2.2 REMOTE PROCEDURE CALL

Many distributed systems have been based on explicit message exchange between processes. However, the procedures send and receive do not conceal communication, which is important to achieve access transparency in distributed systems. This problem has long been known, but little was done about it until a paper by Birrell and Nelson (1984) introduced a completely different way of handling communication. Although the idea is refreshingly simple (once someone has thought of it), the implications are often subtle. In this section we will examine the concept, its implementation, its strengths, and its weaknesses.

In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on other machines. When a process on machine *A* calls a procedure on machine *B*, the calling process on *A* is suspended, and execution of the called procedure takes place on *B*. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer. This method is known as **Remote Procedure Call**, or often just **RPC**.

While the basic idea sounds simple and elegant, subtle problems exist. To start with, because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications. Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical. Finally, both machines can crash and each of the possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed systems.

2.2.1 Basic RPC Operation

We first start with discussing conventional procedure calls, and then explain how the call itself can be split into a client and server part that are each executed on different machines.

Conventional Procedure Call

To understand how RPC works, it is important first to fully understand how a conventional (i.e., single machine) procedure call works. Consider a call in C like

```
count = read(fd, buf, nbytes);
```

where *fd* is an integer indicating a file, *buf* is an array of characters into which data are read, and *nbytes* is another integer telling how many bytes to read. If the call is made from the main program, the stack will be as shown in Fig. 2-1(a) before the call. To make the call, the caller pushes the parameters onto the stack in order, last one first, as shown in Fig. 2-1(b). (The reason that C compilers push the parameters in reverse order has to do with *printf*—by doing so, *printf* can always locate its first parameter, the format string.) After read has finished

running, it puts the return value in a register, removes the return address, and transfers control back to the caller. The caller then removes the parameters from the stack, returning it to the original state.

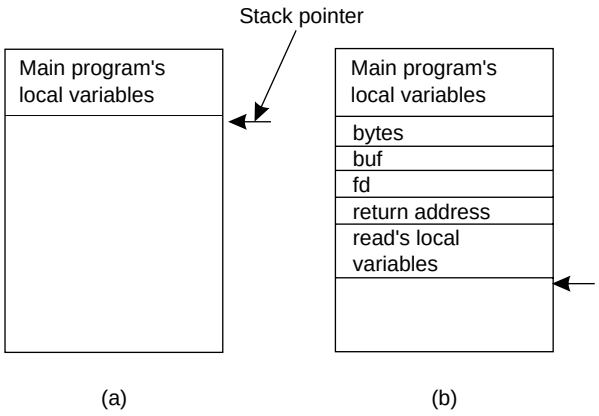

Figure 2-1. (a) Parameter passing in a local procedure call: the stack before the call to read. (b) The stack while the called procedure is active.

Several things are worth noting. For one, in C, parameters can be **call-by-value** or **call-by-reference**. A value parameter, such as *fd* or *nbytes*, is simply copied to the stack as shown in Fig. 2-1(b). To the called procedure, a value parameter is just an initialized local variable. The called procedure may modify it, but such changes do not affect the original value at the calling side.

A reference parameter in C is a pointer to a variable (i.e., the address of the variable), rather than the value of the variable. In the call to read, the second parameter is a reference parameter because arrays are always passed by reference in C. What is actually pushed onto the stack is the address of the character array. If the called procedure uses this parameter to store something into the character array, it *does* modify the array in the calling procedure. The difference between call-by-value and call-by-reference is quite important for RPC, as we shall see.

One other parameter passing mechanism also exists, although it is not used in C. It is called **call-by-copy/restore**. It consists of having the variable copied to the stack by the caller, as in call-by-value, and then copied back after the call, overwriting the caller's original value. Under most conditions, this achieves exactly the same effect as call-by-reference, but in some situations, such as the same parameter being present multiple times in the parameter list, the semantics are different. The call-by-copy/restore mechanism is not used in many languages.

The decision of which parameter passing mechanism to use is normally made by the language designers and is a fixed property of the language. Sometimes it depends on the data type being passed. In C, for example, integers and other scalar types are always passed by value, whereas arrays are always passed by reference, as we have seen. Some Ada compilers use copy/restore for **in out**

parameters, but others use call-by-reference. The language definition permits either choice, which makes the semantics a bit fuzzy.

Client and Server Stubs

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, we want RPC to be transparent—the calling procedure should not be aware that the called procedure is executing on a different machine or vice versa. Suppose that a program needs to read some data from a file. The programmer puts a call to read in the code to get the data. In a traditional (single-processor) system, the read routine is extracted from the library by the linker and inserted into the object program. It is a short procedure, which is generally implemented by calling an equivalent read system call. In other words, the read procedure is a kind of interface between the user code and the local operating system.

