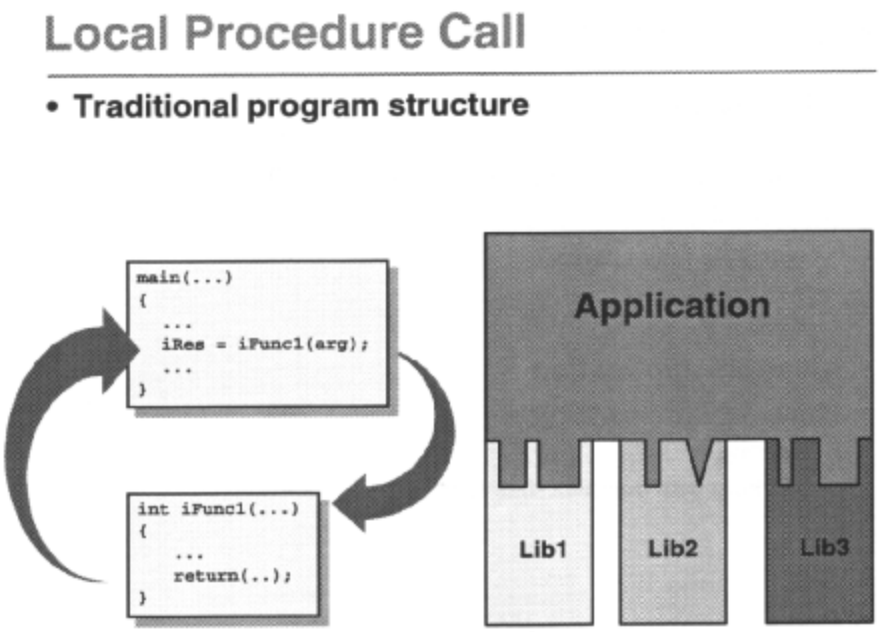


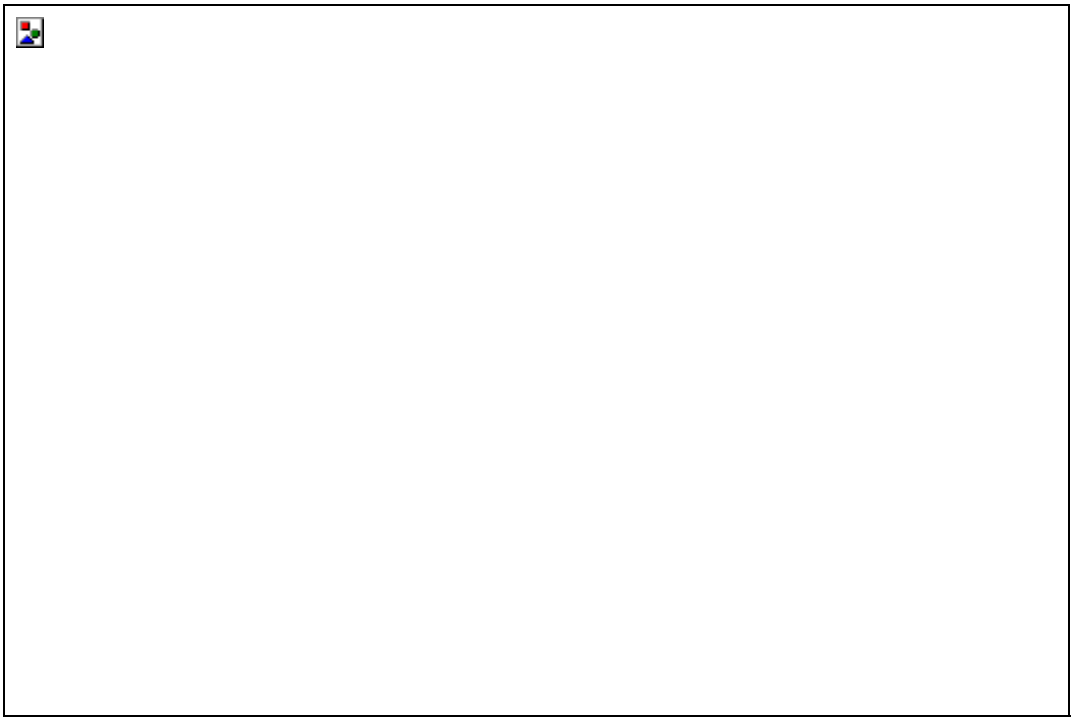
Applications are traditionally composed of a number of a number of *functions,* or *procedures.* Good design principles lead the functions to have a well-defined purpose, and to be as self-contained as possible. They may be included as part of the application itself, or be part of a library that contains a number of functions that are related in some way; for example, a library of mathematical functions.

A function is *called* from another part of the application (usually another function). The calling function may pass *arguments* into the called function; these represent objects to be processed by the called function. The called function may *return* a *result* to the calling function. While the called function is executing, the calling function is suspended.

The specification of the arguments (if any) and result (if any) constitute the *interface* to the function. The interface should represent all that a calling function needs to know about a (called) function to use it.

For example, to use a function to calculate the square root of a number, it is necessary to know that the function accepts (for example) a single-integer argument, and retums a floating-point value that is the square root of its argument. It is *not* necessary to know the algorithm used to calculate the square root.

When an application is built, the *linker* ensures that the necessary code segments are included in the executable image, so that required functions are called correctly as the application executes (either from a *static* or *dynamic* library).

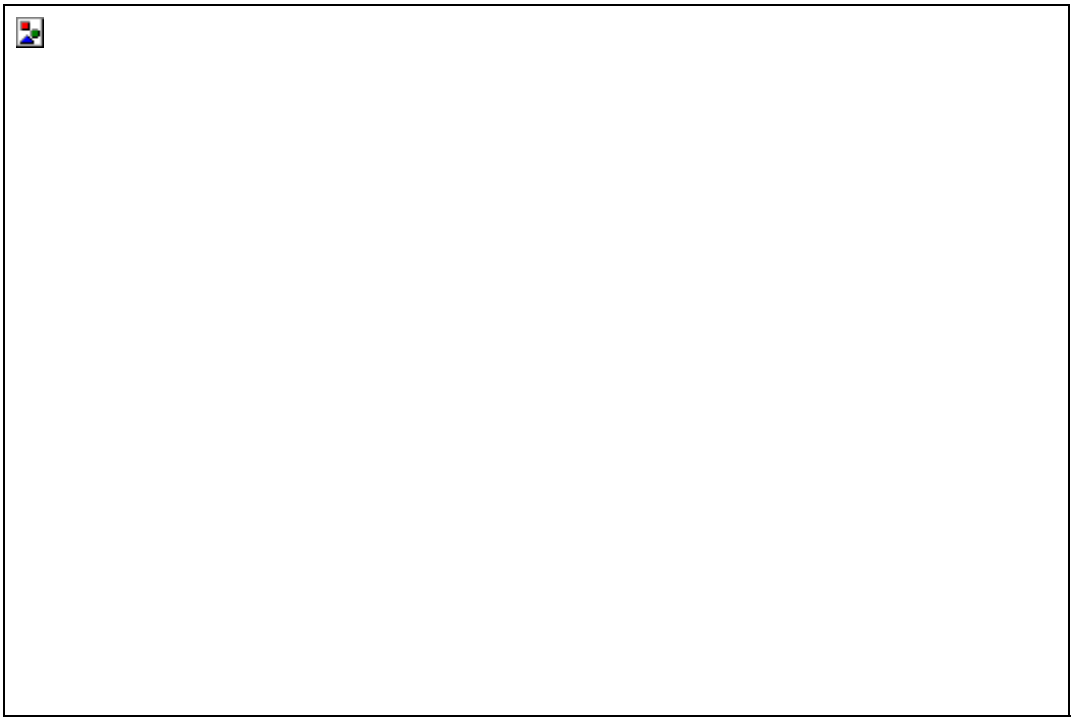


The basic idea of the *Remote Procedure Call* (RPC) paradigm is to extend the concept of calling a procedure or function, so that the caller and the called function can be in separate processes. In particular, the processes can be on different systems in a network.

RPC provides a convenient way of implementing the *Client/Server model* for distributed applications. The calling function is the client and the called function is the server. The most powerful feature of RPC, however, is the relative transparency of the networking/IPC transport mechanism that is used to send the request and arguments to the server, and receive the results back from the server.

If implemented correctly, the communication between processes will be almost transparent in the calling function, and to a slightly lesser extent in the called function. This means that it is relatively straightforward to modify a traditional “stand-alone” application into a distributed application that uses RPC for communication between the client and server components.

The relatively high-level interface provided by RPC shields the programmer from many communications details that often complicate networked applications, such as access to the transport protocol that is in use, and differences in data representation (e.g. byte ordering) on different machines.



While the idea of Remote Procedure Call is relatively straightforward, there are a number of issues that must be addressed when implementing it. Different implementors have taken their own decisions on these issues, leading to many different versions of RPC that do not necessarily interoperate.

Locating the server

This is also known as “binding”. The client stub function must know which process (on which system) is providing the service containing the procedure to call. There are many different approaches to this. The simplest is to incorporate program knowledge of the server into the client, either by “hardwiring” it into the code, or having the user supply such information through, for example, command-line arguments. At the other extreme, the location of the server is discovered at run time by using an external “directory service”. We will discuss binding in relation to Microsoft RPC later in this chapter.

Parameter passing and data consistency

Because the parameters have to be “marshalled” into the message for transmission, and then “unmarshalled” for passing to the server function, there may be restrictions on how programs can pass arguments. For example “call by reference” behavior, where a pointer to the argument is passed to the function rather than its value, may not be supported.

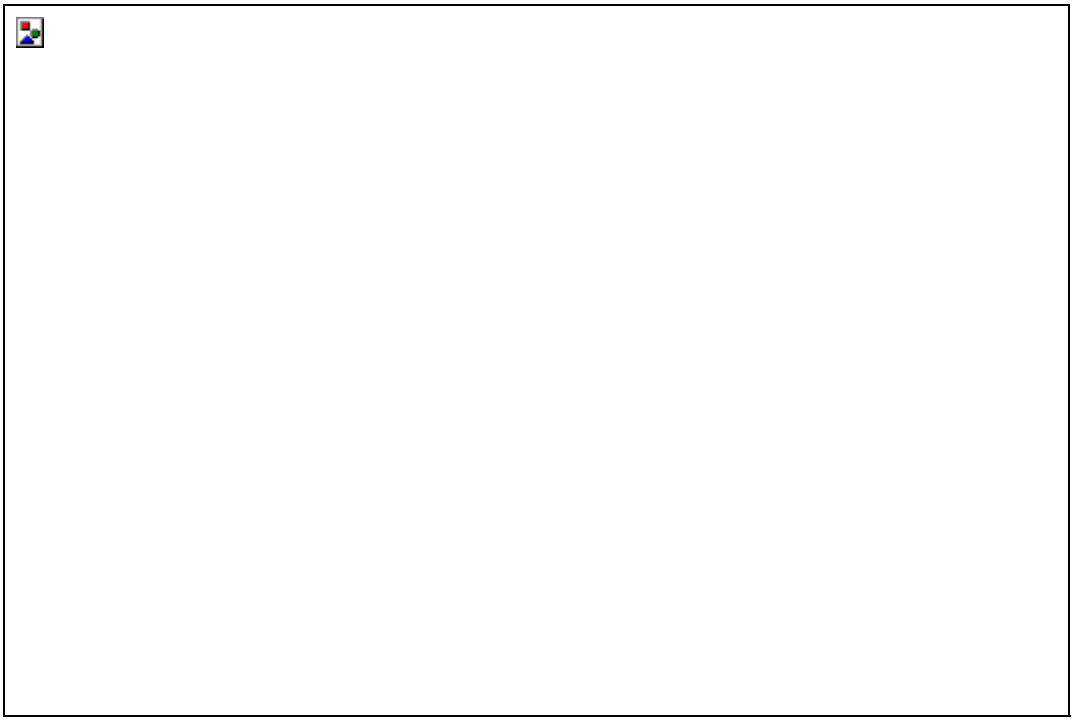
RPC implementations must also define the format of data elements as they are transmitted, so that the data is represented consistently on both client and server, even if they are different hardware.

Transport protocol

Eventually, the request and reply messages must be transmitted over some transport protocol. The choice of transport protocol may influence the interfaces to the RPC mechanism, or its behavior. For example, if an unreliable transport protocol is used, some RPC implementations will retransmit requests if they do not hear a reply within a given time period.

