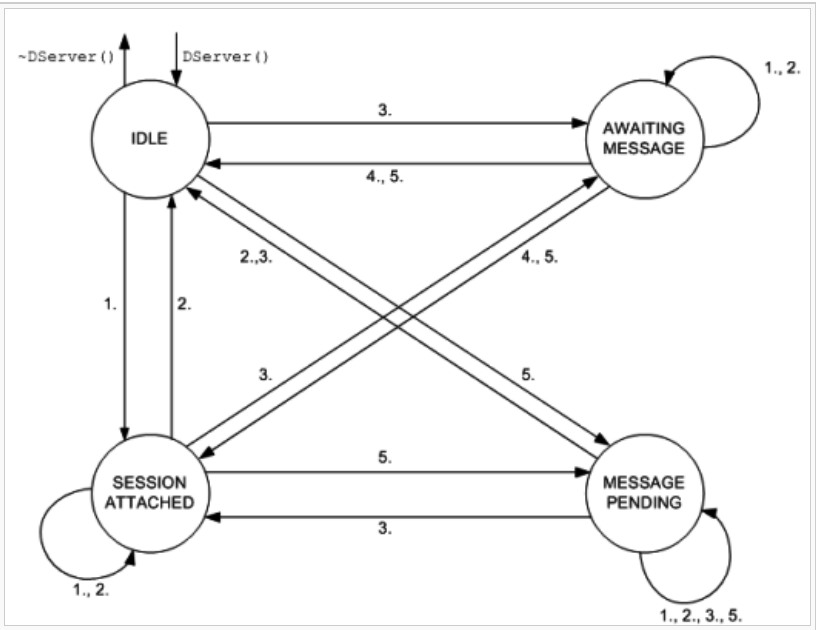


Figure 4.3 *Server object state machine*

The labeled state transitions listed below are executed under the protection of the system lock, to maintain state integrity. Each of these transitions is designed to hold the system lock for constant execution time. This maintains the real-time characteristics of the kernel as there will always be a maximum, constant time for which the system lock will be held before being released:

1. `DSession::Add()`
2. `DSession::Detach()`
3. `DServer::Receive()`
4. `DServer::Cancel()`, `DServer::Close()` [when closing last reference]
5. `DServer::Deliver()`

The states in the above transition diagram are defined as follows:

	Session queue	Message queue	[User] Server request status
IDLE	Empty	Empty	NULL
SESSION ATTACHED	Non-empty	Empty	NULL
AWAITING MESSAGE	Don't care	Empty	Non-NULL
MESSAGE PENDING	Don't care	Non-empty	NULL

The implementation of this state machine can be seen in the methods and members of the `DServer` class:

```
class DServer : public DObject
  {
public:
  DServer();
  virtual ~DServer();
  virtual TInt Close(TAny*);
  virtual TInt RequestUserHandle(DThread* aThread, TOwnerType aType);
  // aMessage bit 0 = 0 -> RMessage, bit 0 = 1 -> RMessage2
  void Receive(TRequestStatus& aStatus, TAny* aMessage);
```

```
  void Cancel();
  void Accept(RMessageK* aMsg);
  void Deliver(RMessageK* aMsg);

public:
  inline TBool IsClosing();

public:
  DThread* iOwningThread; // thread which receives messages
  TAny* iMsgDest; // where to deliver messages
  TRequestStatus* iStatus; // completed to signal message arrival
  SDblQue iSessionQ; // list of sessions
  SDblQue iDeliveredQ; // messages delivered but not yet accepted
  TUint8 iSessionType; // TIpcSessionType
  };
```

The server object itself is created by `ExecHandler::ServerCreate()`, called (indirectly via `RServer2)` from `CServer2::Start()`. Once `ExecHandler::ServerCreate()` has created the kernel-side server object, it opens a handle on the current (server) thread, so the pointer to the server thread (iOwningThread) is always valid.

This is required because a process-relative handle may be created to the server object by any thread in the server's process, using `Duplicate()` and hence the server objects may be held open after the server thread terminates. The first handle to the server object that is created and subsequently vested within the `RServer2` will be a process-relative handle if the server is anonymous (that is, has a zero-length name) and thread-relative otherwise. To exercise control over the use of `Duplicate()`, `DServer` over-rides DObject's `RequestUserHandle()` method, which is called whenever a user-mode thread wishes to create a handle to an object. `DServer` enforces the policy that handles to it may only be created within the server's process. The server object then closes the reference to the server thread in its destructor, so the thread object will only ever be safely destroyed after the server object has finished using it.

`DServer::Receive()` and `DServer::Cancel()` provide the kernel-side implementation of the private `RServer2API` used to retrieve messages for the server. These receive and cancel functions provide an API for an asynchronous request to de-queue the message at the head of the FIFO queue. After de-queuing a message, the request is completed on the server thread. If a message is present in the server's queue when the request is made, this operation is performed immediately. Otherwise no action is taken and the next message delivered to the DServer object is used to immediately complete this request, rather than being placed on the server's queue.

The server thread may choose to block until a message is available (for example, using `User::WaitForRequest()`) or may use another method to wait for the request completion. In the case of a standard Symbian OS server, the `CServer2`-derived class is an active object and uses the active scheduler as a mechanism to wait for the request for the next message to be completed.

The procedure of writing a message to the server process's address space and signaling it to notify it of the completion of its request for a message is known as *accepting* a message. `DServer::Deliver()` is the client thread's API to deliver a message to the server's message queue. Both `DServer::Receive()` and `DServer::Deliver()` will accept a message immediately, where appropriate. These methods both call a common subroutine `DServer::Accept()`, which contains the code to accept a message. It updates the message's state to reflect the fact the server has accepted its delivery before writing the message to the server's address space and finally signaling the server's request status to indicate completion of its request.

The kernel-side message object (RMessageK) is converted into the correct format for user-side message object by using a utility classes whose structure mirrors `RMessage2`:

```
class RMessageU2
  {
public:
  inline RMessageU2(const RMessageK& a);

public:
  TInt iHandle;
  TInt iFunction;
```

```
TInt iArgs[KMaxMessageArguments];
TUint32 iSpare1;
const TAny* iSessionPtr;
};
```

Note that the `iSpare1member` of `RMessageU2` is simply zeroed by the kernel when writing the message to the user (that is, the server thread), but the other *unused* members of `RMessage2` will not be overwritten by the kernel when it writes a message to user-space. A separate structure is used here as the format of `RMessageK` is private to the kernel itself and this class therefore provides translation between the internal message format used by the kernel and the public message format of `RMessage2` used by user-side code.

Most of the methods mentioned above can be written to hold the system lock for a constant time with relative ease as they encompass tasks such as adding to or removing from a doubly linked list, updating state and performing a fast write of a small amount of data. However, when the `DServer` object is closed for the last time in `DServer::Close()`, the session list has to be iterated to detach all the sessions still attached to the server. This must be done under the protection of the system lock so that the server's state is updated in a consistent manner. As there are an arbitrary number of sessions to detach, this operation has an execution time linearly proportional to the number of sessions, as opposed to a constant execution time.

This operation is therefore carefully split up into n separate operations, each of which only hold the system lock for a constant time and each of which leave the data structures in a consistent state. `DServer::Close()` acquires the system lock before detaching each session by calling `DSession::Detach()`, which will release the lock before returning. `DSession::Detach()` is an operation made up of freeing an arbitrary number of uncompleted messages that have been sent by that session to the server. Again, this operation is split up by acquiring the system lock before freeing each message and then releasing it again afterwards, so the system lock is never held for more than a constant, bounded time whilst one message is freed or one session is removed from the server's queue.