Even though read does a system call, it is called in the usual way, by pushing the parameters onto the stack, as shown in Fig. 2-1(b). Thus the programmer does not know that read is actually doing something fishy.

RPC achieves its transparency in an analogous way. When read is actually a remote procedure (e.g., one that will run on the file server's machine), a different version of read, called a **client stub**, is put into the library. Like the original one, it, too, is called using the calling sequence of Fig. 2-1(b). Also like the original one, it too, does a call to the local operating system. Only unlike the original one, it does not ask the operating system to give it data. Instead, it packs the parameters into a message and requests that message to be sent to the server as illustrated in Fig. 2-2. Following the call to send, the client stub calls receive, blocking itself until the reply comes back.

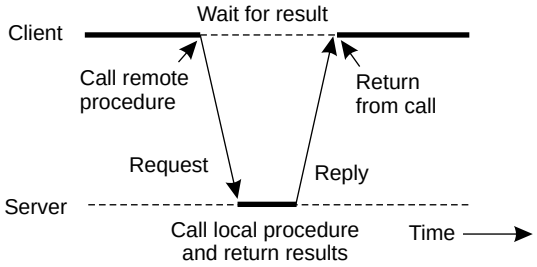

Figure 2-2. Principle of RPC between a client and server program.

When the message arrives at the server, the server's operating system passes it up to a **server stub**. A server stub is the server-side equivalent of a client stub: it is a piece of code that transforms requests coming in over the network into local procedure calls. Typically the server stub will have called receive and be blocked waiting for incoming messages. The server stub unpacks the parameters from the

message and then calls the server procedure in the usual way (i.e., as in Fig. 2-1). From the server's point of view, it is as though it is being called directly by the client—the parameters and return address are all on the stack where they belong and nothing seems unusual. The server performs its work and then returns the result to the caller in the usual way. For example, in the case of read, the server will fill the buffer, pointed to by the second parameter, with the data. This buffer will be internal to the server stub.

When the server stub gets control back after the call has completed, it packs the result (the buffer) in a message and calls send to return it to the client. After that, the server stub usually does a call to receive again, to wait for the next incoming request.

When the message gets back to the client machine, the client's operating system sees that it is addressed to the client process (or actually the client stub, but the operating system cannot see the difference). The message is copied to the waiting buffer and the client process unblocked. The client stub inspects the message, unpacks the result, copies it to its caller, and returns in the usual way. When the caller gets control following the call to read, all it knows is that its data are available. It has no idea that the work was done remotely instead of by the local operating system.

This blissful ignorance on the part of the client is the beauty of the whole scheme. As far as it is concerned, remote services are accessed by making ordinary (i.e., local) procedure calls, not by calling send and receive. All the details of the message passing are hidden away in the two library procedures, just as the details of actually making system calls are hidden away in traditional libraries.

To summarize, a remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

The net effect of all these steps is to convert the local call by the client procedure to the client stub, to a local call to the server procedure without either client or

server being aware of the intermediate steps.

2.2.2 Parameter Passing

The function of the client stub is to take its parameters, pack them into a message, and send them to the server stub. While this sounds straightforward, it is not quite as simple as it at first appears. In this section we will look at some of the issues concerned with parameter passing in RPC systems.

Passing Value Parameters

Packing parameters into a message is called **parameter marshaling**. As a very simple example, consider a remote procedure, add(i, j), that takes two integer parameters *i* and *j* and returns their arithmetic sum as a result. (As a practical matter, one would not normally make such a simple procedure remote due to the overhead, but as an example it will do.) The call to add, is shown in the left-hand portion (in the client process) in Fig. 2-3. The client stub takes its two parameters and puts them in a message as indicated. It also puts the name or number of the procedure to be called in the message because the server might support several different calls, and it has to be told which one is required.

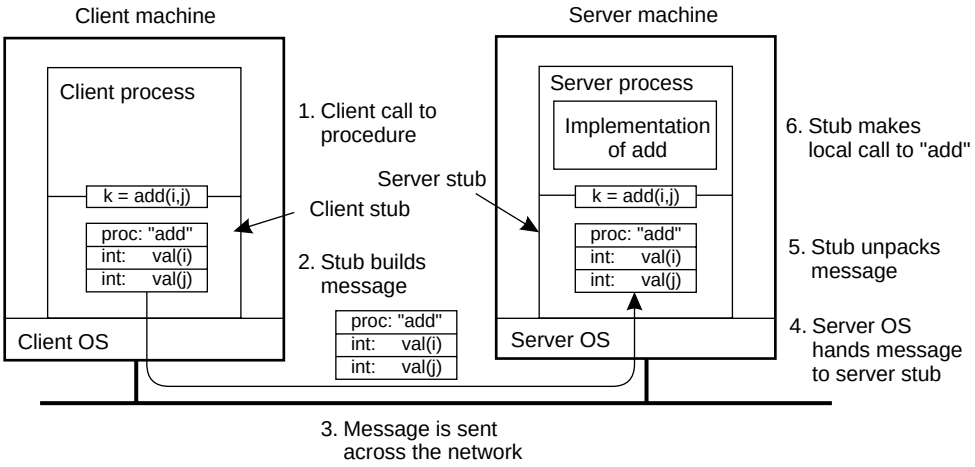

Figure 2-3. The steps involved in a doing a remote computation through RPC.