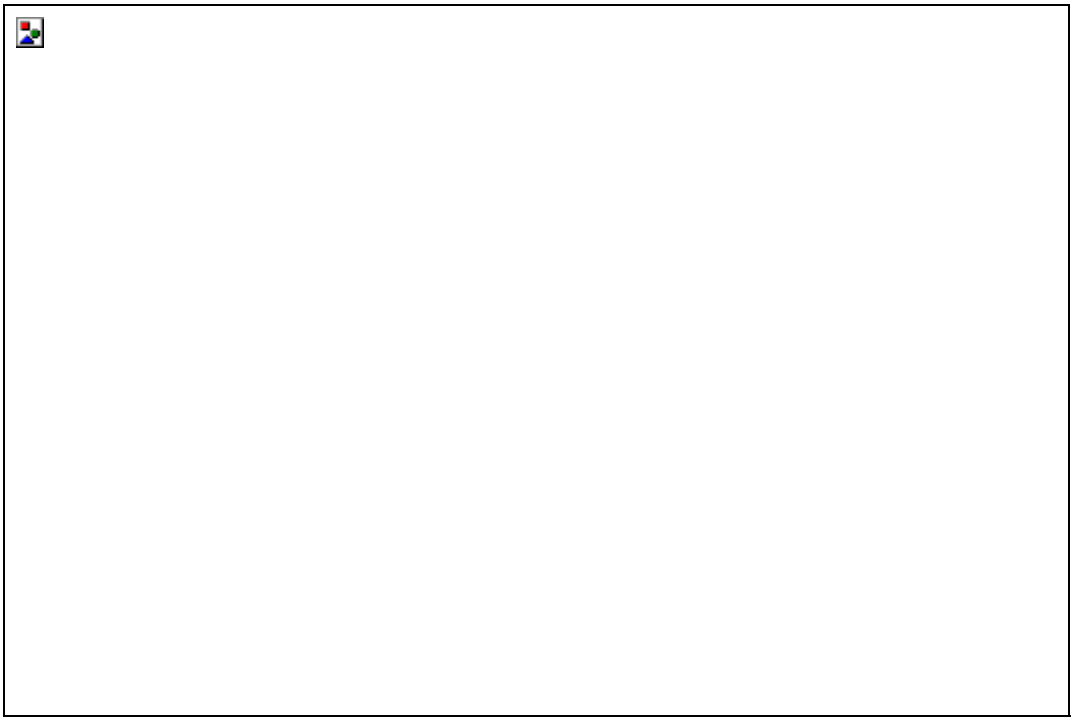
Dealing with failures

Because network environments are generally less reliable than the “local procedure call” environment, failures are more common and must be dealt with. For example, the server system may crash, before or after the function has been called. The request message or reply message may be lost (but how do you tell which?). This item often relates closely to the operation of the chosen transport protocol.



Since **RPC** is largely independent of the transport protocol used, it is normally implemented as a runtime module whose main job is to implement the higher-level interface and use the facilities of the transport protocol to transmit and receive the data. This means that different protocols can be used with only minor changes (if any) to the application.

The interface to the RPC run-time functionality is provided through the stub functions at the client and server.



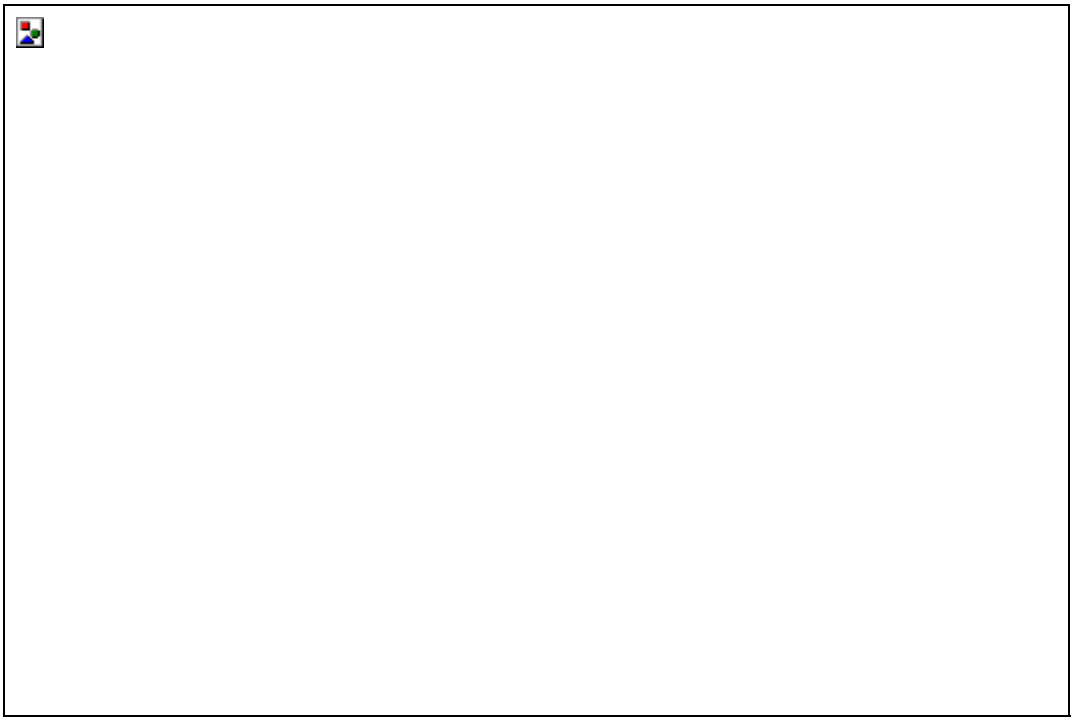
The **interface** between client and server is a central part of any RPC-based application. As in the traditional programming model, good design leads the calling function (or client) to know the called function (or server) through its arguments and result (if any). In RPC, the greater separation of calling and called function reinforces this style.

An **RPC** service is characterized by the functions that it makes available (or exports). An RPC client needs to know:

- Which functions are available on the server.
- The number and types of the arguments to each of the functions.
- The result types of the functions.

If the details of just one of the server's functions changes, this can invalidate the client/server link. (The client may pass the wrong arguments to the function.)

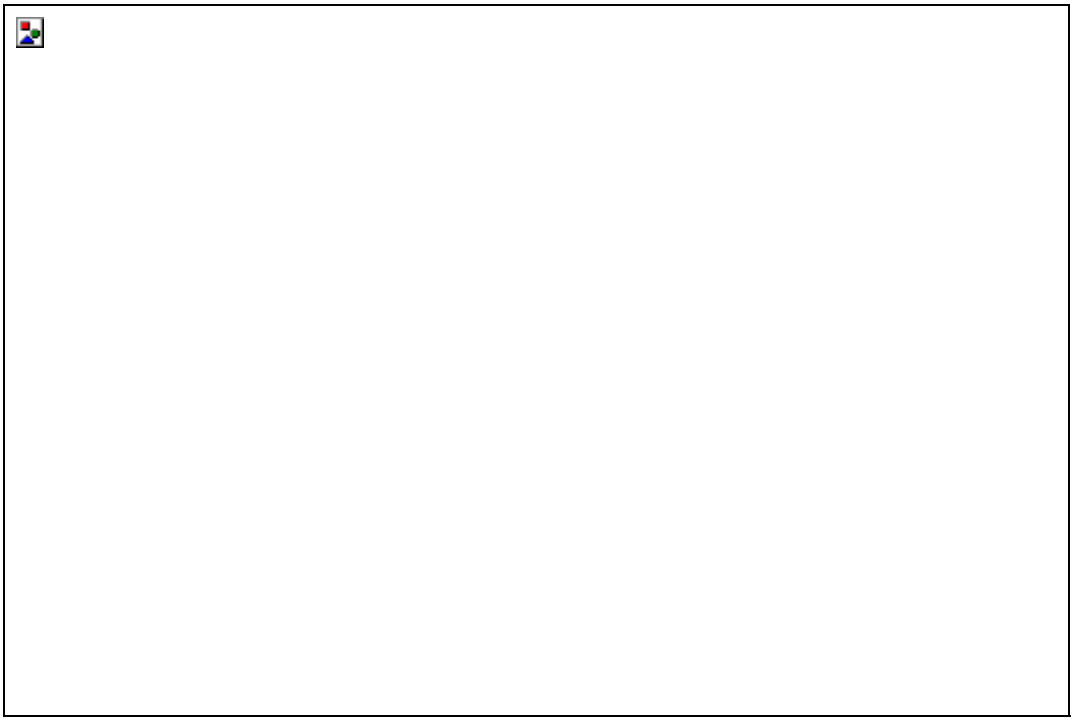
It is usual to use a representation of the server's interface as the primary means of a client identifying its corresponding server. In ONC RPC, this is a combination of two numbers (program number, version number). In DCE and Microsoft RPC this is provided by the Universal Unique Identifier, or UUID. (Microsoft NT refers to the UUID as the Globally Unique Identifier or GUID.)



Most **RPC** toolkits include facilities to simplify the production of the interface portion of an RPC application. Typically, the interface is specified in a high-level definition language that is then translated into the target development language, using a code generation utility.

The code generator (often called the compiler) normally produces stub functions for both client and server, and a list of common definitions and declarations for inclusion in the client and server programs.

Each of the main RPC implementations has its own language for specifying interfaces, and its own rules about what needs to be defined in the specification file. The target language for the code generator is normally C, although the code produced can normally be processed by a C++ compiler or incorporated into C++ programs.

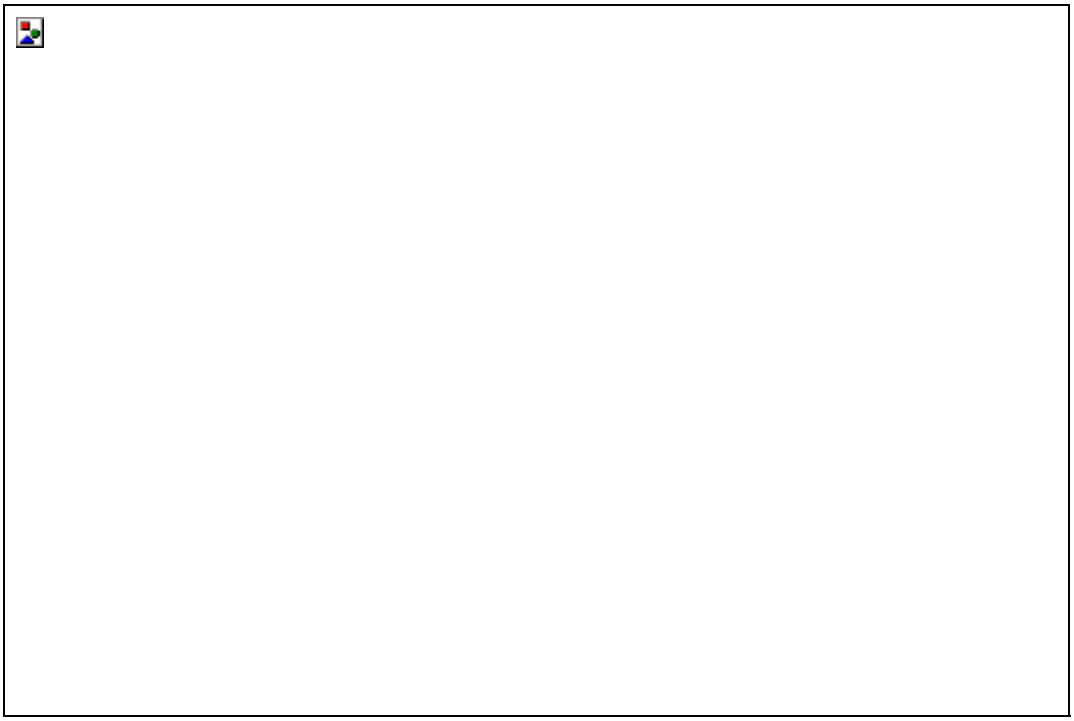


With Microsoft **RPC**, you specify the **RPC** interface in a language known as **MIDL** (Microsoft Interface Definition Language). MIDL is Microsoft's implementation of the DCE interface specification language IDL. The two languages are largely equivalent to each other, but there are a small number of differences. For example, IDL does not support the notion of the named pipe transport, and the DCE feature known as "pipes" (used for bulk data transfer) is not supported in MIDL.

The interface specification is placed in a file with suffix "*.idl*". The MIDL compiler then processes the file and generates the stubs and a header file for use in the application.

You must provide a further file as input to the MIDL compiler. This is the *Application Configuration File*, and it is used to specify various additional options as to how the interface will operate. More about this later.

The compiler may also produce auxiliary files for client and server. These are used for marshalling data and dealing with complex data types.



The first stage in building the application is to define the interface. This consists of the two files that will form input to the MIDL compiler: the interface definition file and application configuration file.

Interface Definition File

This file contains the definitions of the remote procedure(s) exported by the server, and details of the arguments and results. There are three sections to the file: the Interface Header, the Interface Name and the Interface Body.