To achieve the consistency required whilst splitting up these operations, the fact that a server or session is closing (iAccessCount is 0) is used to restrict what actions may occur. For example, new sessions cannot be attached to a server whilst it is closing and messages cannot be completed whilst a session is closing.

Sessions - delivery and message pool management

In the previous section, I described how sessions provide the context for communication between the client and server. Specifically, the kernel session objects manage the delivery of messages to the server and ensure message completion, even under out-of-memory conditions. They also manage user-mode access to a session, as specified by the session's *sharability*.

To ensure message completion, the session object maintains a queue of message objects distinct from that of the server. This queue also includes messages sent by the session that have not yet been completed by the server. The interaction of this queue with both the lifetime of the client and the server is controlled via the state machine shown in Figure 4.4.

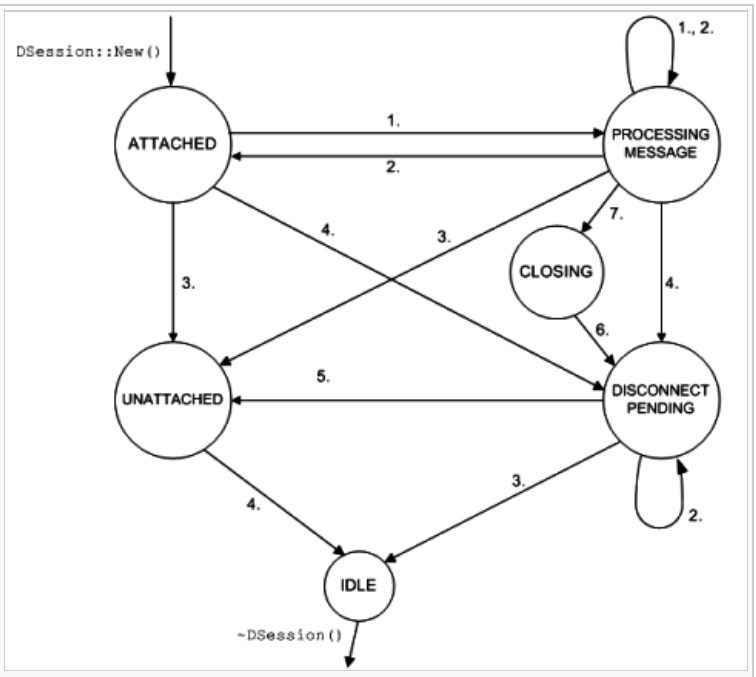


Figure 4.4 *Session object state machine*

Again, the labeled state transitions listed below are executed under the protection of the system lock to maintain state integrity. These transitions are also designed to hold the system lock for a constant execution time, in order to maintain the real-time characteristics of the kernel:

1. `DSession::Send()`
2. `ExecHandler::MessageComplete()`
3. `DSession::Detach()`
4. `DSession::Close()` [when closing last reference], [no connect message pending or not accepted by server]
5. `DSession::CloseFromDisconnect()`
6. `ExecHandler::SetSessionPtr()`
7. `DSession::Close()` [when closing last reference], [connect message pending and accepted by server]

The states in Figure 4.4 are defined like this:

	Server queue (of sessions)	**Client references**	**Message queue**
ATTACHED	Queued	Open	Empty
PROCESSING MESSAGE	Queued	Open	Non-empty
CLOSING	Queued	Closed	Non-empty, contains connect msg.
DISCONNECT PENDING	Queued	Closed	Non-empty
UNATTACHED	De-queued	Open	Empty
IDLE	De-queued	Closed	Empty

The implementation of `DSession` to support this is as follows:

```
class DSession : public DObject
  {
public:
  DSession();
  virtual ~DSession();
  virtual TInt Close(TAny*);
  virtual TInt RequestUserHandle(DThread* aThread, TOwnerType aType);
  void Detach(TInt aReason);
  RMessageK* GetNextFreeMessage();
  RMessageK* ExpandGlobalPool();
  void CloseFromDisconnect();
  static TInt New(DSession*& aS, TInt aMsgSlots, TInt aMode);
  TInt Add(DServer* aSvr, const TSecurityPolicy* aSecurityPolicy);
  TInt MakeHandle();
  TInt Send(TInt aFunction, const TInt* aPtr, TRequestStatus* aStatus);
  TInt SendSync(TInt aFunction, const TInt* aPtr, TRequestStatus* aStatus);
  TInt Send(RMessageK* aMsg, TInt aFunction, const TInt* aPtr, TRequestStatus* aStatus);

public:
  inline TBool IsClosing();

public:
  DServer* iServer; // pointer to kernel-side server object
  SDblQueLink iServerLink; // link to attach session to server
  const TAny* iSessionPtr;
  // pointer to server-side CSession2 (user cookie)
  TUint16 iTotalAccessCount;
  TUint8 iSessionType; // TIpcSessionType
  TUint8 iSvrSessionType;// TIpcSessionType
  TInt iMsgCount;
  // total number of outstanding messages on this session
  TInt iMsgLimit;
  // max number of outstanding messages on this session
  SDblQue iMsgQ;
```

```
// q of outstanding msgs on this session (by iSessionLink)
RMessageK* iNextFreeMessage; // pointer to next free message in
// per-session message pool, if any
RMessageK* iPool; // pointer to per-session message pool, if any
RMessageK* iConnectMsg; // pointer to connect msg, if any
RMessageKBase iDisconnectMsg; // vestigial disconnect message
};
```

The session is a standard kernel reference-counted object that user-mode clients hold references to via handles (I'll discuss this mechanism in the next chapter, *Kernel Services*). However, the lifetime of the session must extend beyond that given by client handles, because it needs to stay in existence whilst the server processes the disconnect message that is sent to the server when a session is closed. To do this, we use `iTotalAccessCount`, modified under the protection of the system lock, to keep track of both whether there are any client references and when the session is attached to a server, giving it a count of 1 for either, or 2 for both. The **IDLE** state is equivalent to `iTotalAccessCount` reaching 0.

The creation of a session is performed as two distinct operations in two separate executive calls by the client - firstly the creation of the kernel-side session object itself and secondly the sending of the connect message to the server. The second part of this operation is identical to sending any other message to a server. However, since both a client can connect asynchronously and the server can create a session asynchronously it is now possible for a client to close its handle to the session before the session object has been created in the server.

Normally, closing the last handle to the session would result in a disconnect message being immediately delivered to the server, but if this were done in this case the disconnect message could be accepted by the server, and would contain a null cookie as the session object had not yet been created. There would then be a race between the server setting the cookie after creating the session object and it completing the disconnect request. If the disconnect message is completed first, the kernel-side session object is no longer valid when the server tries to set the cookie using the connect message and it will be panicked. Otherwise, if the session cookie is set first, then the disconnect message will complete as a no-op, since the cookie within it is null, and a session object is leaked in the server.

To avoid either of these situations, we have introduced the **CLOSING** state. If a connect message has been delivered but not yet completed when the last user handle to the session is closed, the delivery of the disconnect message to the server is delayed until the session cookie is updated to indicate successful creation of a user-side session object. If the connect message completes without the cookie having been updated, there is no user-side session object to clean up but a disconnect message is still sent to ensure the lifetime of the session object extends until the completion of other messages which may have been sent to the unconnected session. If there is an undelivered connect message, this is immediately removed from the queue to avoid the possibility of an orphan session object being created.