When the message arrives at the server, the stub examines the message to see which procedure is needed and then makes the appropriate call. If the server also supports other remote procedures, the server stub might have a switch statement in it to select the procedure to be called, depending on the first field of the message. The actual call from the stub to the server looks much like the original client call, except that the parameters are variables initialized from the incoming

message.

When the server has finished, the server stub gains control again. It takes the result provided by the server and packs it into a message. This message is sent back to the client stub, which unpacks it and returns the value to the client procedure.

As long as the client and server machines are identical and all the parameters and results are scalar types, such as integers, characters, and Booleans, this model works fine. However, in a large distributed system, it is common that multiple machine types are present. Each machine often has its own representation for numbers, characters, and other data items. For example, IBM mainframes use the EBCDIC character code, whereas IBM personal computers use ASCII. As a consequence, it is not possible to pass a character parameter from an IBM PC client to an IBM mainframe server using the simple scheme of Fig. 2-3: the server will interpret the character incorrectly.

Similar problems can occur with the representation of integers (one's complement versus two's complement) and floating-point numbers. In addition, an even more annoying problem exists because some machines, such as the Intel Pentium, number their bytes from right to left, whereas others, such as the Sun SPARC, number them the other way. The Intel format is called **little endian** and the SPARC format is called **big endian**, after the politicians in *Gulliver's Travels* who went to war over which end of an egg to break (Cohen, 1981). As an example, consider a procedure with two parameters, an integer and a four-character string. Each parameter requires one 32-bit word. Fig. 2-4(a) shows what the parameter portion of a message built by a client stub on an Intel Pentium might look like. The first word contains the integer parameter, 5 in this case, and the second contains the string "JILL."

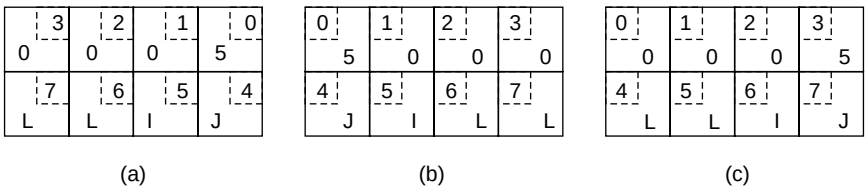

Figure 2-4. (a) The original message on the Pentium. (b) The message after receipt on the SPARC. (c) The message after being inverted. The little numbers in boxes indicate the address of each byte.

Since messages are transferred byte for byte (actually, bit for bit) over the network, the first byte sent is the first byte to arrive. In Fig. 2-4(b) we show what the message of Fig. 2-4(a) would look like if received by a SPARC, which numbers its bytes with byte 0 at the left (high-order byte) instead of at the right (low-order byte) as do all the Intel chips. When the server stub reads the parameters at addresses 0 and 4, respectively, it will find an integer equal to 83,886,080

(5×2^{24}) and a string ''JILL.''

One obvious, but unfortunately incorrect, approach is to simply invert the bytes of each word after they are received, leading to Fig. 2-4(c). Now the integer is 5 and the string is ''LLIJ.'' The problem here is that integers are reversed by the different byte ordering, but strings are not. Without additional information about what is a string and what is an integer, there is no way to repair the damage.

Passing Reference Parameters

We now come to a difficult problem: How are pointers, or in general, references passed? The answer is: only with the greatest of difficulty, if at all. Remember that a pointer is meaningful only within the address space of the process in which it is being used. Getting back to our read example discussed earlier, if the second parameter (the address of the buffer) happens to be 1000 on the client, one cannot just pass the number 1000 to the server and expect it to work. Address 1000 on the server might be in the middle of the program text.

One solution is just to forbid pointers and reference parameters in general. However, these are so important that this solution is highly undesirable. In fact, it is not necessary either. In the read example, the client stub knows that the second parameter points to an array of characters. Suppose, for the moment, that it also knows how big the array is. One strategy then becomes apparent: copy the array into the message and send it to the server. The server stub can then call the server with a pointer to this array, even though this pointer has a different numerical value than the second parameter of read has. Changes the server makes using the pointer (e.g., storing data into it) directly affect the message buffer inside the server stub. When the server finishes, the original message can be sent back to the client stub, which then copies it back to the client. In effect, call-by-reference has been replaced by copy/restore. Although this is not always identical, it frequently is good enough.