The Interface Header specifies the global attributes of the interface. In the example, these are the UUID (GUID) and the version of the interface.

The UUID is a long string, into which is encoded information that uniquely identifies this interface anywhere on the network. You generate a UUID automatically, using the utility **uuidgen.**

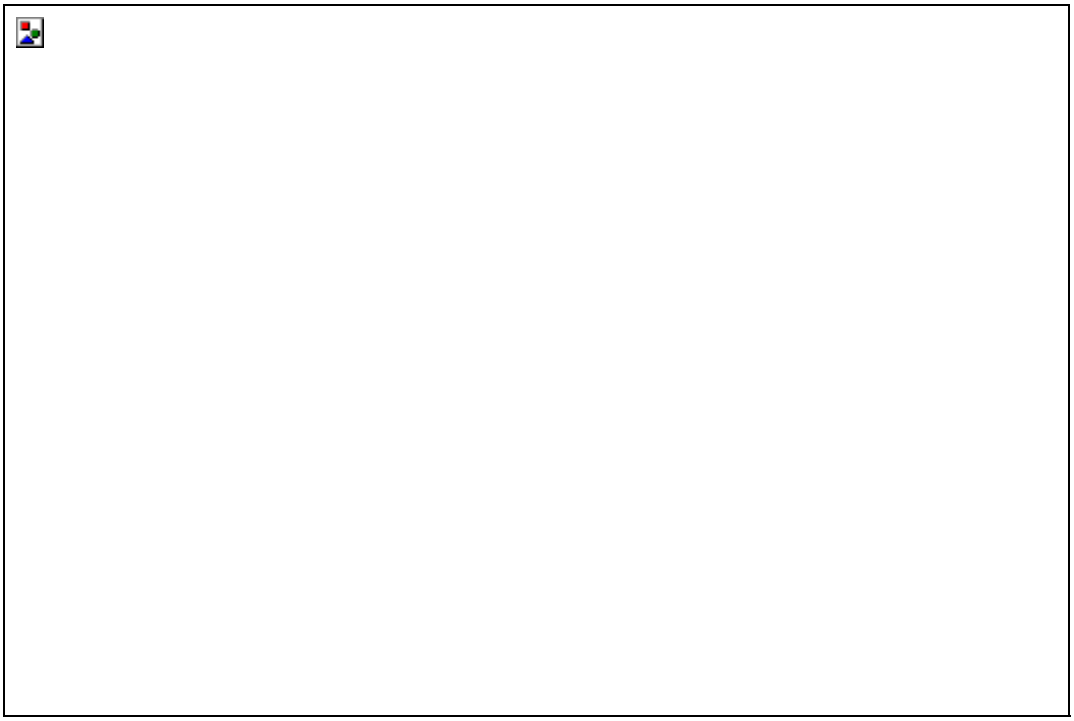
The version number is included after the **version** keyword. The number can be any single decimal integer or pair of decimal integers in the range 0 to 65535.

The interface body contains “prototypes” of the functions that are exported by the server. Data-type definitions used by the interface functions may also be defined here. This example is very simple; the server exports the function **RPrintProc** to do the main work, and a second function **Shutdown** to allow the server itself to be closed down via a Remote Procedure Call from a client.

Neither function returns a value, so they are declared as **void. RPrintProc** takes a single argument, a string **pszString.** The argument is qualified by one or more attributes, specified inside square brackets. In the example, the argument is marked as **in,** to say that it is being passed from client to server only, and string, to mark that it is a special case of a character array. By telling the MIDL compiler how arguments are going to be used, it is possible to optimize the argument-handling code that is generated.

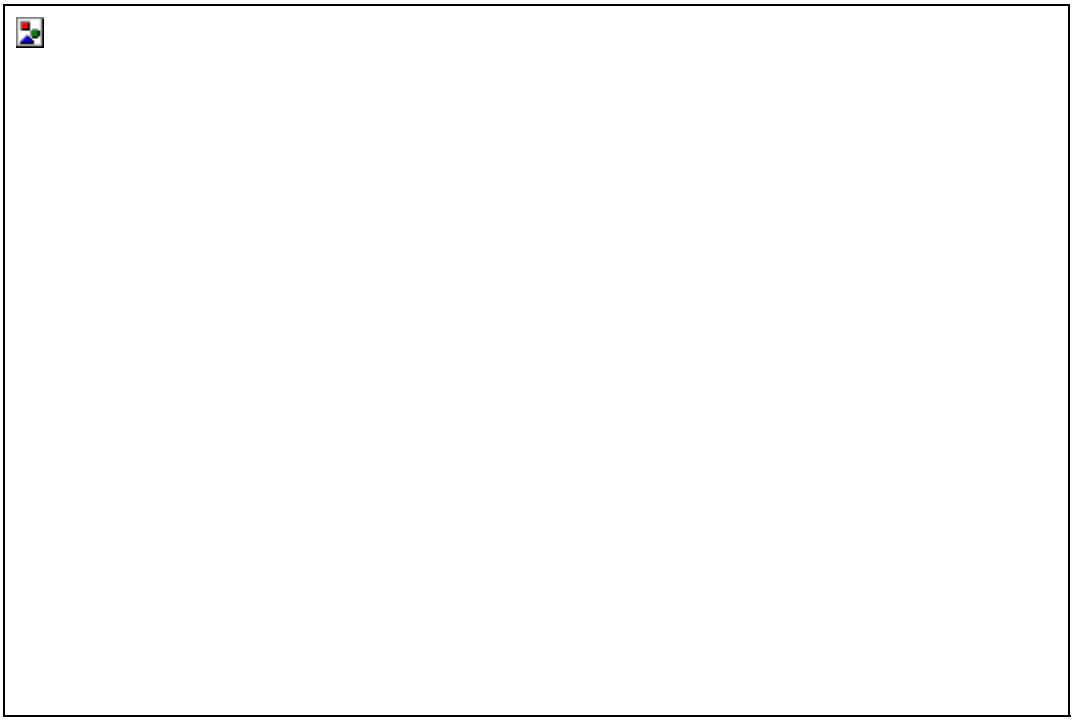
Application Configuration File

The job of this file is to specify attributes of the interface not directly connected with data transmission. The format is similar to the Interface Definition File. This example is very simple. Here we specify that the connection between client and server, known as the **binding,** is to be managed automatically by code generated in the stub files. We will look at alternative approaches to binding later in the chapter.
The Interface Name is **rprint.** It is common practice to use this name as the base name of all the source files.



Because we have chosen automatic binding in this example, the client main program is extremely simple. The stub files generated by the MIDL compiler and the RPC run-time routines will automatically make contact with the correct server, based on the information that was included in the Interface Definition File. All that is required in the client program is to call the remote function. Here, after the client has called **RPrintProc,** it arranges to shut down the server by calling **Shutdown.**

In the example, the calls to the remote procedures are wrapped inside the standard RPC exception-handling mechanism, to simplify the gathering of any failure diagnostics. This is just a wrapper around the SEH provided by the system.



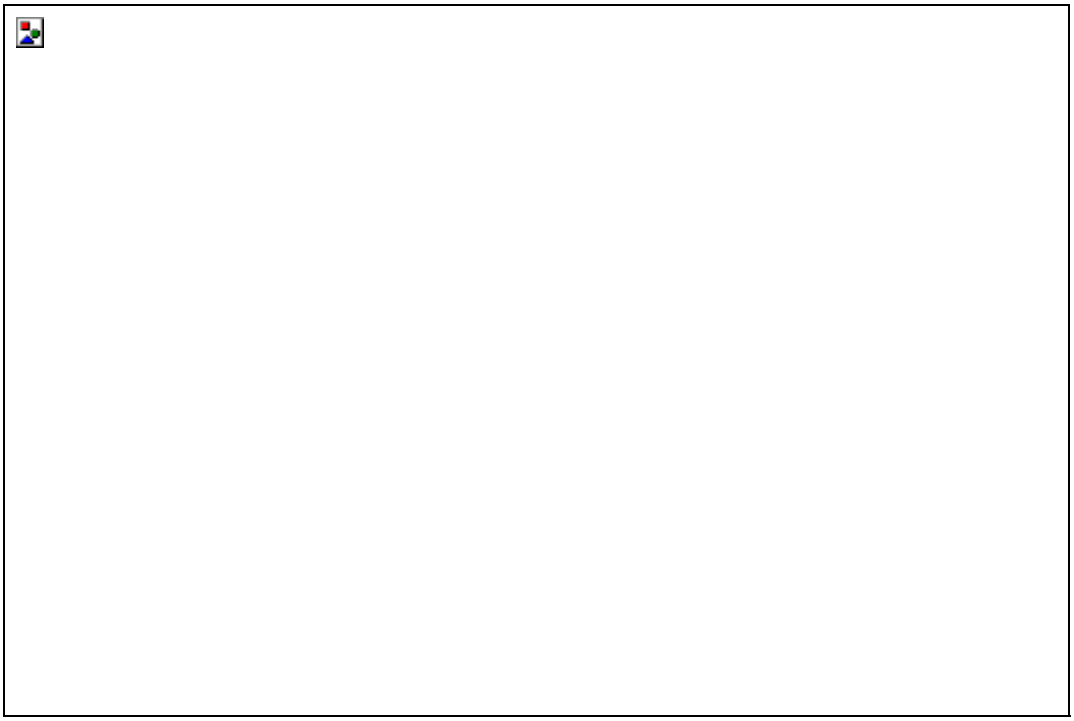
The server component of an RPC-based application is more complex than the client. There are two main components to a server program:

Remote Procedure Implementation

You must include the code that implements the functions that will be called from the client. This is the easy part; the function can be coded as if it were a local function, so long as the interface properties (i.e. argument number and types and result) are maintained.

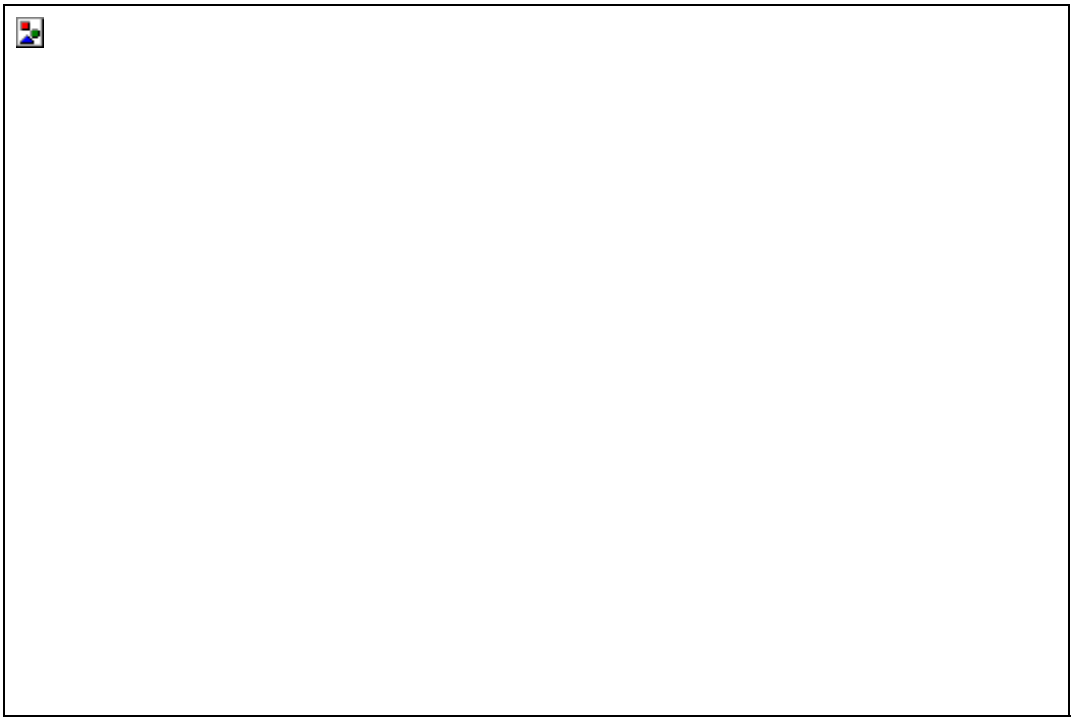
Server Initialization

Most of the hard work in a server is involved with initialization, when the server makes known its existence and the services it provides. Different components of the RPC system need to be informed about the server, so that the RPC run-time functions on the client can locate and use its services when required.



The implementation of the function(s) that are available through this server must be defined. Here, the file **serverp.c** contains the definitions of the two functions.