Note that the ability to create process-relative session handles and the asynchronous nature of sending a connect message mean that you can send other messages to a server both before and after a connect message has been sent, and before the connect message has been completed, so care must be taken over this possibility. However, when a disconnect message is sent no more messages may be sent to the session by virtue of the fact that it is sent once all user-side handles to the session have been closed. Therefore the disconnect message is always the last message to a session that is completed by the server and the session object may be safely destroyed after completing it, without causing lifetime issues for the server.

There are two methods to send synchronous and asynchronous messages - `SendSync()` and `Send()` (first overload above), respectively. These validate certain preconditions, select an appropriate message object to store the new message, and then pass the selected message object to this method:

```
Send(RMessageK* aMsg, TInt aFunction, const TInt* aPtr, TRequestStatus* aStatus);
```

This method populates the message object, increments the current thread IPC count (`DThread::iIpcCount`), adds the message to the session queue and delivers it to the server. If the server has terminated, or is in the process of doing so, sending the message fails immediately.

Neither Send() method permits the sending of a disconnect message, since this is sent automatically by `DSession::Close()` when the last client session handle is closed.

At the other end of a normal message's life, it is completed in `ExecHandler::MessageComplete()` (called from `RMessage-Ptr2::Complete()`). If the message is a disconnect message, then execution is simply transferred to

`DSession::CloseFromDisconnect()`. If not, then what happens next is pretty much the reverse of the send procedure: the message is removed from the session queue, the sending thread's IPC count is decremented and the client thread's request is completed. The only exception to this is if the session is closing; this happens if the client has closed all handles to the session but the disconnect message has not yet been completed by the server. In this case the client thread's request is not completed.

Messages can also be completed from `DSession::Detach()`,which is called either when the server terminates (that is, when the last reference to the server is closed in `DServer::Close()`) or when completing a disconnect message. In this case, the message is again removed from the session queue, the sending thread's IPC count decremented and the client request is completed (if the session is not closing).

We have just seen that it is possible for a message not to be completed - this happens when the session is closing, as we saw above. And yet previously I said that guaranteed message completion is one of the properties of the client-server system. The explanation here is that the client having an outstanding message when calling Close() on a session is considered to be a client-side programming error. There is clearly a race between the server completing such outstanding asynchronous requests and the disconnect request being processed. Those requests processed after the session has been closed cannot be completed whereas those before can, so the behavior of client-server has always been undefined by Symbian in this situation.

The actual behavior of EKA2 differs from EKA1 here. In EKA1, disconnect messages overtake all undelivered messages to the server and these undelivered messages are discarded. Now, in EKA2, the server processes all delivered messages before processing the disconnect message - although, as we've seen, such messages can still not be completed to the client. All this said, we don't advise you to rely on this new behaviour of EKA2, because we explicitly state this to be a programming error and may change the behavior of this area in future.

One of the other main functions of the session object is to control the session's *sharability*. With the advent of a fully message-centric design, there is no requirement for the session to hold a handle to the client thread, and providing different levels of accessibility to a session - restricted to one thread, restricted to one process or sharable with any thread - is now a simple matter of recording the stated intentions of the user-mode server in `iSvrSessionType` and then creating a handle to the session for any client that is allowed access to it, when requested. That is, whether a given thread can access a session is now determined purely by whether it has a handle to the session or not as the access check is performed at handle-creation time.

This accounts for the introduction of the new method `DObject::RequestUserHandle()`. Suppose a server only supports non-sharable sessions. Then a user-mode thread with a handle to a session could just duplicate that handle, making a process-relative handle, and over-ride the settings of the server. But `DSession`'s over-ridden `RequestUserHandle()` checks the value in `iSvrSessionType` to see whether the requested sharing level is allowed by the server, and thereby enforces the user-mode server's requested policy on session sharing.

To maintain backwards compatibility, the user-side APIs for creating sessions default to creating a session that is *unshareable*, even if shared sessions are supported by the server. This *current sharability level* - specified when creating the initial handle to the session, is stored in iSessionType and is validated against the *sharability* level the server supports. To share this session with other threads, the session has either to explicitly create a session that supports the level of sharability required (the preferred method) or subsequently call `ShareAuto()` (to share within process) or `ShareProtected()` (to share between processes), as required. If the ShareXxx()method succeeds, it creates a new process-relative handle and closes the old one. The new session creation overloads that allow the *sharability* of a session to be specified from session creation are the preferred method, as they avoid the expensive operation of creating a new handle where it is not needed.

These new session creation overloads also support an optional security policy that the client can use to verify the security credentials of the server it is connecting to. Similarly, there are overloads of the new APIs to open a handle to a session shared over client-server IPC or from a creator process which allow the session to be validated against a security policy. This allows you to prevent a handle to a spoof server being passed by these mechanisms, as you may verify the identity of the server whose session you are accepting a handle to. The initial session type and security policy parameters are then marshalled into the call to `DSession::Add()`, where they are used to fail session creation if the server does not meet the required policy.

The final responsibility of the session is to find - and allocate if necessary - kernel memory for messages to be stored in. I will discuss these message pools in the next section. At session creation time, you can specify whether the session uses a pool specific to the session or a global kernel pool. The session stores the type of pool it is using in `iPool`. If it is using a per-session pool, then it maintains a pointer to the next available free message in `iNextFreeMessage`.

During a send, the session will then use one of the session's disconnect message, the thread's synchronous message or the next free message from the selected pool. If the session is using the global pool and there are no more free messages the system lock is relinquished (to avoid holding it for an unbounded period of time whilst allocating), the message pool is expanded, then the lock is reclaimed and sending proceeds as before.

Messages - minimal states and message pool design

Next, I'll consider the design of message pools, that is, the memory used to store messages within the kernel. There is one important constraint on the design of the message pools, namely that there must always be a free message available for a session to send a disconnect message to the server, so that resources may be correctly freed in OOM situations. This disconnect message is naturally associated with the session whose disconnection it is notifying and will always be available if the message is embedded within the session object itself. To minimize the memory used by this disconnect message object, we have designed the message object to have a base class, `RMessageKBase`, which contains only the data required for the disconnect message, and then derive from it the (larger) message class, `RMessageK`, which is used for normal messages:

```
class RMessageKBase : public SDblQueLink
  {
public:
  TBool IsFree() const { return !iNext; }
  TBool IsDelivered() const
    { return iNext!=0 && (TLinAddr(iNext) & 3)==0; }
  TBool IsAccepted() const
    { return ((TLinAddr)iNext & 3)==3; }

public:
  TInt iFunction;
  };

class RMessageK : public RMessageKBase
  {
public:
  enum TMsgType {EDisc=0, ESync=1, ESession=2, EGlobal=3};
  inline TInt ArgType(TInt aParam) const;
  inline TInt Arg(TInt aParam) const;
  void Free();
  static RMessageK* NewMsgBlock(TInt aCount, TInt aType);
  IMPORT_C DThread* Thread() const;
  static RMessageK* MessageK(TInt aHandle, DThread* aThread);
  IMPORT_C static RMessageK* MessageK(TInt aHandle);

public:
  TInt iArgs[4];
  TUint16 iArgFlags; // describes which arguments are descriptors/handles
  TUint8 iPool; // 0=disconnect msg, 1=thread sync message,
    // 2=from session pool, 3=from global pool
  TUint8 iPad;
  DSession* iSession; // pointer to session
  SDblQueLink iSessionLink; // attaches message to session
  DThread* iClient; // pointer to client thread (not reference counted)
  TRequestStatus* iStatus; // pointer to user side TRequestStatus
  };
```

After we have ensured that session cleanup works properly, the next most important concern in designing the message allocation strategy is to minimize the memory that the kernel uses for sending messages. In the original client-server implementation of Symbian OS v5, a fixed number of message objects were allocated for each session, resulting in poor message object utilization, considering that most IPC calls were synchronous and hence only one of the message objects was in use at any one time!