One optimization makes this mechanism twice as efficient. If the stubs know whether the buffer is an input parameter or an output parameter to the server, one of the copies can be eliminated. If the array is input to the server (e.g., in a call to write) it need not be copied back. If it is output, it need not be sent over in the first place.

As a final comment, it is worth noting that although we can now handle pointers to simple arrays and structures, we still cannot handle the most general case of a pointer to an arbitrary data structure such as a complex graph. Some systems attempt to deal with this case by actually passing the pointer to the server stub and generating special code in the server procedure for using pointers. For example, a request may be sent back to the client to provide the referenced data.

Parameter Specification and Stub Generation

From what we have explained so far, it is clear that hiding a remote procedure call requires that the caller and the callee agree on the format of the messages they exchange, and that they follow the same steps when it comes to, for example, passing complex data structures. In other words, both sides in an RPC should follow the same protocol.

As a simple example, consider the procedure of Fig. 2-5(a). It has three parameters, a character, a floating-point number, and an array of five integers. Assuming a word is four bytes, the RPC protocol might prescribe that we should transmit a character in the rightmost byte of a word (leaving the next 3 bytes empty), a float as a whole word, and an array as a group of words equal to the array length, preceded by a word giving the length, as shown in Fig. 2-5(b). Thus given these rules, the client stub for foobar knows that it must use the format of Fig. 2-5(b), and the server stub knows that incoming messages for foobar will have the format of Fig. 2-5(b).

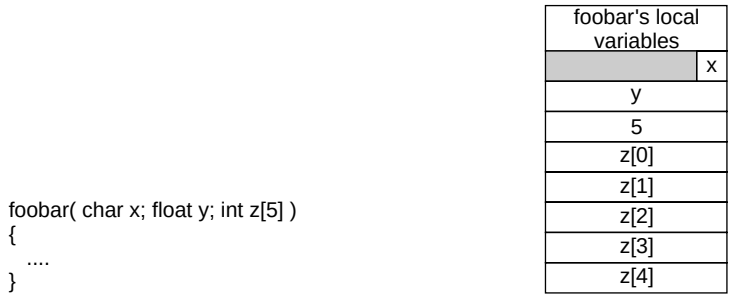

Figure 2-5. (a) A procedure. (b) The corresponding message.

Defining the message format is one aspect of an RPC protocol, but it is not sufficient. What we also need is the client and the server to agree on the representation of simple data structures, such as integers, characters, Booleans, etc. For example, the protocol could prescribe that integers are represented in two's complement, characters in 16-bit Unicode, and floats in the IEEE standard #754 format, with everything stored in little endian. With this additional information, messages can be unambiguously interpreted.

With the encoding rules now pinned down to the last bit, the only thing that remains to be done is that the caller and callee agree on the actual exchange of messages. For example, it may be decided to use a connection-oriented transport service such as TCP/IP. An alternative is to use an unreliable datagram service and let the client and server implement an error control scheme as part of the RPC protocol. In practice, several variants exist.

Once the RPC protocol has been completely defined, the client and server

stubs need to be implemented. Fortunately, stubs for the same protocol but different procedures generally differ only in their interface to the applications. An interface consists of a collection of procedures that can be called by a client, and which are implemented by a server. An interface is generally available in the same programming language as the one in which the client or server is written (although this is strictly speaking, not necessary). To simplify matters, interfaces are often specified by means of an **Interface Definition Language** (**IDL**). An interface specified in such an IDL, is then subsequently compiled into a client stub and a server stub, along with the appropriate compile-time or run-time interfaces.

Practice shows that using an interface definition language considerably simplifies client-server applications based on RPCs. Because it is easy to fully generate client and server stubs, all RPC-based middleware systems offer an IDL to support application development. In some cases, using the IDL is even mandatory, as we shall see in later chapters.

2.2.3 Extended RPC Models

Remote procedure calls have become a de facto standard for communication in distributed systems. The popularity of the model is due to its apparent simplicity. In this section, we take a brief look at two extensions to the original RPC model that have been designed to solve some of its shortcomings.

Doors

The original RPC model assumes that the caller and callee can communicate only by means of passing messages over a network. In general, this assumption is correct. However, suppose that the client and server reside on the same machine. Normally, we would make use of the local interprocess communication (IPC) facilities that the underlying operating system offers to processes running on the same machine. For example, in UNIX such facilities include shared memory, pipes, and message queues (see Stevens, 1999 for a detailed discussion on IPC in UNIX systems).

Local IPC facilities tend to be much more efficient than networking facilities, even if the latter are used for communication between processes on the same machine. Consequently, when performance is an issue, different interprocess communication mechanisms may need to be combined depending on whether or not the processes we are dealing with are located on the same machine.