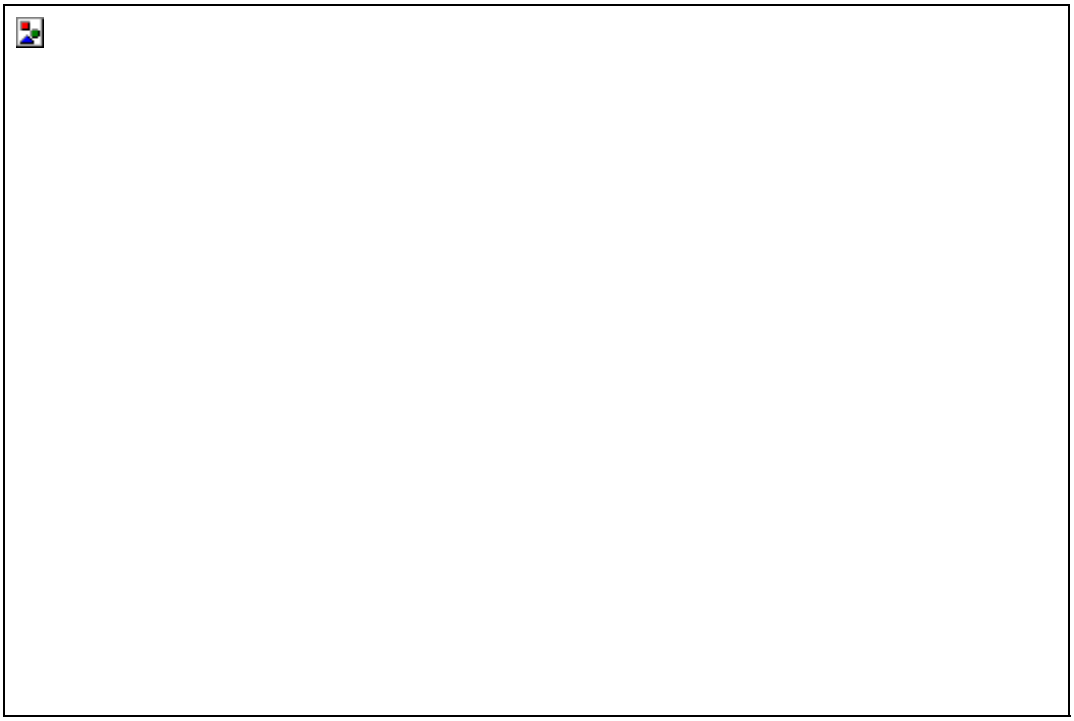
The functions should be written as if they were normal local functions.



This is the main program for the server. Its job is to start a new thread, referred to by **ahListen, in** which the server will operate. The function **vListenRPrint** sets up the necessary information for the server, and begins listening for requests from clients.

When the client calls the **Shutdown** function, this causes the event **hEventQuit** to be posted to the process (see the previous slide). The main thread will be waiting for this event, and upon receipt will tell the server to stop listening for more requests **(**`RpcMgmtStopServerListening()`**).**

This leads to the termination of the server thread abListen, and thus the main program will itself terminate.



The function **vListenRPrint** sets up the server so that RPCs from clients can be accepted.

First, binding information for the server must be established. The *binding* is the mechanism by which the client attaches to the server before issuing calls. A server can choose to make its service available on a specific protocol (or protocol sequence as DCE calls it), or to use all available protocols. In the example, the server chooses the latter option through the function `RpcServerUseAllProtseqs().` The first argument here specifies how many calls the server will handle concurrently. Choose a value of your own, or use RPC_C_PROTSEQMAXREQSDEFAULT for the default. The second argument is NT specific, a security descriptor that can be left NULL for now.

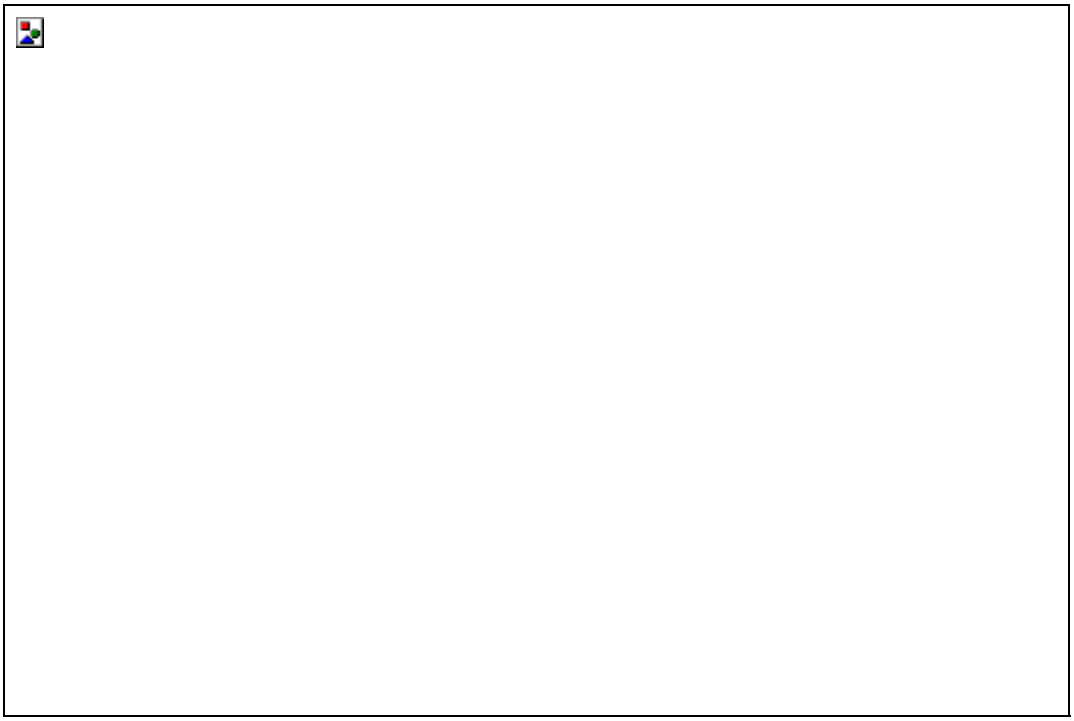
(All the server functions return a status value, of type **RPC** STATUS. If this is not zero, an error has occurred.)

Next, the interface must be registered with the RPC run-time routines on the server system. This is done with the function `RpcServerRegisterIf().` The first argument to this function is an interface handle", whose name is generated in the MIDL compilation process, and defined in the header file rpnnt.h. The remaining arguments are for more specialised use and can be left as NULL for now.

After registration of the interface and the protocol sequences, the binding information must be notified to the name service, so that the client can find out about the server when it wants. Before this can be done, we must obtain that information about this service. (There are other ways of setting up a server, so that the binding information is built explicitly, removing the need for this step.)

The function `RpcServerlnqBindings()` obtains a list of bindings for this server; the name for this is the *bindings vector.* The bindings vector is a set of handles, each of which references a particular binding.

Once the bindings vector has been obtained, the server must register with the 'end-point mapper' using the function `RpcEpRegister().` This places the sever-address information into the local end-point database. An endpoint would be a pipe name if the transport was named pipes, or a port number if the transport was sockets. The call to `RpcEpRegister()` is needed because we used `RpcServerUseAllProtseqs().`



Once the bindings vector has been obtained, the server can “advertise” itself in the name service that clients will use to locate servers. This is done using the function `RpcNsBindingExport().`

The first two arguments, nNameSyntaxType and pszEntryName describe the name by which the service will be identified in the name-service database. nNameSyntaxType specifies how the string held in pszEntryName is to be interpreted. For the default behavior, set this to RPC_C_NS_SYNTAX_DEFAULT.

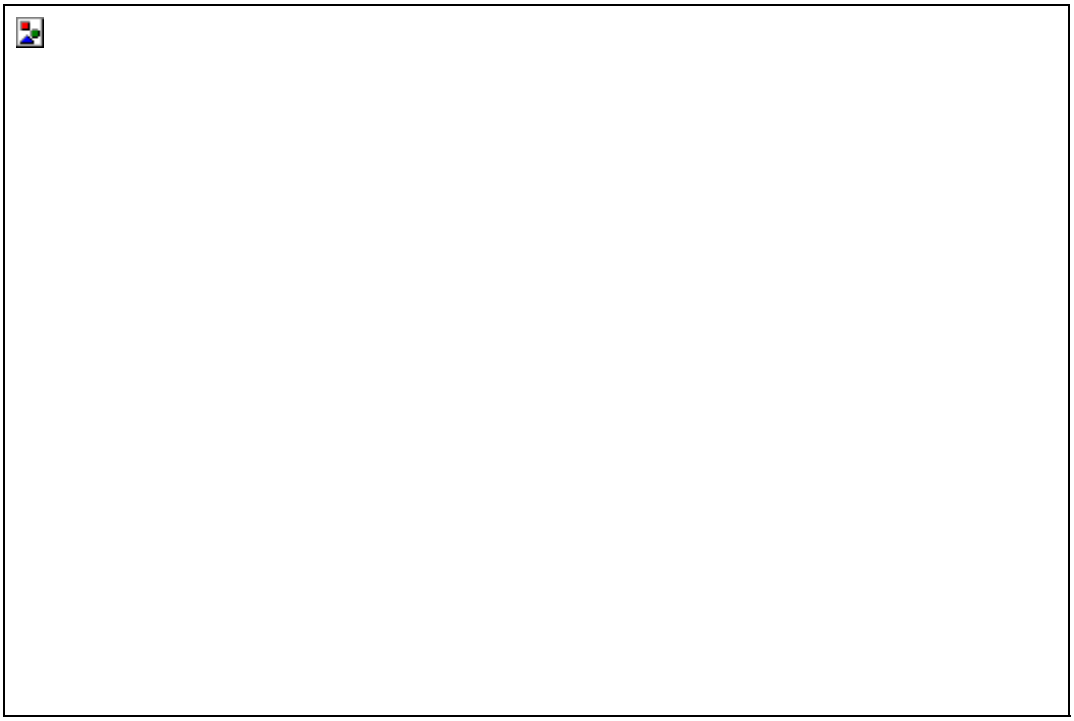
The third argument is the interface handle, as generated by MIDL and found in the stub. The fourth argument is the bindings vector, obtained from the call to `RpcServerInqBindings().` The fifth is for more advanced use, and can be left NULL.

Now the service is available and the server can wait for calls from clients. The function `RpcServerListen()` starts the ability of the server to accept calls. There are three arguments. **nMinCalls** specifies a minimum number of threads that will be used to deal with calls. **nMaxCalls** is the maximum number of calls that can be handled by the server simultaneously. This is a guide, not an absolute limit. Use the RPC C LISTEN MAX CALLS DEFAULT for a default value.

Normal behavior (as defined in DCE) is that `RpcServerListen()` will not return until another thread calls the function `RpcMgmtStopServerListening()`. In NT, this is slightly different. The third parameter to `RpcServerListen()`, **nDontWait,** is used to indicate whether the function is to effectively block the calling thread, or to return after the basic set-up operations have been completed giving a non-blocking mode of operation. If **nDontWait** is zero, the default (blocking) DCE behaviour is achieved. If not zero, the function returns as soon as the setup specified by the API has been completed.

Now the server will handle calls from client processes. Remember that in this example, the client can shut down the server through the `Shutdown()` function. This causes the function `RpcMgmtStopServerListening()` to be called, thus causing `RpcServerListen()` to return.

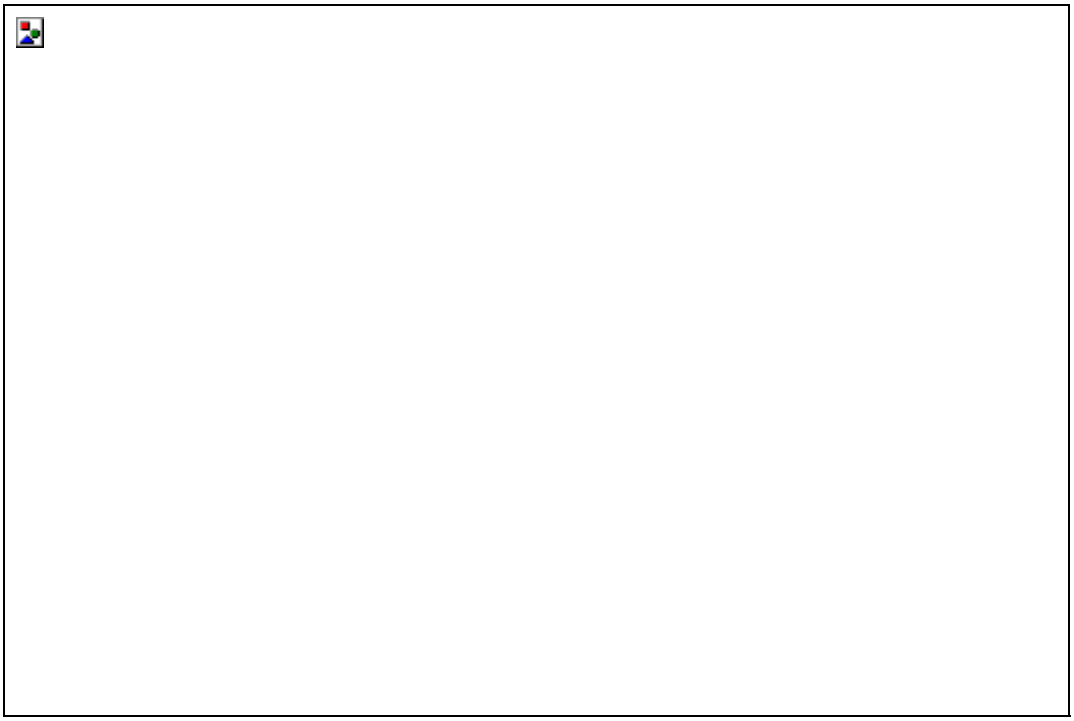
The calls following `RpcServerListen()` are used to remove the server from the end-point database and the name service, and to free up any bindings vectors.