Analysis of the client-server system, by instrumenting EUSER and EKERN, has shown that as many as 99% of the calls to `RSession-Base::SendReceive()` are to the synchronous overload. Obviously, a thread cannot send a synchronous message to more than one server at a time, because it waits for the synchronous message's completion inside EUSER immediately after dispatching it. This means that we can use a per-thread message object to avoid having to allocate message objects for all the synchronous messages that are sent. This message object (RMessageK) is embedded within the `DThread` object, and therefore

avoids allocation issues by being allocated as part of the thread object itself.

Thus all that remains to be determined is the allocation strategy for message objects used by asynchronous IPC (such messages are typically used for remote I/O such as sockets or for event notification). These message objects are allocated on a per-session basis, or dynamically from a global pool. As we only need a small number of them, the overhead for non-utilization of these message objects is not large.

You may wonder why we do not insist on a global pool, and cut memory requirements further. This is because for real-time code a guaranteed response time is important, and a global, dynamic pool does not provide those guarantees (as it may require memory allocation in the kernel). This means that we must provide the option of creating a per-session pool, which allows the real-time code to manage the time it takes to process an asynchronous request precisely. That said, it is more common for servers to use asynchronous message completion for event notification, in which case using the dynamic global pool becomes more attractive due to its smaller memory footprint. This is therefore the recommended option where real-time guarantees are not required for any asynchronous IPC calls to the server.

This allocation scheme allows any number of threads to invoke synchronous IPC on a server using the same session without having to increase the session's message pool and it also provides guaranteed message sending for synchronous IPC.

We have achieved further memory savings by minimizing the state that is required within the message objects themselves. There are only three states that a message can have, as shown in Figure 4.5.

FREE - The message is not currently in use

DELIVERED - The message has been sent but the server hasn't seen it yet

ACCEPTED - The server has received the message but not yet completed it.

To assure a message's cleanup when a session closes, we attach it to a session queue whilst it is not free. We also have to ensure no message can persist beyond the lifetime of the thread that sent it, as such a message can no longer be completed. By assuming a strategy that messages should never be discarded prematurely (for example, when the thread Exit()s), but only at the last possible moment (just before the thread object is destroyed), we can avoid the complication of maintaining an open reference on the thread and the associated state required for this. When a thread exits, therefore, we need not iterate through a queue to discard **DELIVERED** messages - they are simply allowed to propagate through the server as usual. Instead of an open reference on the thread in each message, we need only maintain a count of the outstanding messages for each thread (in `DThread::iIpcCount`).

When a thread exits it checks this count and if it is non-zero it increments its own reference count so it is not destroyed and sets a flag (the top bit of the message count) to indicate this has been done. Completing a message decrements the outstanding message count for the client thread and if its value reaches 0x80000000, this means the message count for the thread has reached zero and the flag has been set. The extra reference on the thread is then closed, allowing it to be safely destroyed.

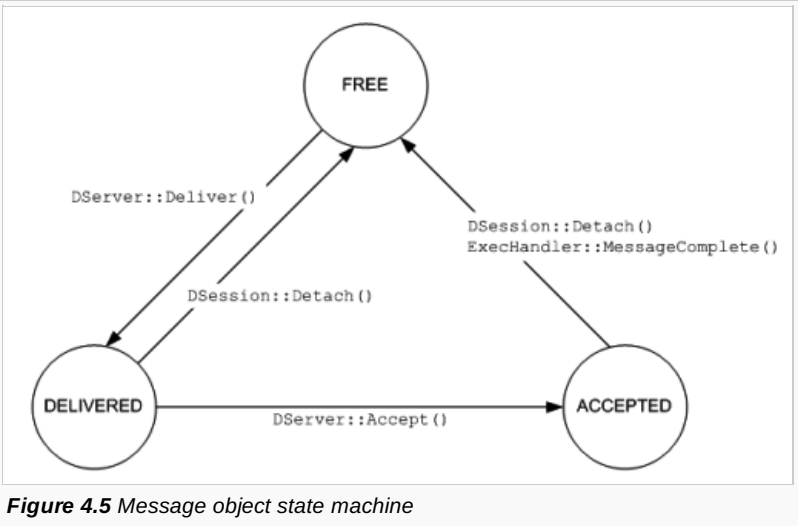


Figure 4.5 *Message object state machine*

Closing a session does not unilaterally discard **DELIVERED** messages either - again they are simply allowed to propagate through the server. This means that session close needs only to send the disconnect message to the server and has no need to iterate its queue of messages.

So, we only need to perform message queue iteration either when the server itself terminates or when the server completes a disconnect message. In the latter case, the iteration is needed to free any remaining **ACCEPTED** messages (no **DELIVERED** messages can remain since the disconnect message is guaranteed to be the last message received from that session). The messages concerned are not on any other queue so there is no contention when we iterate over the session queue. When the

server itself terminates, the complete system of server and sessions is frozen, because clients are not allowed to send a message to the server whilst it is terminating, so again there is no contention on the session queue when a message is being completed and hence no need for an intermediate **COMPLETED** state to deal with delayed removal of messages from the session queue.

The three states are then encoded in a minimal way into the doubly linked list fields:

	iLink.iNext	**iLink.iPrev**
FREE	NULL	N/A
DELIVERED	Valid address (multiple of 4) [bottom bits == 00b]	N/A
ACCEPTED	~(this) [bottom bits == 11b]	~&(server `DProcess`)

4.1.5.4 Message handles

Another key design decision we took for message objects was not to derive them from `DObject`. This means that they do not have standard object containers and user handles. Rather, the user-side handle for an `RMessageK` is in fact its address on the kernel heap. To verify the handles that user-mode operations give, the kernel uses the fact that a user-side component only ever has a valid handle to a message when it is in the **ACCEPTED** state and does the following:

- Checks the address and size of the object are within the kernel heap
- If so, reads the memory under an exception trap (the machine coded versions of these functions use the magic exception immunity mechanism,

see Section 5.4.3.1)

- Check that msg.iNext == ~(& msg)
- Check that msg.iPrev == ~(requestingThread->iOwning Process).

If these tests pass, then the message object is taken to be valid and the requested operation can proceed.

Changes for IPCv2

The main change from the IPCv1 implementation of client-server has been the removal of the insecure APIs. These were:

```
class CSharableSession
class CSession
class CServer
class RMessagePtr
class RMessage
class RServer

RSessionBase::Share(TAttachMode aAttachMode=EExplicitAttach)
RSessionBase::Attach()
RSessionBase::Send(TInt aFunction,TAny* aPtr)
RSessionBase::SendReceive(TInt aFunction,TAny* aPtr, TRequestStatus& aStatus)
RSessionBase::SendReceive(TInt aFunction,TAny* aPtr)
RSubSessionBase::CreateSubSession(RSessionBase&,TInt aFunction, const TAny* aPtr)
RSubSessionBase::Send(TInt aFunction,const TAny* aPtr)
RSubSessionBase::SendReceive(TInt aFunction,const TAny* aPtr, TRequestStatus&)
RSubSessionBase::SendReceive(TInt aFunction,const TAny* aPtr)

RThread::GetDesLength(const TAny* aPtr)
RThread::GetDesMaxLength(const TAny* aPtr)
RThread::ReadL(const TAny* aPtr,TDes8& aDes,TInt anOffset)
RThread::ReadL(const TAny* aPtr,TDes16 &aDes,TInt anOffset)
RThread::WriteL(const TAny* aPtr,const TDesC8& aDes,TInt anOffset)
RThread::WriteL(const TAny* aPtr,const TDesC16& aDes,TInt anOffset)
RThread::RequestComplete(TRequestStatus*& aStatus,TInt aReason)
RThread::Kill(TInt aReason)
RThread::Terminate(TInt aReason)
RThread::Panic(const TDesC& aCategory,TInt aReason)
```

We have replaced these APIs with the framework I have described above. To aid migration to the new APIs, all EKA1-based

Symbian OS releases from 7.0 s onwards contain both the new Rendezvous() thread/process APIs discussed in Section 3.3.6, and a functioning (but not secure) implementation of the IPCv2 APIs.