As a compromise, a few operating systems offer an equivalent of RPCs for processes that are colocated on the same machine, called doors. A **door** is a generic name for a procedure in the address space of a server process that can be called by processes colocated with the server. Doors were originally designed for the Spring operating system (Mitchell et al., 1994), and are described extensively

in (Hamilton and Kougiouris 1993). A similar mechanism, called Lightweight RPC, was developed by Bershad et al. (1990).

Calling doors requires support from the local operating system, as shown in Fig. 2-6. In particular, the server process must first register a door before it can be called. When registering a door, an identifier for that door is returned that can be used to later give the door a symbolic name. Registration is done by a call to door_create. A registered door can be made available to other processes by simply associating a name with the identifier returned when the door was registered. For example, in Solaris, each door has a file name, which is associated with the door's identifier by a call to fattach. A client calls a door by means of the system call door_call, to which it passes the identification of the door as well as any necessary parameters. The operating system then does an *upcall* to the server process that registered the door. An upcall results in an invocation of the door by the server. The results of invoking the door are returned to the client process through the system call door_return.

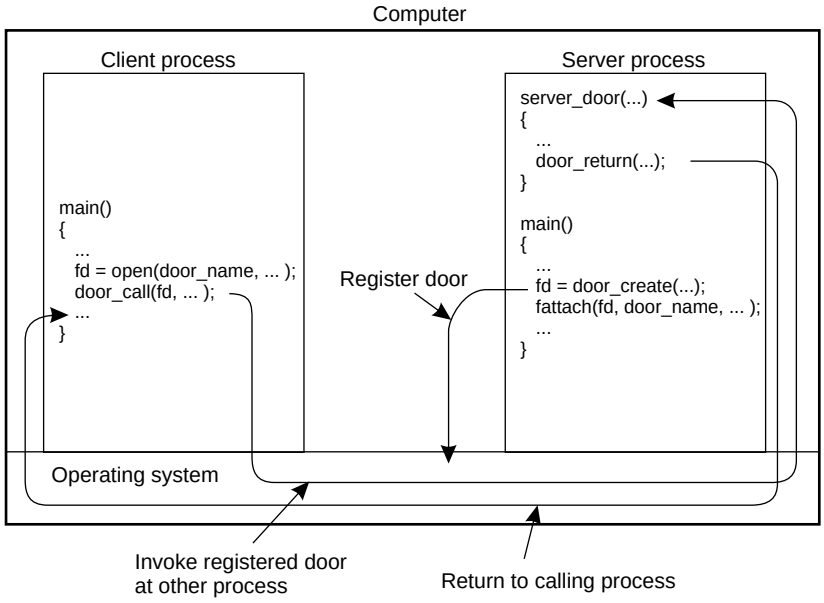

Figure 2-6. The principle of using doors as IPC mechanism.

The main benefit of doors is that they allow the use of a single mechanism, namely procedure calls, for communication in a distributed system. Unfortunately, application developers still need to be aware whether a call is done local within the current process, local to a different process on the same machine, or to a remote process.

Asynchronous RPC

As in conventional procedure calls, when a client calls a remote procedure, the client will block until a reply is returned. This strict request-reply behavior is unnecessary when there is no result to return, and only leads to blocking the client while it could have proceeded and have done useful work just after requesting the remote procedure to be called. Examples of where there is often no need to wait for a reply include: transferring money from one account to another, adding entries into a database, starting remote services, batch processing, and so on.

To support such situations, RPC systems may provide facilities for what are called **asynchronous RPCs**, by which a client immediately continues after issuing the RPC request. With asynchronous RPCs, the server immediately sends a reply back to the client the moment the RPC request is received, after which it calls the requested procedure. The reply acts as an acknowledgement to the client that the server is going to process the RPC. The client will continue without further blocking as soon as it has received the server's acknowledgement. Fig. 2-7(b) shows how client and server interact in the case of asynchronous RPCs. For comparison, Fig. 2-7(a) shows the normal request-reply behavior.

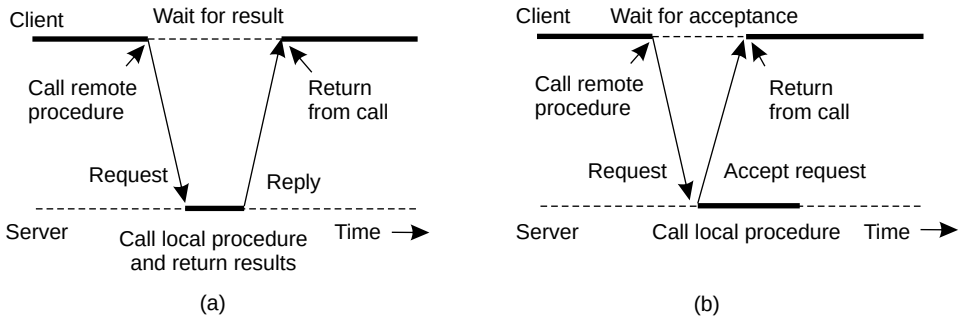

Figure 2-7. (a) The interaction between client and server in a traditional RPC. (b) The interaction using asynchronous RPC.

