CHAPTER 4

Remote Procedure Calls

4.1 INTRODUCTION

The general message-passing model of interprocess communication (IPC) was presented in the previous chapter. The IPC part of a distributed application can often be adequately and efficiently handled by using an IPC protocol based on the message-passing model. However, an independently developed IPC protocol is tailored specifically to one application and does not provide a foundation on which to build a variety of distributed applications. Therefore, a need was felt for a general IPC protocol that can be used for designing several distributed applications. The Remote Procedure Call (RPC) facility emerged out of this need. It is a special case of the general message-passing model of IPC. Providing the programmers with a familiar mechanism for building distributed systems is one of the primary motivations for developing the RPC facility. While the RPC facility is not a universal panacea for all types of distributed applications, it does provide a valuable communication mechanism that is suitable for building a fairly large number of distributed applications.

The RPC has become a widely accepted IPC mechanism in distributed systems. The popularity of RPC as the primary communication mechanism for distributed applications is due to its following features:

1. Simple call syntax.
2. Familiar semantics (because of its similarity to local procedure calls).
3. Its specification of a well-defined interface. This property is used to support compile-time type checking and automated interface generation.
4. Its ease of use. The clean and simple semantics of a procedure call makes it easier to build distributed computations and to get them right.
5. Its generality. This feature is owing to the fact that in single-machine computations procedure calls are often the most important mechanism for communication between parts of the algorithm [Birrell and Nelson 1984].
6. Its efficiency. Procedure calls are simple enough for communication to be quite rapid.
7. It can be used as an IPC mechanism to communicate between processes on different machines as well as between different processes on the same machine.

4.2 THE RPC MODEL

The RPC model is similar to the well-known and well-understood procedure call model used for the transfer of control and data within a program in the following manner:

1. For making a procedure call, the caller places arguments to the procedure in some well-specified location.
2. Control is then transferred to the sequence of instructions that constitutes the body of the procedure.
3. The procedure body is executed in a newly created execution environment that includes copies of the arguments given in the calling instruction.
4. After the procedure's execution is over, control returns to the calling point, possibly returning a result.

The RPC mechanism is an extension of the procedure call mechanism in the sense that it enables a call to be made to a procedure that does not reside in the address space of the calling process. The called procedure (commonly called *remote procedure*) may be on the same computer as the calling process or on a different computer.

In case of RPC, since the caller and the callee processes have disjoint address spaces (possibly on different computers), the remote procedure has no access to data and variables of the caller's environment. Therefore the RPC facility uses a message-passing scheme for information exchange between the caller and the callee processes. As shown in Figure 4.1, when a remote procedure call is made, the caller and the callee processes interact in the following manner:

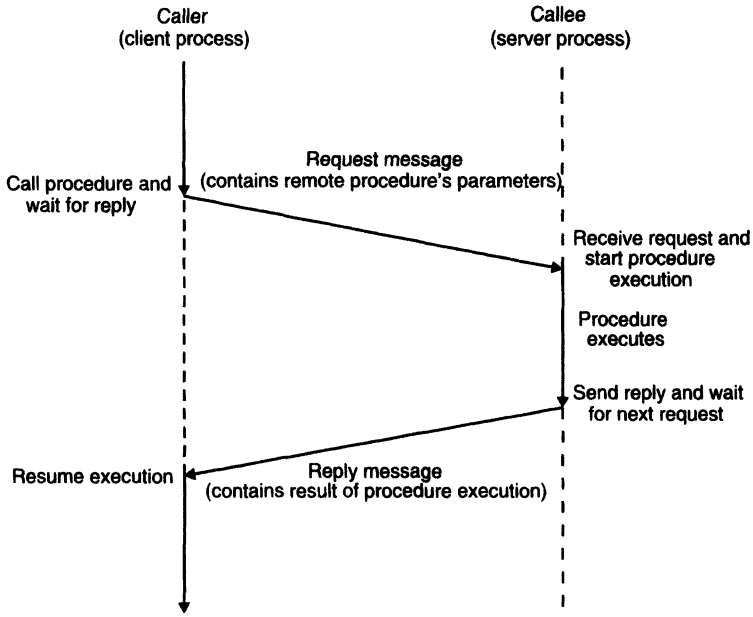


Fig. 4.1 A typical model of Remote Procedure Call.

1. The caller (commonly known as *client process*) sends a call (request) message to the callee (commonly known as *server process*) and waits (blocks) for a reply message. The request message contains the remote procedure's parameters, among other things.
2. The server process executes the procedure and then returns the result of procedure execution in a reply message to the client process.
3. Once the reply message is received, the result of procedure execution is extracted, and the caller's execution is resumed.