IPCv2 summary

To summarize, IPCv2 has brought the following benefits and features:

- Secure access to the client's address space based on the permissions and data types it specifies in messages to the server
- The ability to prevent spoofing of server names and to validate a security policy against the server when connecting to a session or opening a handle to a shared session
- Real-time server performance is possible through the use of asynchronous session connect and disconnect operations
- The ability to share server sessions between processes
- Asynchronous creation of a connection with a server to prevent a malicious server blocking a client indefinitely.

4.2 Asynchronous message queues

A message queue is a mechanism for passing data between threads, which may be in the same process, or in different processes. The message itself is usually an instance of a class, and its size must be a multiple of 4 bytes.

The asynchronous message queue mechanism provides a way to send a message without needing to know the identity of a recipient, or indeed, if anyone is actually listening.

You define and fix the size of the queue when you create it, choosing the maximum number of messages it can contain and the size of those messages. So, you would normally create a new queue to deal with messages of a particular type. There is no fixed maximum to either the message size or the queue size - these are only limited by system resources.

Many readers and writers may share a single queue. Sending and receiving messages are real-time operations and operate with real-time guarantees.

We represent a message queue by a `DMsgQueue` kernel-side object, to which the reader and the writer can open a handle. This is a reference counted object derived from `DObject`, which means that it is not persistent; the kernel deletes it when the last handle to it is closed. The queue itself is simply a block of memory divided into slots. Here is the `DMsgQueue` class that manages it:

```
class DMsgQueue : public DObject
  {
public:
  enum TQueueState {EEmpty, EPartial, EFull};
  enum {KMaxLength = 256};

public:
  ~DMsgQueue();
  TInt Create(DObject* aOwner, const TDesC* aName, TInt aMsgLength, TInt aSlotCount,
TBool aVisible = ETrue);
  TInt Send(const TAny* aPtr, TInt aLength);
  TInt Receive(TAny* aPtr, TInt aLength);
  void NotifySpaceAvailable(TRequestStatus* aStatus);
  void NotifyDataAvailable(TRequestStatus* aStatus);
  void CancelSpaceAvailable();
  void CancelDataAvailable();
  TInt MessageSize() const;

private:
  void CancelRequest(DThread* aThread, TRequestStatus*& aStatus);
  void CompleteRequestIfPending(DThread* aThread, TRequestStatus*& aStatus, TInt
aCompletionVal);

private:
  TUint8* iMsgPool;
  TUint8* iFirstFreeSlot;
  TUint8* iFirstFullSlot;
  TUint8* iEndOfPool;
```

```
  DThread* iThreadWaitingOnSpaceAvail;
  DThread* iThreadWaitingOnDataAvail;
  TRequestStatus* iDataAvailStat;
  TRequestStatus* iSpaceAvailStat;
  TUint16 iMaxMsgLength;
  TUint8 iState;
  TUint8 iSpare;

public:
  friend class Monitor;
  };
```

Key member data of `DMsgQueue`

Field	Description
`iMsgPool`	A pointer to the block of memory used for the message slots.
`iFirstFreeSlot`	A pointer to the first free slot in the message pool, unless the pool is full, `iState==EFull`.
`iFirstFullSlot`	A pointer to the first full slot in the message pool, unless the pool is empty, `iState==EEmpty`.
`iEndOfPool`	A pointer to the byte of memory that is just past the end of the poll of message slots.
`iState`	Whether the pool is empty, full or somewhere in between.

You perform actions (such as creation, opening, writing and reading) to a message queue through a message queue handle, which is an `RMsgQueue` object. This is a templated class, where the template parameter defines the message type.

RMsgQueue is derived from `RMsgQueueBase`, which together form a thin template class/base class pair. `RMsgQueueBase` provides the implementation, while `RMsgQueue` provides type safety. An `RMsgQueueBase` object is a valid message queue handle, but does not offer the type safety that `RMsgQueue` does.

Visibility

A message queue can be:

1. Named and be visible to all processes - a global queue
2. Nameless, but accessible from other processes. A handle may be passed to another process by a process currently owning a handle to the queue, using a handle-sharing mechanism - a protected queue
3. Nameless and local to the current process, hence not visible to any other process - a local queue.

The choice clearly depends on the use you have in mind for the queue.

Kernel-side messages

Kernel-side messages are a means of communication that are used to communicate with a Symbian OS thread that is executing kernel-side code. Typically, you would use this communication method if you were writing a device driver - to communicate between your client thread, usually a user-mode thread, and a supervisor-mode thread running the actual device driver code.

The mechanism consists of a message containing data, and a queue that is associated with a DFC. The DFC runs to process each message.

We represent a kernel-side message by a `TMessageBase` object; this allows a single 32-bit argument to be passed, and returns a single 32-bit value. If you want to pass more arguments, then you must derive a new message class from `TMessageBase`.

Every Symbian OS thread has a `TThreadMessage` object embedded within it. `TThreadMessage` is derived from `TMessageBase`, and contains space for 10 extra 32-bit arguments. You can use these objects for communication with device driver threads.

Both `TMessageBase` and `TThreadMessage` are defined in kernel.h. The following example shows the `TMessageBase` class:

```
class TMessageBase : public SDblQueLink
  {
public:
  enum TState {EFree,EDelivered,EAccepted};
```

```
public:
  TMessageBase() : iState(EFree), iQueue(NULL) {}
  IMPORT_C void Send(TMessageQue* aQ);
  IMPORT_C TInt SendReceive(TMessageQue* aQ);
  IMPORT_C void Forward(TMessageQue* aQ, TBool aReceiveNext);
  IMPORT_C void Complete(TInt aResult, TBool aReceiveNext);
  IMPORT_C void Cancel();
  IMPORT_C void PanicClient(const TDesC& aCategory, TInt aReason);

public:
  IMPORT_C DThread* Client();

public:
  TUint8 iState;
  TMessageQue* iQueue;
  NFastSemaphore iSem;
  TInt iValue;
  };
```

Key member data of `TMessageBase`

Field	Description
`iState`	Indicates whether message is free, delivered or accepted.
`iQueue`	A pointer to the message queue to which the message was delivered.
`iSem`	A fast semaphore used to block the sending thread if the message was sent synchronously. The `iOwningThread` field of this semaphore is used as a pointer to the thread that sent the message.
`iValue`	Used to hold a single integer argument when the message is sent; holds completion code when message is completed.