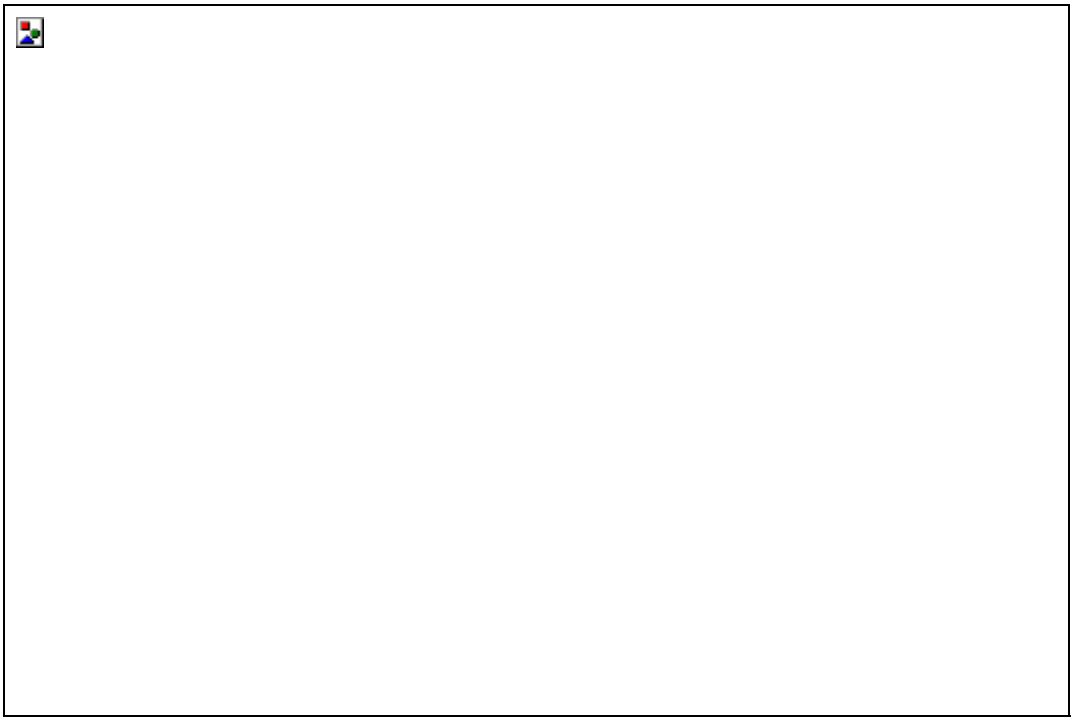
To complete the server, it is necessary to provide definitions of two functions that are called from within the server stubs, to handle dynamic memory management for parameters to the remote functions.

midi_user_allocate and **midi_user_free** are used to allocate memory and to free memory, to store parameters within the server stub, before they are passed to the appropriate function.

You can define your own heap-management strategy if you wish, and use these functions to implement it. However it is more common, certainly with simpler applications, to simply call `malloc()` and `free()` to get the default heap management from the standard library.



Because of the many components in the application, it is common to use a builder utility such as **nmake** to control compilation and linking. This slide shows an excerpt from a Makefile that can be used here.



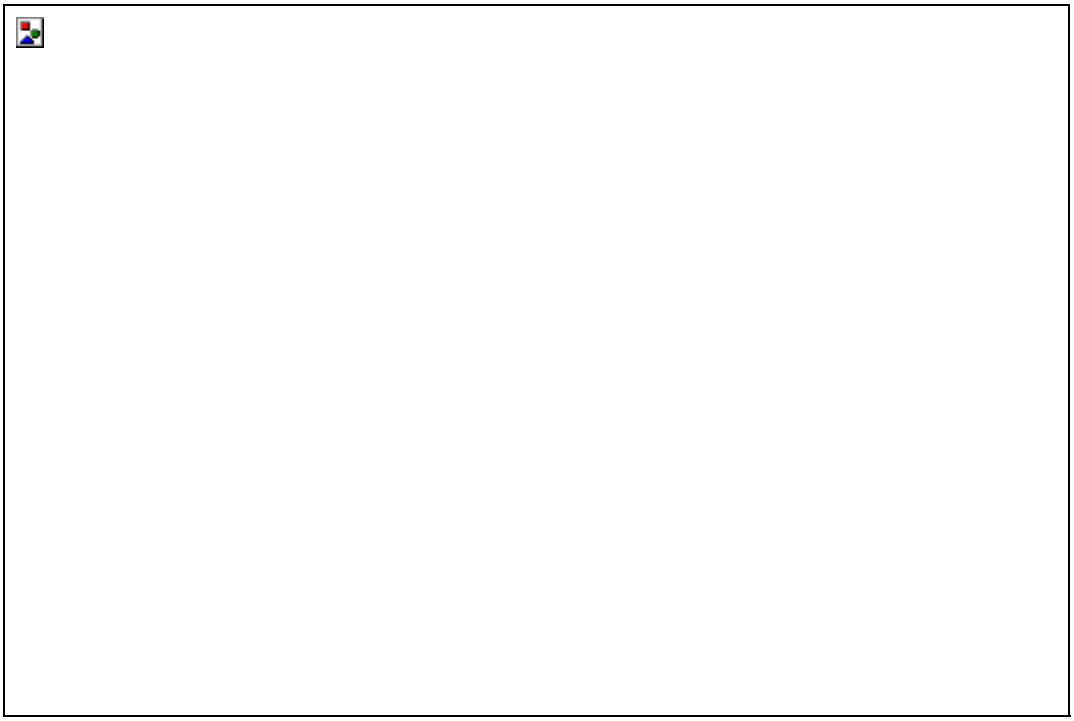
When the server is written, it sets up its interface and registers with the **RPC** run time on the local system, usually with a name service.

A client wishing to use this service must be able to find it on the network. The mechanism of locating a server is known as binding. Microsoft has implemented the same highly flexible approach to binding that is part of DCE.

Broadly speaking, a client can choose to use *automatic* binding, or to manage binding itself. In the example shown earlier in this chapter, automatic binding was used. This was indicated principally in the ACE for the interface, and affected the client stub function that was generated by MIDL.

Automatic binding is the simplest mechanism to use from the client. No special RPC run-time functions need to be called, as the stub will take care of locating the required server.

Application-managed binding requires more work at the client, but does not necessarily require a server to be different from one that uses automatic binding. It is generally more flexible than automatic binding.



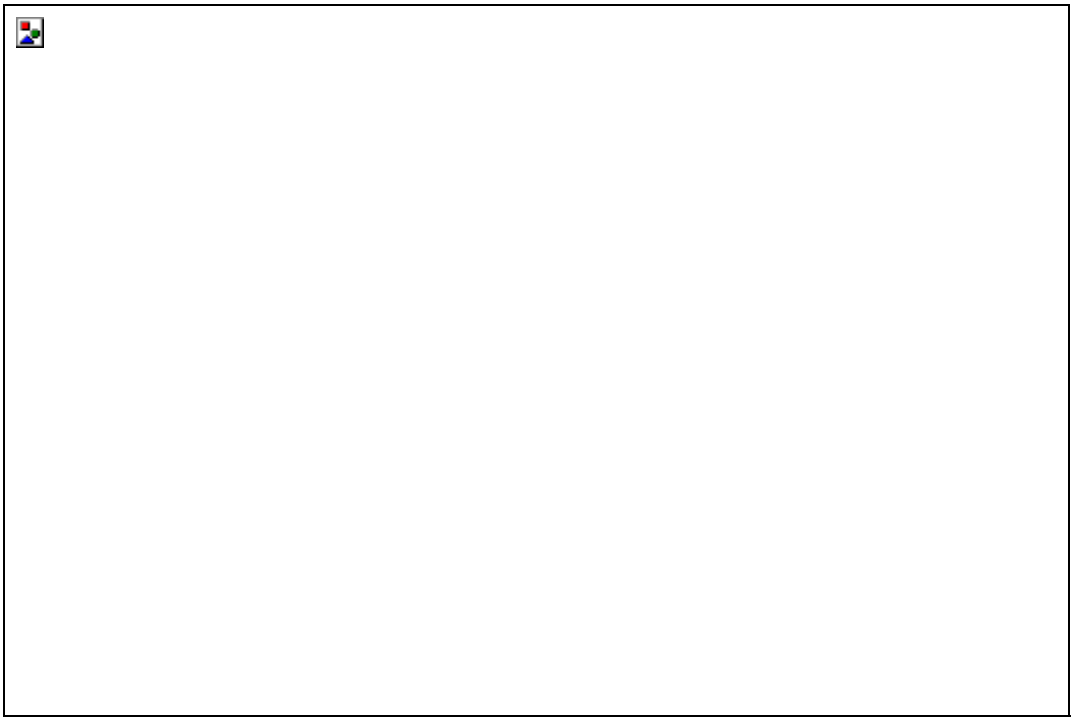
A binding handle is the mechanism by which a client specifies the server for its RPC. With automatic binding, the handles are set up inside the stub function and hidden from the client. The server stub deals with the binding handle as well.

With **application-managed** binding, the handle needs to be set up at the client, and may be required in the definition of the server functions. The handle may be of a user-defined type or, more commonly, of a standard "primitive" handle type.

With **explicit** binding, the handle must be attached to every call of a remote procedure. The client stub passes the handle to the server, and it must be included in the definitions of each server function.

With **implicit** binding, the client must still set up the handle, but here it is stored in a global variable on the client. The client stub attaches the handle to the request that is made of the server function. This means that the server function does not have to deal with the handle, in other words the server can be the same as that used for automatic binding.

For certain more advanced applications, a special form of handle is available, called a **context handle.** This is similar to a binding handle but it allows *state* information to be maintained between client and server. This can be important for certain classes of application, such as file servers, but introducing state maintenance into a distributed application will complicate it considerably.



A binding handle contains a number of elements that describe how a client contacts a server:

Protocol Sequence

This is the transport protocol that is to be used. Sequences are specified by strings. The currently supported protocol sequences in NT are:

`ncacn_ip_tcp`	TCP/IP
`ncacn_nb_tcp`	NETBIOS over TCP/IP
`ncacn_nb_nb`	NETBIOS over NETBEUI
`ncacn_np`	Named Pipes
`ncalrpc`	Local RPC (i.e. call to a server on the local system)

Notice that all these represent connection-oriented protocols. Currently, Microsoft RPC is not supported over connectionless transport protocols. If you are writing applications to interoperate with DCE networks, the only protocol sequence common to the two is TCP/IP (ncacn_jp_tcp).

Network Address

This component is used by the client to direct the request to the server. The exact format of this will depend on the protocol sequence chosen. For example, if you use ncacn_ip_tcp (TCP/IP), the network address must be a host name or dotted decimal IP address.