Asynchronous RPCs can also be useful when a reply will be returned but the client is not prepared to wait for it and do nothing in the meantime. For example, a client may want to prefetch the network addresses of a set of hosts that it expects to contact soon. While a naming service is collecting those addresses, the client may want to do other things. In such cases, it makes sense to organize the communication between the client and server through two asynchronous RPCs, as shown in Fig. 2-8. The client first calls the server to hand over a list of host names that should be looked up, and continues when the server has acknowledged the receipt of that list. The second call is done by the server, who calls the client to hand over the addresses it found. Combining two asynchronous RPCs is sometimes also referred to as a **deferred synchronous RPC**.

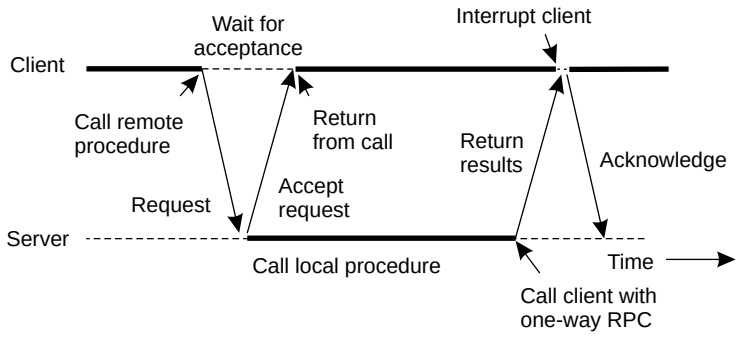

Figure 2-8. A client and server interacting through two asynchronous RPCs.

It should be noted that variants of asynchronous RPCs exist in which the client continues immediately after sending the request to the server. In other words, the client does not wait for an acknowledgement of the server's acceptance of the request. We refer to such RPCs as **one-way RPCs**. The problem with this approach is that if reliability is not guaranteed, the client cannot know for sure whether its request will be processed. We return to these matters in Chap. 7.

2.2.4 Example: DCE RPC

Remote procedure calls have been widely adopted as the basis of middleware and distributed systems in general. In this section, we take a closer look at one specific RPC system: the **Distributed Computing Environment** (**DCE**), which has been developed by the Open Software Foundation (OSF) now called The Open Group. DCE RPC is not as popular as some other RPC systems, notably Sun RPC. However, DCE RPC is highly representative of other RPC systems, and its specifications have been adopted in Microsoft's base system for distributed computing. In addition, as we shall see in a later section, DCE RPC is also illustrative for understanding the relation between RPC systems and distributed objects. We start with a brief introduction to DCE, after which we consider the principal workings of DCE RPC.

Introduction to DCE

DCE is a true middleware system in that it is designed to execute as a layer of abstraction between existing (network) operating systems and distributed applications. Initially designed for UNIX, it has now been ported to all major operating systems including VMS and Windows NT, as well as desktop operating systems. The idea is that the customer can take a collection of existing machines, add the DCE software, and then be able to run distributed applications, all without disturbing existing (nondistributed) applications. Although most of the DCE package

runs in user space, in some configurations a piece (part of the distributed file system) must be added to the kernel. The Open Group itself only sells source code, which vendors integrate into their systems.

The programming model underlying all of DCE is the client-server model, which was extensively discussed in the previous chapter. User processes act as clients to access remote services provided by server processes. Some of these services are part of DCE itself, but others belong to the applications and are written by the applications programmers. All communication between clients and servers takes place by means of RPCs.

There are a number of services that form part of DCE itself. The **distributed file service** is a worldwide file system that provides a transparent way of accessing any file in the system in the same way. It can either be built on top of the hosts' native file systems or be used instead of them. The **directory service** is used to keep track of the location of all resources in the system. These resources include machines, printers, servers, data, and much more, and they may be distributed geographically over the entire world. The directory service allows a process to ask for a resource and not have to be concerned about where it is, unless the process cares. The **security service** allows resources of all kinds to be protected, so access can be restricted to authorized persons. Finally, the **distributed time service** is a service that attempts to keep clocks on the different machines globally synchronized. As we shall see in later chapters, having some notion of global time makes it much easier to ensure consistency in a distributed system.

Goals of DCE RPC

The goals of the DCE RPC system are relatively traditional. First and foremost, the RPC system makes it possible for a client to access a remote service by simply calling a local procedure. This interface makes it possible for client (i.e., application) programs to be written in a simple way, familiar to most programmers. It also makes it easy to have large volumes of existing code run in a distributed environment with few, if any, changes.