The server process is normally dormant, awaiting the arrival of a request message. When one arrives, the server process extracts the procedure's parameters, computes the result, sends a reply message, and then awaits the next call message.

Note that in this model of RPC, only one of the two processes is active at any given time. However, in general, the RPC protocol makes no restrictions on the concurrency model implemented, and other models of RPC are possible depending on the details of the parallelism of the caller's and callee's environments and the RPC implementation. For example, an implementation may choose to have RPC calls to be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a thread (threads are described in Chapter 8) to process an incoming request, so that the server can be free to receive other requests.

4.3 TRANSPARENCY OF RPC

A major issue in the design of an RPC facility is its transparency property. A transparent RPC mechanism is one in which local procedures and remote procedures are (effectively) indistinguishable to programmers. This requires the following two types of transparencies [Wilbur and Bacarisse 1987]:

1. *Syntactic transparency* means that a remote procedure call should have exactly the same syntax as a local procedure call.
2. *Semantic transparency* means that the semantics of a remote procedure call are identical to those of a local procedure call.

It is not very difficult to achieve syntactic transparency of an RPC mechanism, and we have seen that the semantics of remote procedure calls are also analogous to that of local procedure calls for most parts:

- The calling process is suspended until the called procedure returns.
- The caller can pass arguments to the called procedure (remote procedure).
- The called procedure (remote procedure) can return results to the caller.

Unfortunately, achieving exactly the same semantics for remote procedure calls as for local procedure calls is close to impossible [Tanenbaum and Van Renesse 1988]. This is mainly because of the following differences between remote procedure calls and local procedure calls:

1. Unlike local procedure calls, with remote procedure calls, the called procedure is executed in an address space that is disjoint from the calling program's address space. Due to this reason, the called (remote) procedure cannot have access to any variables or data values in the calling program's environment. Thus in the absence of shared memory, it is meaningless to pass addresses in arguments, making call-by-reference pointers highly unattractive. Similarly, it is meaningless to pass argument values containing pointer structures (e.g., linked lists), since pointers are normally represented by memory addresses. According to Bal et al. [1989], dereferencing a pointer passed by the caller has to be done at the caller's side, which implies extra communication. An alternative implementation is to send a copy of the value pointed at the receiver, but this has subtly different semantics and may be difficult to implement if the pointer points into the middle of a complex data structure, such as a directed graph. Similarly, call by reference can be replaced by copy in/copy out, but at the cost of slightly different semantics.

2. Remote procedure calls are more vulnerable to failure than local procedure calls, since they involve two different processes and possibly a network and two different computers. Therefore programs that make use of remote procedure calls must have the capability of handling even those errors that cannot occur in local procedure calls. The need for the ability to take care of the possibility of processor crashes and communication

problems of a network makes it even more difficult to obtain the same semantics for remote procedure calls as for local procedure calls.

3. Remote procedure calls consume much more time (100–1000 times more) than local procedure calls. This is mainly due to the involvement of a communication network in RPCs. Therefore applications using RPCs must also have the capability to handle the long delays that may possibly occur due to network congestion.

Because of these difficulties in achieving normal call semantics for remote procedure calls, some researchers feel that the RPC facility should be nontransparent. For example, Hamilton [1984] argues that remote procedures should be treated differently from local procedures from the start, resulting in a nontransparent RPC mechanism. Similarly, the designers of RPC in Argus [Liskov and Scheifler 1983] were of the opinion that although the RPC system should hide low-level details of message passing from the users, failures and long delays should not be hidden from the caller. That is, the caller should have the flexibility of handling failures and long delays in an application-dependent manner. In conclusion, although in most environments total semantic transparency is impossible, enough can be done to ensure that distributed application programmers feel comfortable.

4.4 IMPLEMENTING RPC MECHANISM

To achieve the goal of semantic transparency, the implementation of an RPC mechanism is based on the concept of *stubs*, which provide a perfectly normal (local) procedure call abstraction by concealing from programs the interface to the underlying RPC system. We saw that an RPC involves a client process and a server process. Therefore, to conceal the interface of the underlying RPC system from both the client and server processes, a separate stub procedure is associated with each of the two processes. Moreover, to hide the existence and functional details of the underlying network, an RPC communication package (known as *RPCRuntime*) is used on both the client and server sides. Thus, implementation of an RPC mechanism usually involves the following five elements of program [Birrell and Nelson 1984]:

1. The client
2. The client stub
3. The RPCRuntime
4. The server stub
5. The server

The interaction between them is shown in Figure 4.2. The client, the client stub, and one instance of RPCRuntime execute on the client machine, while the server, the server stub, and another instance of RPCRuntime execute on the server machine. The job of each of these elements is described below.