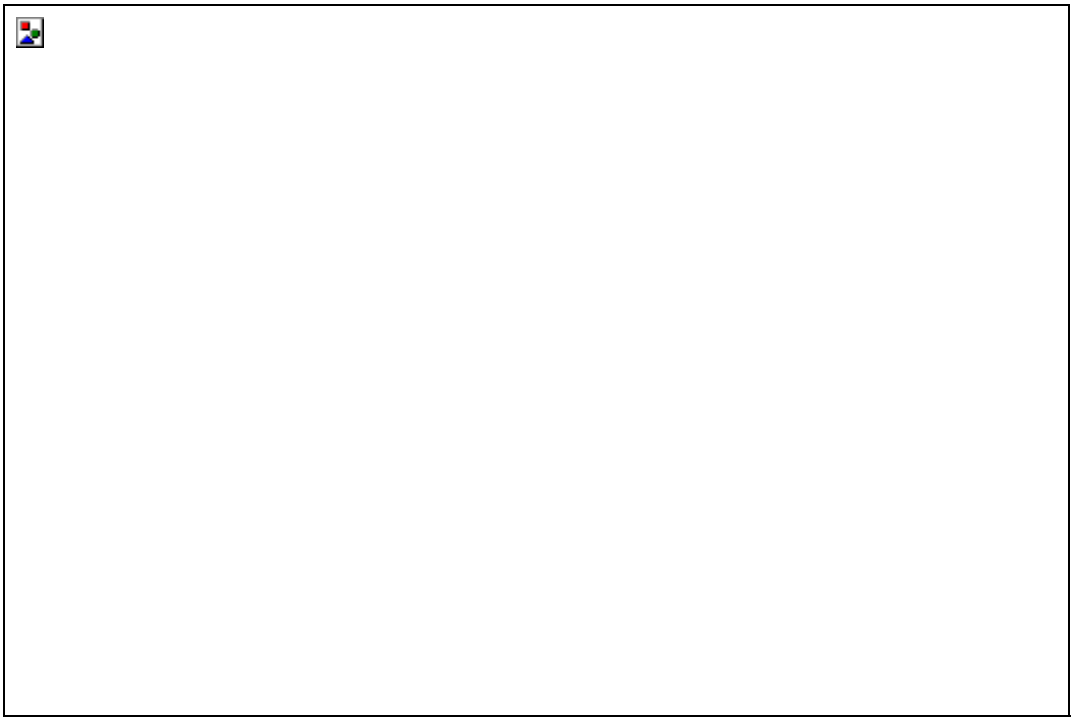
Endpoint

This is the mechanism for locating a particular server process on a given system. Any given server host may have several servers operating concurrently, so each has to be identifiable individually.

Once again, the format of this element of the handle depends on the protocol sequence. For ncacn_np (named pipes), it should be the name of the pipe (e.g. \pipe\rprintsrv). For TCP/IP it should be a port number.

Name and Name Syntax

The service should be given a name, so that it can be registered and located using a name service. These elements of the binding handle contain the name as a string and an indication of the name syntax, so that the run-time functions know how to interpret the name string.



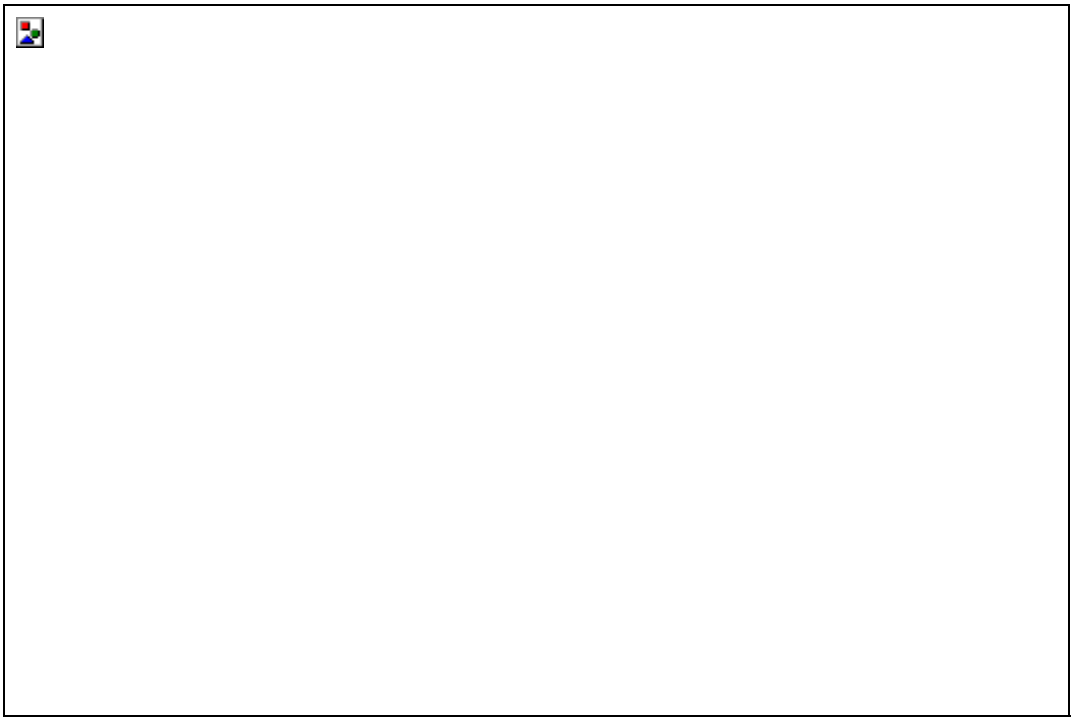
There are two ways a client can obtain binding information for a service. The first involves the client having the necessary details either passed into it from command-line arguments or some other configuration method (known as the *String* binding method) and the other involves interrogating a *Name Service* for the details, based on the service name.

The String binding method is not encouraged; it requires details to be hard coded into the application, or the user must supply those details when starting the server. Hard coding details into the application remove a level of flexibility. Requiring the user to supply the information is contrary to one of the goals of distributed applications, as described in the DCE; that the user should not be aware of the networking aspects of the applications that are used (such as machine locations).

The preferred method of obtaining binding information is to use the Name Service. DCE and Microsoft supply a name service as part of the RPC operation. The two are different, but future releases of Microsoft RPC will interoperate with the DCE Name Service, known as the Cell Directory Service (CDS).

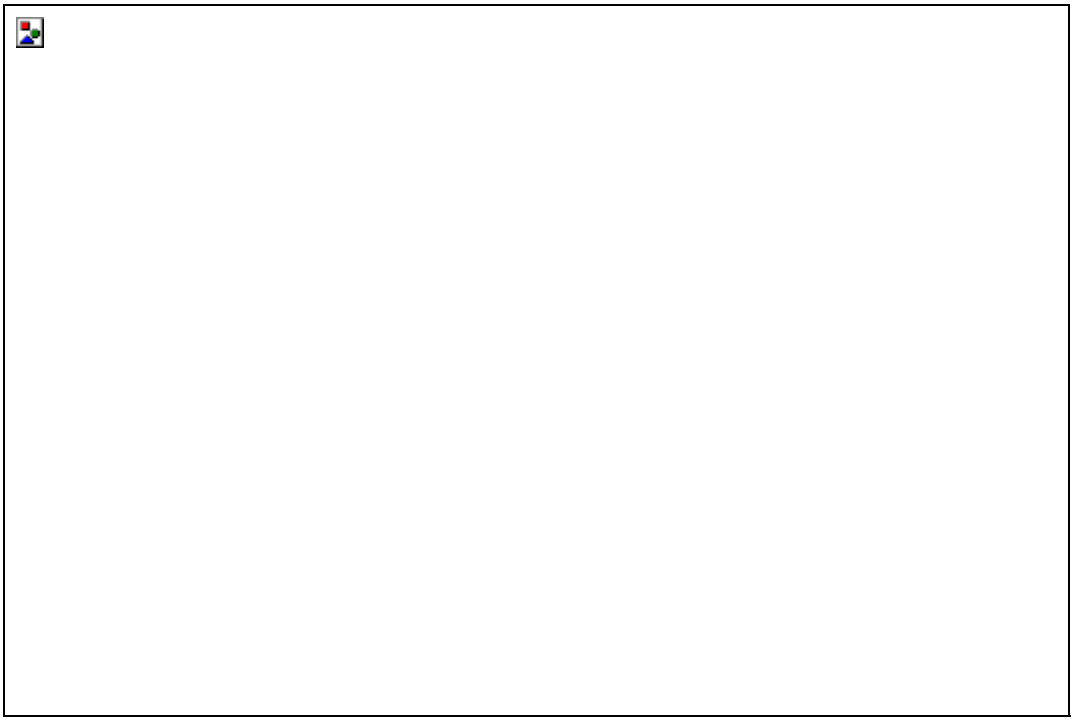
Remember from the example shown earlier in the chapter, part of the server initialization involves calling the function `RpcNsBindingExport()`. The job of this function is to register the binding details (host name, protocol sequence(s), etc.) of the service with the name service, which may be on a separate system. The service is identified by its *name,* specified in the function call. To obtain the binding details, the client asks the name service for the details corresponding to the service name. The server responds with the necessary information and the client can then complete its binding to the server (either implicitly or explicitly).

With automatic binding, as shown in the example earlier, the client stab function carries out these steps. Here, you must incorporate them into the client main program.



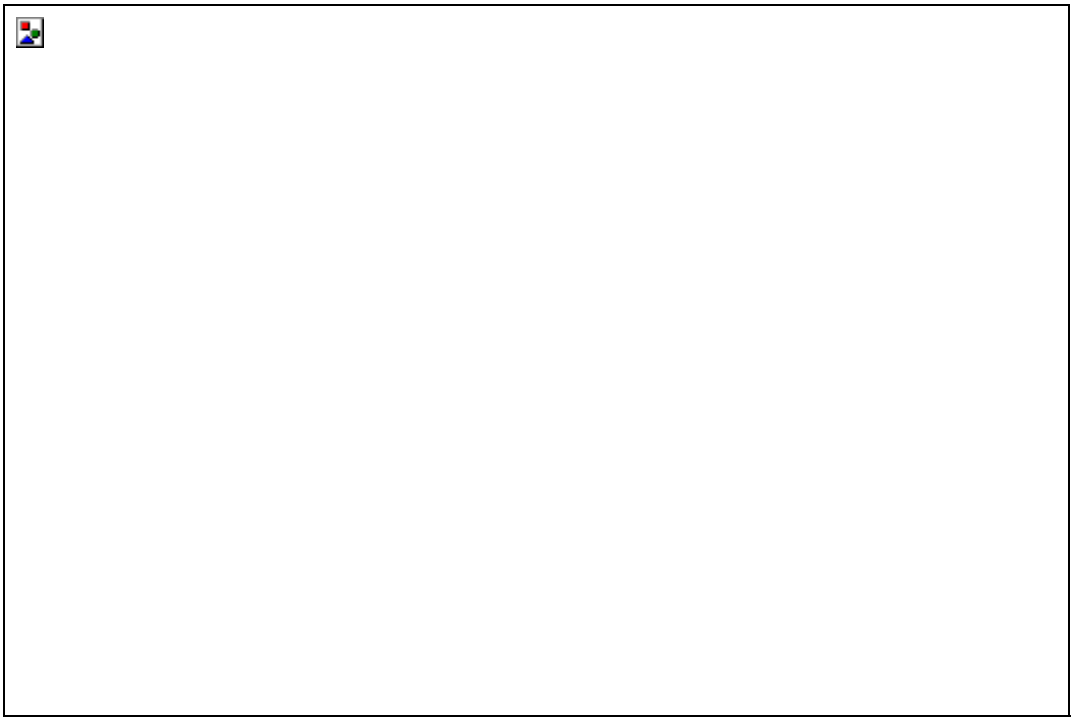
The following slides show how the example from earlier in the chapter can be modified to use the explicit binding approach, and to obtain binding information from the Name Service.

Because we are using explicit binding, the *Interface Definition File* must be altered slightly, so that the remote procedure definitions include the binding handle in their arguments. The Application Configuration File must also change, as it is here that explicit binding is specified.



The changes required at the server are small. Indeed, if implicit binding was chosen, then no changes at all would be necessary. Here, the binding handles are included in the argument list of the service procedures.

The main advantage of using explicit binding is that the server procedure can find out binding information about the client who has made the call. This can be useful when there are several clients interacting with a single server.



Most of the changes to the example as shown earlier occur in the client program. It is not possible to simply start the program and make the remote procedure calls. The client must establish the binding before calling the remote procedures.