TMessageQue The kernel sends kernel-side messages to a message queue, which is represented by a `TMessageQue` object. This consists of a DFC and a doubly linked list of received messages. The class is shown below:

```
class TMessageQue : private TDfc
  {
public:
  IMPORT_C TMessageQue(TDfcFn aFunction, TAny* aPtr, TDfcQue* aDfcQ, TInt aPriority);
  IMPORT_C void Receive();
  IMPORT_C TMessageBase* Poll();
  IMPORT_C TMessageBase* Last();
  IMPORT_C void CompleteAll(TInt aResult);
  using TDfc::SetDfcQ;

public:
  inline static void Lock() {NKern::FMWait(&MsgLock);}
  inline static void Unlock() {NKern::FMSignal(&MsgLock);}
  inline void UnlockAndKick() {Enque(&MsgLock);}

public:
  SDblQue iQ;
  TBool iReady;
  TMessageBase* iMessage;
  static NFastMutex MsgLock;
  friend class TMessageBase;
  };
```

Key member data of `TMessageQue`

Field	Description
TDfc (the base class)	This DFC is attached to the thread receiving the messages. It runs whenever the message queue is ready to receive and a message is available.
iQ	A doubly linked list of messages that have been delivered to this queue.
iReady	A Boolean flag indicating whether the message queue is ready to receive. If TRUE, the DFC will run as soon as a message is delivered; if FALSE the message will simply remain on the delivered queue and the DFC will not run.
iMessage	Pointer to the last message accepted by the receiving thread.

Kernel-side messaging in operation

When a message is sent to the queue, either:

- The kernel accepts the message immediately, and the receiving thread's DFC runs. This happens if the message queue is ready to receive, which is the case if the message queue is empty and the receiving thread has requested the next message.

Or

- The kernel places the message on the delivered message queue, and the DFC does not run. This happens if there are other messages queued ahead of this one or if the receiving thread has not (yet) requested another message.

A kernel-side message may be in one of three states at any time:

1. FREE - represented by the `TMessageBase::EFree` enum value. This indicates that the message is not currently in use
2. DELIVERED - represented by the `TMessageBase::EDelivered` enum value. This indicates that the message is attached to a message queue but is not currently in use by the receiving thread. It may be removed from the queue and discarded with no ill effects on the receiving thread
3. ACCEPTED - represented by the `TMessageBase::EAccepted` enum value. This indicates that the message is not attached to a message queue but is currently in use by the receiving thread. The message may not be discarded.

Transitions between these states, including adding the message to and removing it from a message queue, occur under the protection of the global `TMessageQue::MsgLock` fast mutex. We need to use a mutex to avoid queue corruption in the case of, for example, multiple threads sending to the same message queue at the same time. By using a fast mutex, we ensure that message-passing operations may only be invoked from a thread context.

You can send kernel-side messages either synchronously or asynchronously. Each `TMessageBase` object contains an `NFastSemaphore` on which the sending thread will wait after sending a synchronous message. The receiving thread signals the semaphore after the kernel has processed the message and written the completion code. The kernel then releases the sending thread, and when it runs, it picks up the return code.

The `NFastSemaphore` also contains a pointer to the sending `NThread`; this serves to identify the sending thread and is therefore set up for both synchronous and asynchronous message send. We reference count this pointer - incrementing the access count of the originating `DThread` when the message is sent. This prevents the sending DThread object disappearing if the thread terminates unexpectedly.

When the kernel completes the message it removes the extra access asynchronously - the thread completing the message will not need to close the `DThread` itself. We do this to avoid unpredictable execution times for message completion. Also note that even messages that are sent asynchronously must be completed; this is so that the kernel can set the message state back to FREE and remove the access count from the sending thread.

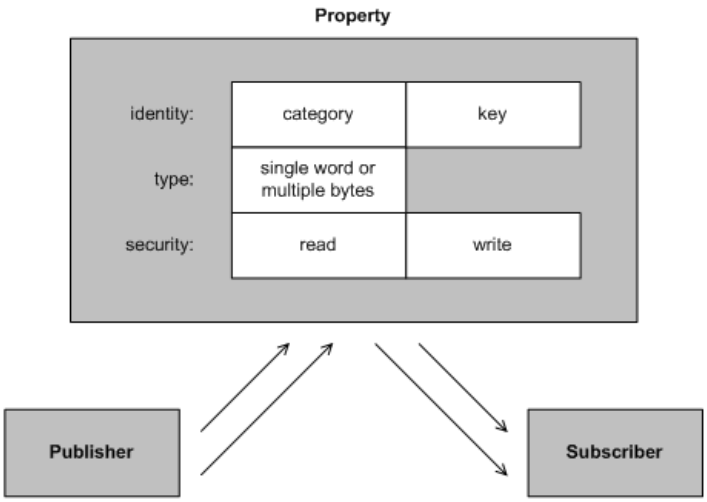
The kernel always sends the `TThreadMessage` objects embedded in Symbian OS thread control blocks synchronously - this ensures that one message per thread will always suffice. The kernel cancels these messages if the corresponding thread terminates. Canceling an ACCEPTED message has no effect, but canceling a DELIVERED message means that the kernel will remove the message from the queue and also remove the access count held by the message on the sending thread. Because of this, the receiving thread should only use any of the member data of TMessageBase if the message is in the ACCEPTED state.

Publish and subscribe

Publish and subscribe, also known as *properties*, provides:

1. System-wide global variables
2. A new IPC mechanism, for asynchronous peer-to-peer communication between threads.

An overview is given in Figure 4.6.



Publish and subscribe can be used by both user and kernel code, through similar APIs, and so this method also allows communication between user and kernel code.

From the user side, you would use the `RProperty` handle, defined in e32property.h:

```
class RProperty : public RHandleBase
  {

public:
  enum { KMaxPropertySize = 512 };
  enum { KMaxLargePropertySize = 65535 };
  enum TType
    {
    EInt,
    EByteArray,
    EText = EByteArray,
    ELargeByteArray,
    ELargeText = ELargeByteArray,
    ETypeLimit,
    ETypeMask = 0xff
    };

public:
  IMPORT_C static TInt Define(TUid aCategory, TUint aKey, TInt aAttr, TInt
aPreallocate=0);
  IMPORT_C static TInt Define(TUid aCategory, TUint aKey, TInt aAttr, const
TSecurityPolicy& aReadPolicy,
    const TSecurityPolicy& aWritePolicy, TInt aPreallocated=0);
  IMPORT_C static TInt Delete(TUid aCategory, TUint aKey);
  IMPORT_C static TInt Get(TUid aCategory, TUint aKey, TInt& aValue);
  IMPORT_C static TInt Get(TUid aCategory, TUint aKey, TDes8& aValue);
  IMPORT_C static TInt Set(TUid aCategory, TUint aKey, TInt aValue);
  IMPORT_C static TInt Set(TUid aCategory, TUint aKey, const TDesC8& aValue);
  IMPORT_C TInt Attach(TUid aCategory, TUint aKey, TOwnerType aType = EOwnerProcess);
  IMPORT_C void Subscribe(TRequestStatus& aRequest);
  IMPORT_C void Cancel();
  IMPORT_C TInt Get(TInt& aValue);
  IMPORT_C TInt Get(TDes8& aValue);
  IMPORT_C TInt Set(TInt aValue);
  IMPORT_C TInt Set(const TDesC8& aValue);
  };
```