It is up to the RPC system to hide all the details from the clients, and, to some extent, from the servers as well. To start with, the RPC system can automatically locate the correct server, and subsequently set up the communication between client and server software (generally called **binding**). It can also handle the message transport in both directions, fragmenting and reassembling them as needed (e.g., if one of the parameters is a large array). Finally, the RPC system can automatically handle data type conversions between the client and the server, even if they run on different architectures and have a different byte ordering.

As a consequence of the RPC system's ability to hide the details, clients and servers are highly independent of one another. A client can be written in Java and a server in C, or vice versa. A client and server can run on different hardware platforms and use different operating systems. A variety of network protocols and

data representations are also supported, all without any intervention from the client or server.

Writing a Client and a Server

The DCE RPC system consists of a number of components, including languages, libraries, daemons, and utility programs, among others. Together these make it possible to write clients and servers. In this section we will describe the pieces and how they fit together. The entire process of writing and using an RPC client and server is summarized in Fig. 2-9.

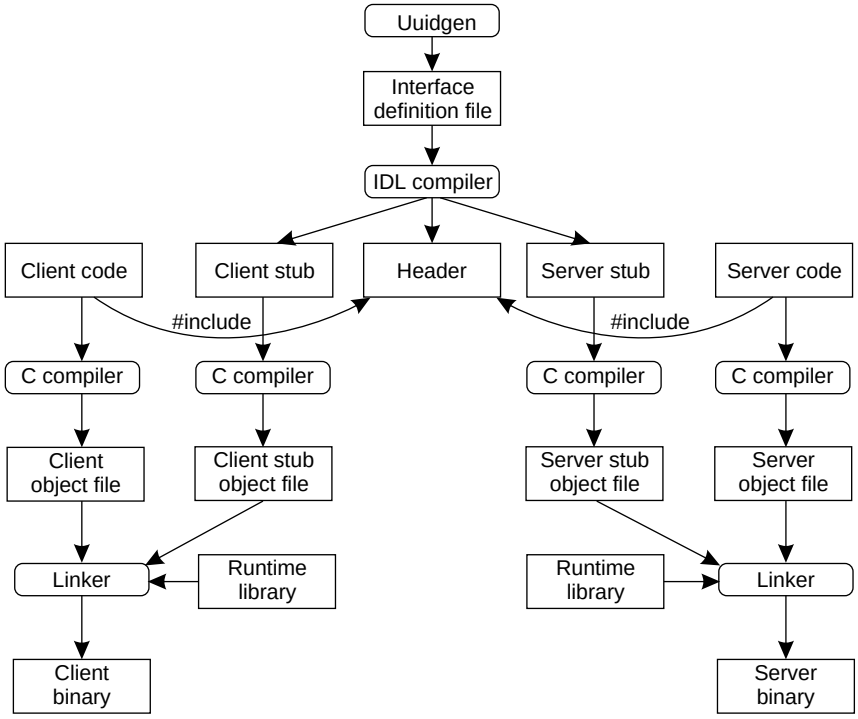

Figure 2-9. The steps in writing a client and a server in DCE RPC.

In a client-server system, the glue that holds everything together is the interface definition, as specified in the **Interface Definition Language**, or **IDL**. It permits procedure declarations in a form closely resembling function prototypes in ANSI C. IDL files can also contain type definitions, constant declarations, and other information needed to correctly marshal parameters and unmarshal results. Ideally, the interface definition should also contain a formal definition of what the procedures do, but such a definition is beyond the current state of the art, so the interface definition just defines the syntax of the calls, not their semantics. At best

the writer can add a few comments describing what the procedures do.

A crucial element in every IDL file is a globally unique identifier for the specified interface. The client sends this identifier in the first RPC message and the server verifies that it is correct. In this way, if a client inadvertently tries to bind to the wrong server, or even to an older version of the right server, the server will detect the error and the binding will not take place.

Interface definitions and unique identifiers are closely related in DCE. As illustrated in Fig. 2-9, the first step in writing a client/server application is usually calling the *uuidgen* program, asking it to generate a prototype IDL file containing an interface identifier guaranteed never to be used again in any interface generated anywhere by *uuidgen*. Uniqueness is ensured by encoding in it the location and time of creation. It consists of a 128-bit binary number represented in the IDL file as an ASCII string in hexadecimal.

The next step is editing the IDL file, filling in the names of the remote procedures and their parameters. It is worth noting that RPC is not totally transparent—for example, the client and server cannot share global variables—but the IDL rules make it impossible to express constructs that are not supported.