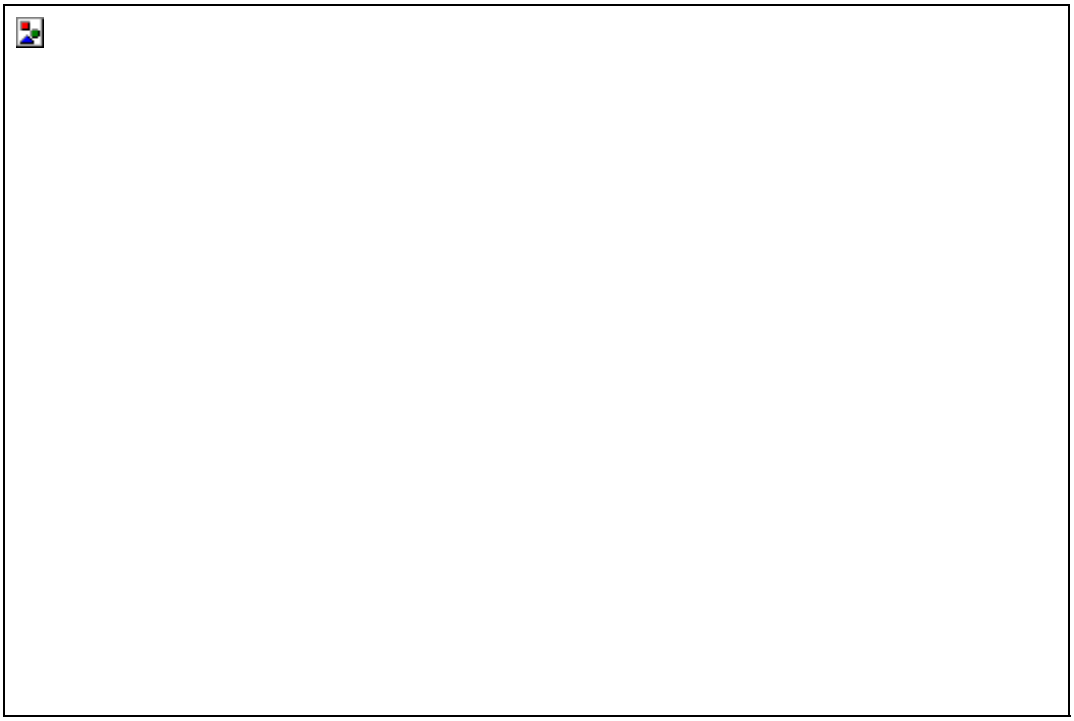
In this particular example, we have chosen to use the name service to obtain the binding information. The server already registers the service with the name server, under the name "*/.:/rprint*". This is the name that the client uses to look up the service.

The client must first initialize for the search. This is done by calling `RpcNsBindingImportBegin()`. This function creates an Import Context, used in calls to the function `RpcNsBindingImportNext()` that follow. You must specify the name details of the service and the client's interface handle (generated by MIDL). The name service import context is returned in the argument hnsNameService.

Now the client can repeatedly call RpcNsBindinglmportNext until the required information is retrieved. This function returns a server binding handle compatible with the requests from the client.
Repeated calls return other compatible servers, until there is none left (when the function returns RPC_S_NO_MORE_BINDINGS).

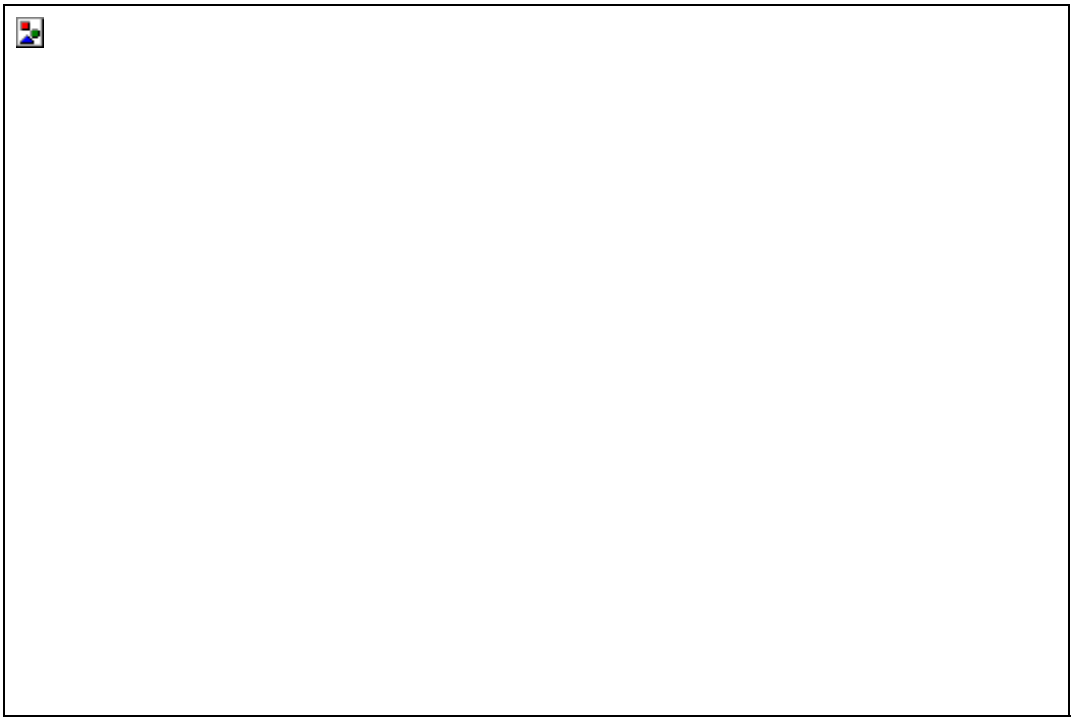
In the example, we are happy to use the first binding handle that is returned.

You can also use functions `RpcNsBindingLookupBegin()` and `RpcNsBindingLookupNext()` instead of the **...Import...** functions. These return vectors of binding handles, all of which are compatible with the requested server. You can then use your own method of selection amongst them.



We have chosen explicit binding, so the binding handle obtained from the name service must be passed with the remote procedure calls. Otherwise, this portion of the client is the same as in the earlier example.

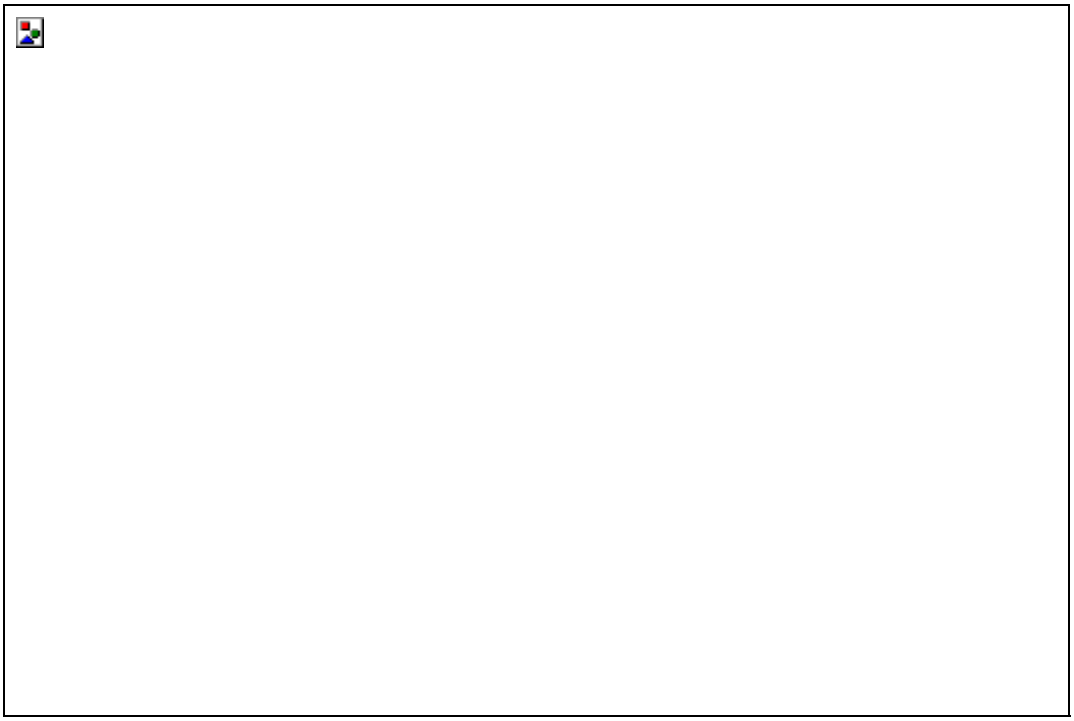
When the client is finished, the function `RpcNsBindinglmportDone()` is called, to free the name service handle that was being used.



Call semantics define what will happen if an **RPC** call fails. By default remote procedures are defined as non-idempotent. This means that they are assumed to modify some state information or have some other effect that varies with the number of calls. If a non— idempotent remote procedure fails, it will not be re-executed. This gives at-most-once semantics; the remote procedure is executed once or not at all. If a remote procedure is marked as idempotent then if a call fails to complete then the remote procedure will be re-executed automatically. This gives at-least-once semantics; the remote procedure is executed one or more times.

maybe is an altenative call semantic. This can be used on remote procedures where it does not matter if they do not execute successfully every time. Remote procedure marked as maybe cannot contain output parameters and is always idempotent.

Sometimes a server application may need to execute some code on the client. This is achieved by use of the callback modifier. An interface may contain static callback functions. These callback functions are callable from within any remote procedure in the interface. Callbacks cannot contain handles as parameters. Because they execute within the context of the remote call, the binding handle information for the client and server is already known, so a handle is unnecessary and illegal. Callbacks may be nested.

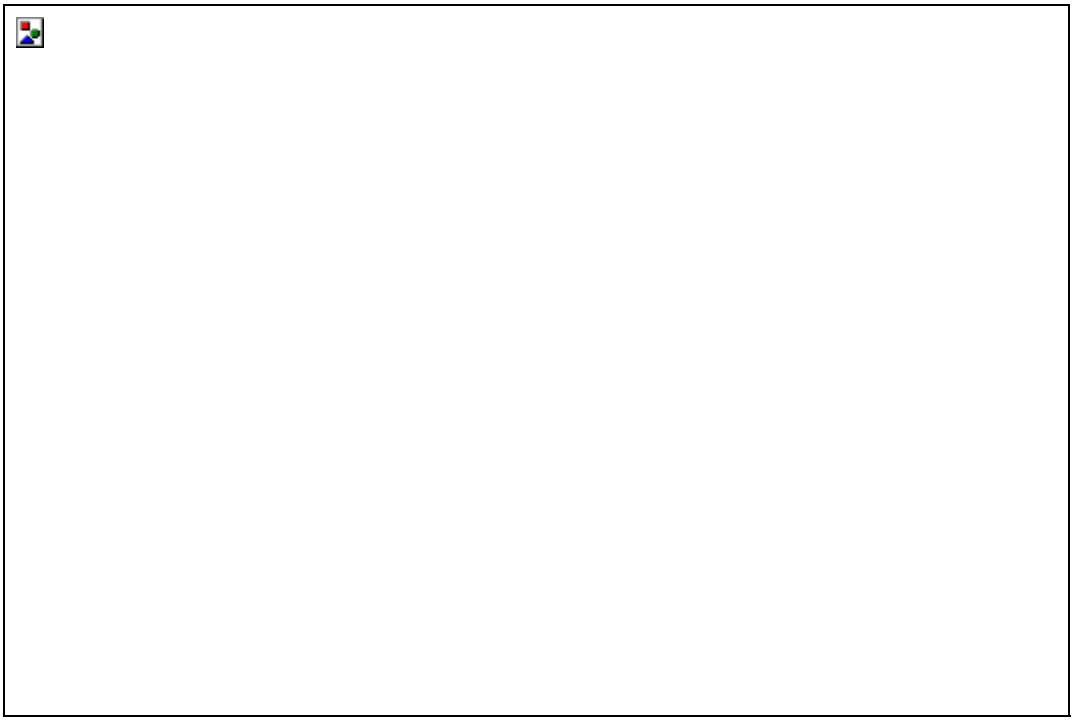


One of the basic problems in writing code for a heterogeneous networked environment is the difference in data representations among different systems. This is the problem that should be dealt with by protocols at layer 6 of the ISO 7 layer model for Open Systems (the Presentation Layer).

For example, when communicating between older PCs and new RISC workstations, the basic sizes of some types will be different (e.g. *int* may be a 16-bit or 32-bit type; C does not specify which). There is an additional problem that can occur when communicating between systems of a different hardware architecture. The order in which the hardware interprets the bytes of an integer-based value differs on certain systems. Some interpret the byte at the lowest (byte) address as being the most significant byte of the value (called *Big Endian* byte ordering) and others interpret the byte at the lowest address to be the least significant *(Little Endian* byte ordering). There may also be data alignment rules on particular hardware.