From the kernel side, you use the `RPropertyRef` and `TPropertySubsRequest` classes defined in sproperty.h, and the `TPropertyInfo` class defined in u32property.h. Note that `TPropertySubsRequest` is on a single queue protected by the system lock.

```
class RPropertyRef
  {
public:
  RPropertyRef() {iProp = NULL;}
  IMPORT_C TInt Attach(TUid aCategory, TInt aKey);
  IMPORT_C TInt Open(TUid aCategory, TInt aKey);
  IMPORT_C void Close();
  IMPORT_C TInt Define(TInt aAttr, const TSecurityPolicy& aReadPolicy, const
TSecurityPolicy& aWritePolicy,
    TInt aPreallocate=0, DProcess* aProcess = NULL);
  IMPORT_C TInt Delete(DProcess* aProcess = NULL);
  IMPORT_C TInt Subscribe(TPropertySubsRequest& aRequest, DProcess* aProcess = NULL);
  IMPORT_C void Cancel(TPropertySubsRequest& aRequest);
  IMPORT_C TInt Get(TInt& aValue, DProcess* aProcess = NULL);
  IMPORT_C TInt Set(TInt aValue, DProcess* aProcess = NULL);
  IMPORT_C TInt Get(TDes8& aDes, DProcess* aProcess = NULL);
  IMPORT_C TInt Set(const TDesC8& aDes, DProcess* aProcess = NULL);
  IMPORT_C TBool GetStatus(TPropertyStatus& aStatus);

private:
  TProperty* iProp;
  };

class TPropertySubsRequest : public SdblQueLink
  {
public:
  TPropertySubsRequest(TPropertyCompleteFn aCompleteFn, TAny* aPtr)
    {
    iNext = NULL;
    iCompleteFn = aCompleteFn;
    iPtr = aPtr;
    }
  TPropertyCompleteFn iCompleteFn;
  TAny* iPtr;

private:
  friend class TProperty;
  DProcess* iProcess;
  };

class TPropertyInfo
  {
public:
  TUintiAttr;
  TUint16 iSize;
  RProperty::TType iType;
  TSecurityPolicy iReadPolicy;
  TSecurityPolicy iWritePolicy;
  };
```

Key entities

There are three key entities in the publish and subscribe system. I will describe them below; you may also want to refer back to

the overview in Figure 4.6.

Entities	Description
Properties	This is data: either a single 32-bit data value or a variable-length set of bytes, identified by a 64-bit integer.
Publishers	Publishers are threads that define and update a property.
Subscribers	Subscribers are threads that listen for changes to a property and can get the current value of a property.

Now let's have a look at properties in a little more detail.

Properties

Internally, the kernel stores a property as an instance of the `TProperty` class, defined in sproperty.cpp. I will give a very cut-down version here, as this is a surprisingly large class:

```
class TProperty
{
public:
  static TInt Init();
  static TInt Attach(TUid aCategory, TUint aKey, TProperty** aProp);
  static TInt Open(TUid aCategory, TUint aKey, TProperty** aProp);
  void Close();
  TInt Define(const TPropertyInfo*, DProcess*);
  TInt Delete(DProcess*);
  TInt Subscribe(TPropertySubsRequest* aSubs, DProcess*);
  void Cancel(TPropertySubsRequest* aSubs);
  TInt GetI(TInt* aValue, DProcess*);
  TInt GetB(TUint8* aBuf, TInt* aSize, DProcess*, TBool aUser);
  TInt SetI(TInt aValue, DProcess*);
  TInt SetB(const TUint8*, TInt aSize, DProcess*, TBool aUser);
  const TUid iCategory;
  const TUint iKey;

private:
  enum { KCompletionDfcPriority = 2 };
  static TDfc CompletionDfc;

  static SDblQue CompletionQue;
  static DMutex* FeatureLock;

  static TProperty* Table[KHashTableLimit];
  TUint8 iType;
  TUint8 iAttr;
  TCompiledSecurityPolicy iReadPolicy;
  TCompiledSecurityPolicy iWritePolicy;
  TUint32 iOwner;
  TUint iRefCount;

  // The property value
  // Meaningful for defined properties only
  // (ie. iType != RProperty::ETypeLimit)

  union // the value is protected by the system lock
    {
    TBuf* iBuf;
    TInt iValue;
    };
  };
```

A property has three key attributes: identity, type and security.

The identity and type of a property is the only information that must be shared between a publisher and a subscriber - there is no need to provide interface classes or functions, though that may often be desirable.

Identity

A property is identified by a 64-bit integer made up of two 32-bit parts: the category and the key.

A property is said to belong to a category, which is a standard Symbian OS UID.

The key is a 32-bit value that identifies a specific property within a category. The meaning applied to the key depends on the kind of enumeration scheme set up for the category. At its simplest, a key can be an index value. It can also be another UID, if you are designing the category to be generally extensible.

Type

A property can be:

1. A single 32-bit value
2. A contiguous set of bytes, referred to as a byte-array. The length of this can go as high as `KMaxLargePropertySize`, 65,535 bytes, but real-time guarantees are made only if the length is below `RProperty::KMaxPropertySize`, 512 bytes. Memory for the smaller byte arrays may be allocated at definition time: if this is done publishing cannot fail with `KErrNoMemory`, and we can satisfy those real-time guarantees
3. Unicode text. This is for properties and access or functions that accept Unicode descriptors, and is just a convenience for programmers wishing to store Unicode text properties. The implementation treats Unicode text as a byte-array; the API hides the detail.

Security

A property has two `TCompiledSecurityPolicy` members. One of these is for read operations - that is, `Get()` and `Subscribe()` calls on `RProperty` - and the other is for write operations - that is Set() calls on `RProperty`.

These members are set up when the property is defined, passing in two `TSecurityPolicy` parameters to the `RProperty::Define()` function:

```
IMPORT_C static TInt Define(
  TUid aCategory,
  TUint aKey,
  TInt aAttr,
  const TSecurityPolicy& aReadPolicy,
  const TSecurityPolicy& aWritePolicy,
  TInt aPreallocated=0
  );
```

You can turn to Chapter 8, *Platform Security*, for more information.

Using publish and subscribe

There are six basic operations that you can perform on a property: define, delete, publish, retrieve, subscribe and unsubscribe. I give an overview of these operations in the table below, and in subsequent sections I will describe some of these functions in more detail.

Operations	Description
Define	Create a property variable and define its type and access controls.
Delete	Remove a property from the system.
Publish	Change the value of a property.
Retrieve	Get the current value of a property.
Subscribe	Register for notification of changes to a property.
Unsubscribe	Say that you no longer want to be notified of changes.

Defining a property

As we saw above, you define a property by using the `RProperty::Define()` function to specify the attributes of the property.

You don't have to define a property before it is accessed. This means that either the publisher, or one of the subscribers, may define a property.

On a secure implementation of Symbian OS, outstanding subscriptions at the point of definition may be completed with `KErrPermissionDenied` if they fail the security policy check.

Once defined, the property persists in the kernel until the operating system reboots or the property is deleted. The property's lifetime is not tied to that of the thread or process that defined it. This means that it is a good idea to check the return code from `RProperty::Define()` in case that the property was previously defined and not deleted.

You can delete a property using the `RProperty::Delete()` function. The kernel will complete any outstanding subscriptions for this property with `KErrNotFound`.