When the IDL file is complete, the IDL compiler is called to process it. The output of the IDL compiler consists of three files:

1. A header file (e.g., *interface.h*, in C terms).
2. The client stub.
3. The server stub.

The header file contains the unique identifier, type definitions, constant definitions, and function prototypes. It should be included (using *#include*) in both the client and server code. The client stub contains the actual procedures that the client program will call. These procedures are responsible for collecting and packing the parameters into the outgoing message and then calling the runtime system to send it. The client stub also handles unpacking the reply and returning values to the client. The server stub contains the procedures called by the runtime system on the server machine when an incoming message arrives. These, in turn, call the actual server procedures that do the work.

The next step is for the application writer to write the client and server code. Both of these are then compiled, as are the two stub procedures. The resulting client code and client stub object files are then linked with the runtime library to produce the executable binary for the client. Similarly, the server code and server stub are compiled and linked to produce the server's binary. At runtime, the client and server are started so that the application is actually executed as well.

Binding a Client to a Server

To allow a client to call a server, it is necessary that the server be registered and prepared to accept incoming calls. Registration of a server makes it possible for a client to actually locate the server and bind to it. Server location is done in two steps:

1. Locate the server's machine.
2. Locate the server (i.e., the correct process) on that machine.

The second step is somewhat subtle. Basically, what it comes down to is that to communicate with a server, the client needs to know an **endpoint**, on the server's machine to which it can send messages. An endpoint (also commonly known as a **port**) is used by the server's operating system to distinguish incoming messages for different processes. In DCE, a table of *(server, endpoint)*-pairs is maintained on each server machine by a process called the **DCE daemon**. Before it becomes available for incoming requests, the server must ask the operating system for an endpoint. It then registers this endpoint with the DCE daemon. The DCE daemon records this information (including which protocols the server speaks) in the endpoint table for future use.

The server also registers with the directory service by providing it the network address of the server's machine and a name under which the server can be looked up. Binding a client to a server then proceeds as shown in Fig. 2-10.

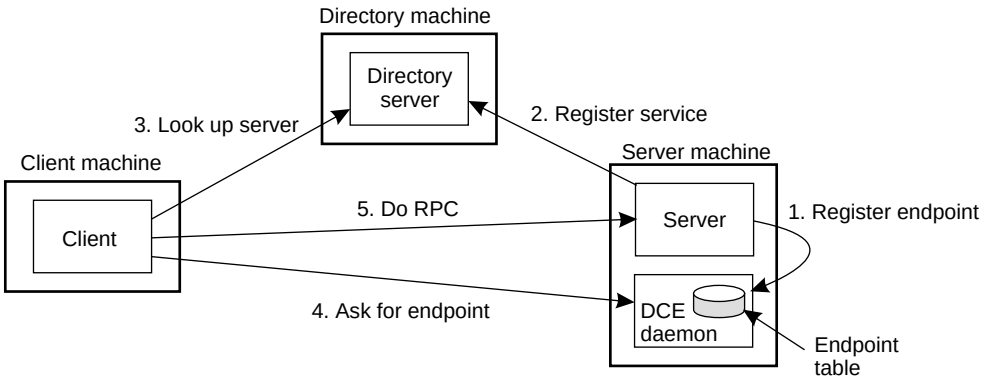

Figure 2-10. Client-to-server binding in DCE.

Let us assume that the client wants to bind to a video server that is locally known under the name */local/multimedia/video/movies*. It passes this name to the directory server, which returns the network address of the machine running the video server. The client then goes to the DCE daemon on that machine (which has

a well-known endpoint), and asks it to look up the endpoint of the video server in its endpoint table. Armed with this information, the RPC can now take place. On subsequent RPCs this lookup is not needed. DCE also gives clients the ability to do more sophisticated searches for a suitable server when that is needed. Secure RPC is also an option.

Performing an RPC

The actual RPC is carried out transparently and in the usual way. The client stub marshals the parameters to the runtime library for transmission using the protocol chosen at binding time. When a message arrives at the server side, it is routed to the correct server based on the endpoint contained in the incoming message. The runtime library passes the message to the server stub, which unmarshals the parameters and calls the server. The reply goes back by the reverse route.

DCE provides several semantic options. The default is **at-most-once operation**, in which case no call is ever carried out more than once, even in the face of system crashes. In practice, what this means is that if a server crashes during an RPC and then recovers quickly, the client does not repeat the operation, for fear that it might already have been carried out once.

Alternatively, it is possible to mark a remote procedure as **idempotent** (in the IDL file), in which case it can be repeated multiple times without harm. For example, reading a specified block from a file can be tried over and over until it succeeds. When an idempotent RPC fails due to a server crash, the client can wait until the server reboots and then try again. Other semantics are also available (but rarely used), including broadcasting the RPC to all the machines on the local network. We return to RPC semantics in Chap. 7, when discussing RPC in the presence of failures.