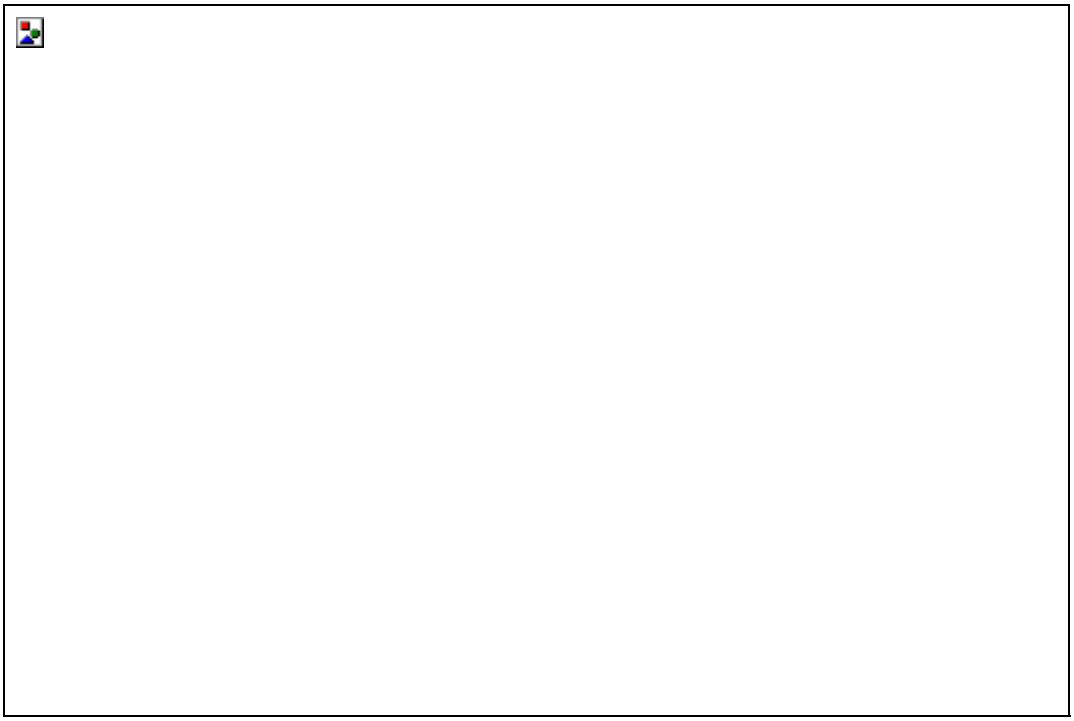
Dealing with these problems introduces additional complexity into your applications. However, **RPC** includes facilities for overcoming them, at little or no cost inside the application.

MIDL defines a set of well-defined (sized) data types that are used inside the interface-definition files. As arguments and results are marshalled for transmission inside the stubs, the base types are converted into a system-independent data format called NDR, or Network Data Representation. They are converted back at the receiving system. Further, you can specify your own transmission format using the **transmit_as** attribute in the interface definition file.



This table lists the base data types that are recognized by IDL (for DCE) and MDL (for Microsoft RPC). Use these types for declaring function arguments and results. The sizes are guaranteed to be the same on all implementations. Notice that there is no type **int,** for example.

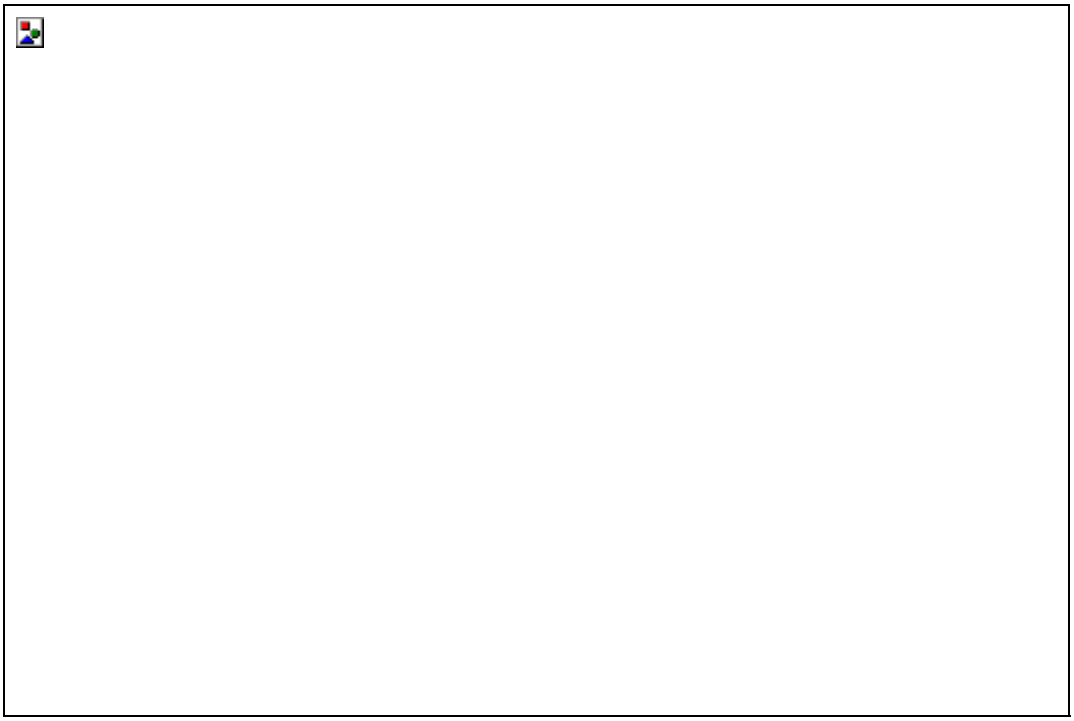
Many of the types can be declared as signed or unsigned. This is useful as it overcomes differences in rules for default signing in different compilers.



It is also possible to specify compound data types, largely similar to those of C, in the MTDL file. You can introduce new types using *typedef,* as in C.

Arrays of fixed and variable length are supported, as are structures.

A special type of union, known as a ***discriminated union*** is available. This is like the Pascal variant record construct, or the C union with an additional element known as the ***discriminant,*** whose purpose is to indicate which of the arms of the union is currently valid. A weak point of the union in C is that there is no indication in the union itself as to which of the (possibly many) different elements is currently valid. The slide shows an example of the union construct that is used in MIDL.



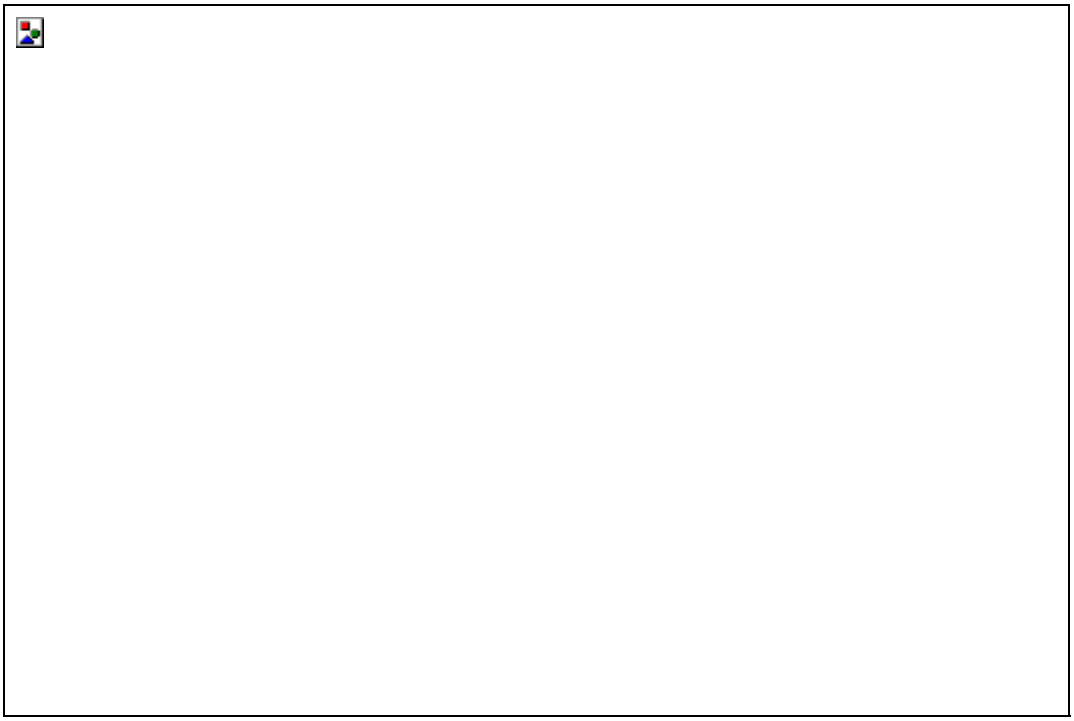
Data attributes are associated with the various data elements in the IDL file.

Directional attributes are used to indicate in which direction the data is flowing.

in means the data is passed from client to server. The data is passed by value, just as in a C function. After the call, the value of the actual parameter will not be changed.

out means the data is passed back from the server to the client. The data element must be a pointer.
When the remote function is called, the pointer refers to memory, the procedure fills in the data and it is copied back to the client.

Parameters can be declared as **in,out.** This means that the server will receive a copy of a data element, change it and the client will see the changed value on return.



When dealing with complex types such as linked lists, it may be convenient to transmit objects of these types over the network in a different format (i.e. a different type). The **transmit_as** attribute allows you to do this. For example, in the slide we choose to send a linked list over the network as an array. So the *typedef* introduces a new type that effectively maps the linked list onto an array for transmission.

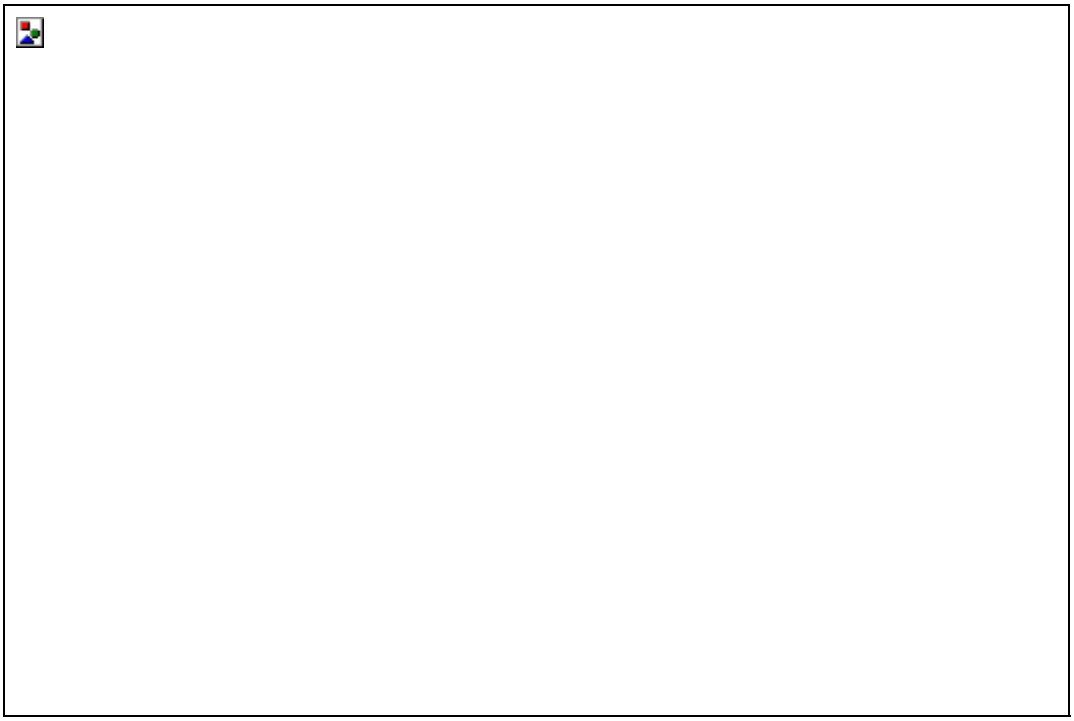
If you use this mechanism, you must write four functions that will be used on client or server (or both), depending on the directional attribute associated with the actual data element. The functions are:

type to_xmit() Convert an item in the presented type (the type on the system) into the type ready for transmission across the network. Required on the client for **in** parameters, on the server for **out** parameters and on both for **in, out** parameters.

***type* from _xmit()** Convert an item from transmitted type into presentation type. Required on the server for in parameters, on the client for **out** parameters and on both for **in, out** parameters.

type_free_inst() Free memory on server (callee) for the presented type. Once the data has been converted to the transmitted type, the memory for the original object is no longer needed. The function must be present with the **type_to_xmit()** function. It will normally involve simply calling the **midi_user_free** function.

type_free_xmit() Free memory on server (callee) for the transmitted type. This memory will have been allocated to hold the object in the transmitted type. Once it has been converted to the presented type, the memory is no longer needed. The function must be present with the **type_free_xmit** function. Like **type_free_inst,** this function will normally involve simply calling the **midl_user_free** function.



There is no security built in to Microsoft RPC 1.0. The security features of the underlying transports can be used, and the Kerberos authentication facility is available as an add-in.

This means that Windows NT **RPC** servers will have difficulty dealing with authenticated requests from DCE clients.

Future releases of Windows NT and Microsoft RPC will support fully authenticated RPC, using the same mechanisms as DCE (i.e. Kerberos).