Note that only an instance of a process from the same EXE as the process which defined the property is allowed to delete it, as the SID of the process (as defined in the EXE image) is checked against the SID of the defining process. Also note that in a secure implementation of Symbian OS, you may only define a property with a category equal to the SID of the process within which you are executing if the category being defined is greater than `KUidSecurityThresholdCategoryValue`. Any process may define a property with a category less than `KUidSecurityThresholdCategoryValue` if it has the WriteDeviceData capability, to ease migration of legacy code whilst still enforcing security on defining properties.

```
const TUid KMyPropertyCat={0x10012345};
enum TMyPropertyKeys= {EMyPropertyCounter,EMyPropertyName};

// define first property to be integer type
Tint r = RProperty::Define(KMyPropertyCat, EMyPropertyCounter, RProperty::EInt);
if (r!=KErrAlreadyExists)
  User::LeaveIfError(r);

// define second property to be a byte array,
// allocating 100 bytes
r=RProperty::Define(KMyPropertyCat, EMyPropertyName, RProperty::EByteArray,100);
if (r!=KErrAlreadyExists)
  User::LeaveIfError(r);

// much later on...
// delete the property
TInt r=RProperty::Delete(KMyPropertyCat, EMyPropertyName);
if (r!=KErrNotFound)
  User::LeaveIfError(r);
```

Creating and closing a handle to a property

You carry out some property operations (such as defining and deleting properties) by specifying a category and key, but other operations (such as subscribing) require a reference to the property to be established beforehand. Some operations, such as publishing, can be done in either way.

To create a reference to a property, you use the `RProperty::Attach()` member function. After this has completed successfully, the `RProperty` object will act like a normal handle to a kernel resource.

When the handle is no longer required, it can be released in the standard way by calling the inherited `RHandleBase::Close()` member function. You should note that releasing the handle does not cause the property to disappear - this only happens if the property is deleted.

As I said before, it is quite legitimate to attach to a property that has not been defined, and in this case no error will be returned. This enables the lazy definition of properties.

```
// attach to the counter property
RProperty counter;
```

```
Tint r = counter.Attach(KMyPropertyCat, EMyPropertyName, EOwnerThread);
User::LeaveIfError(r);

// use the counter object...

// when finished, release the handle
counter.Close();
```

Publishing and retrieving property values

You can publish properties using the `RProperty::Set()` family of functions, and read them using the `RProperty::Get()` family. You can either use a previously attached `RProperty` handle, or you can specify the property category and key with the new value. The former method is guaranteed to have a bounded execution time in most circumstances and is suitable for high-priority, real-time tasks. If you specify a category and key, then the kernel makes no real-time guarantees. See Section 4.4.8 for more on the real-time behavior of publish and subscribe.

The kernel reads and writes property values atomically, so it is not possible for threads reading the property to get a garbled value, or for more than one published value to be confused.

The kernel completes all outstanding subscriptions for the property when a value is published, even if it is exactly the same as the existing value. This means that a property can be used as a simple broadcast notification service.

If you publish a property that is not defined, the `Get()` and `Set()` functions just return an error, rather than panicking your thread. This happens because you may not have made a programming error, see Section 4.4.4.

```
// publish a new name value
TFileName n;
RProcess().Filename(n);
TInt r=RProperty::Set(KMyPropertyCat,EMyPropertyName,n);
User::LeaveIfError(r);

// retrieve the first 10 characters of the name value
TBuf<10> name;
r=RProperty::Get(KMyPropertyCat,EMyPropertyName,name);
if (r!=KErrOverflow)

User::LeaveIfError(r);

// retrieve and publish a new value using the attached counter property
TInt count;
r=counter.Get(count);
if (r==KErrNone)

r=counter.Set(++count);
User::LeaveIfError(r);
```

If another thread is executing the same sequence to increment count, then this last example contains a race condition!

Subscribing to properties

A thread requests notification of property update using the `RProperty::Subscribe()` member function on an already attached property object. You can only make a single subscription from a single `RProperty` instance at any one time, and you can cancel this subscription request later with the `RProperty::Cancel()` member function.

If you subscribe to a property, you are requesting a single notification of when the property is next updated. The kernel does not generate an ongoing sequence of notifications for every update of the property value. Neither does the kernel tell you what the changed value is. Essentially, the notification should be interpreted as *Property X has changed* rather than *Property X has changed to Y*. You must explicitly retrieve the new value if you need it. This means that multiple updates may be collapsed into one notification, and that you, as the subscriber, may not have visibility of all the intermediate values.

This might appear to introduce a window of opportunity for a subscriber to be out of sync with the property value without receiving notification of the update - in particular, if the property is updated again before the subscriber thread has the chance to process the original notification. However, a simple programming pattern (outlined in the example below) ensures this does not happen.

```
// Active object that tracks changes to the name property

class CPropertyWatch : public CActive
  {
  enum {EPriority=0};

public:
  static CPropertyWatch* NewL();

private:
  CPropertyWatch();
  void ConstructL();
  ~CPropertyWatch();
  void RunL();
  void DoCancel();

private:
  RProperty iProperty;
  };

CPropertyWatch* CPropertyWatch::NewL()
  {
  CPropertyWatch* me=new(ELeave) CPropertyWatch;
  CleanupStack::PushL(me);
  me->ConstructL();
  CleanupStack::Pop(me);
  return me;
  }

CPropertyWatch::CPropertyWatch() :CActive(EPriority){}

void CPropertyWatch::ConstructL()
  {
  User::LeaveIfError(iProperty.Attach(KMyPropertyCat, KMyPropertyName));
  CActiveScheduler::Add(this);
  // initial subscription and process current property value
  RunL();
  }

CPropertyWatch::~CPropertyWatch()
  {
  Cancel();
  iProperty.Close();
  }

void CPropertyWatch::DoCancel()
  {
  iProperty.Cancel();
  }

void CPropertyWatch::RunL()
  {
```

```
// resubscribe before processing new value to prevent missing updates
iProperty.Subscribe(iStatus);
SetActive();

// property updated, get new value
TFileName n;
if (iProperty.Get(n)==KErrNotFound)
  {
  // property deleted, do necessary actions here...
  NameDeleted();
  }
else
  {
  // use new value ...
  NameChanged(n);
  }
}
```

Real-time issues

When designing this functionality, we wanted to ensure that publishing a new value to a property was a real-time service, since time-critical threads will surely need to invoke it. For example a communication protocol could use publish and subscribe to indicate that a connection has been established.

However, there can be an arbitrarily large number of subscriptions on any given property, which makes publishing to that property unbounded. We solved this problem by using a DFC queued on the supervisor thread to do the actual completion of subscriptions. The publisher updates the value of the property and the kernel then places the property on a queue of properties for which notifications are outstanding. The DFC, in the supervisor context, then drains the queue and notifies subscribers.

As I showed earlier, you should publish or retrieve properties by using a previously attached `RPropertyhandle`, rather than by specifying the property category and key with the new value. This is guaranteed to have a bounded execution time, unless you are publishing a byte-array property that has grown in size. In this case the kernel will have to allocate memory for the byte-array, and memory allocation is an unbounded operation.

Shared chunks and shared I/O buffers

Shared I/O buffers and shared chunks are mechanisms which allow you to share memory between a user-side process and a kernel-side process, with a minimum of overhead. Such sharing avoids the expensive and time-consuming act of copying (potentially) large amounts of data around the system. Note that shared I/O buffers are a legacy mechanism primarily aimed at providing compatibility with EKA1 and that shared chunks, which are much more efficient and flexible, are the preferred mechanism for sharing memory between device drivers and user threads in EKA2. To understand how these mechanisms work, you need to know a little more about how Symbian OS manages its memory, so I will cover them in Chapter 7, *Memory Models*.

Summary

In this chapter, I have covered several of the mechanisms that EKA2 provides to allow communication between threads: client-server, message queues, publish and subscribe, shared I/O buffers and shared chunks. In the next chapter, I will describe how EKA2 provides services to user mode threads.