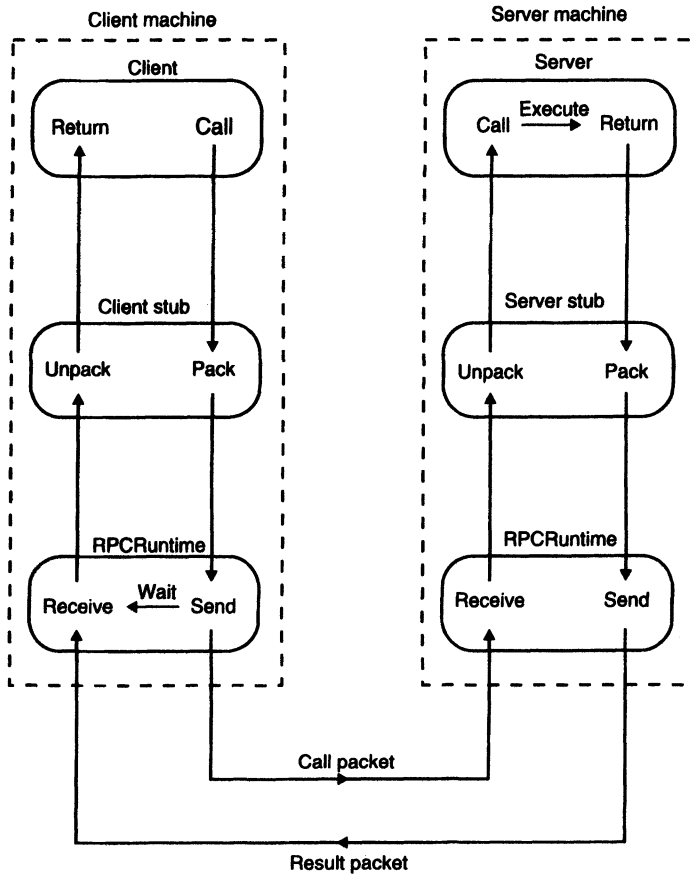


Fig. 4.2 Implementation of RPC mechanism.

4.4.1 Client

The client is a user process that initiates a remote procedure call. To make a remote procedure call, the client makes a perfectly normal local call that invokes a corresponding procedure in the client stub.

4.4.2 Client Stub

The client stub is responsible for carrying out the following two tasks:

- On receipt of a call request from the client, it packs a specification of the target procedure and the arguments into a message and then asks the local RPCRuntime to send it to the server stub.

- On receipt of the result of procedure execution, it unpacks the result and passes it to the client.

4.4.3 RPCRuntime

The RPCRuntime handles transmission of messages across the network between client and server machines. It is responsible for retransmissions, acknowledgments, packet routing, and encryption. The RPCRuntime on the client machine receives the call request message from the client stub and sends it to the server machine. It also receives the message containing the result of procedure execution from the server machine and passes it to the client stub.

On the other hand, the RPCRuntime on the server machine receives the message containing the result of procedure execution from the server stub and sends it to the client machine. It also receives the call request message from the client machine and passes it to the server stub.

4.4.4 Server Stub

The job of the server stub is very similar to that of the client stub. It performs the following two tasks:

- On receipt of the call request message from the local RPCRuntime, the server stub unpacks it and makes a perfectly normal call to invoke the appropriate procedure in the server.
- On receipt of the result of procedure execution from the server, the server stub packs the result into a message and then asks the local RPCRuntime to send it to the client stub.

4.4.5 Server

On receiving a call request from the server stub, the server executes the appropriate procedure and returns the result of procedure execution to the server stub.

Note here that the beauty of the whole scheme is the total ignorance on the part of the client that the work was done remotely instead of by the local kernel. When the client gets control following the procedure call that it made, all it knows is that the results of the procedure execution are available to it. Therefore, as far as the client is concerned, remote services are accessed by making ordinary (local) procedure calls, not by using the *send* and *receive* primitives of Chapter 3. All the details of the message passing are hidden in the client and server stubs, making the steps involved in message passing invisible to both the client and the server.

4.5 STUB GENERATION

Stubs can be generated in one of the following two ways:

1. *Manually.* In this method, the RPC implementor provides a set of translation functions from which a user can construct his or her own stubs. This method is simple to implement and can handle very complex parameter types.

2. *Automatically.* This is the more commonly used method for stub generation. It uses *Interface Definition Language* (*IDL*) that is used to define the interface between a client and a server. An interface definition is mainly a list of procedure names supported by the interface, together with the types of their arguments and results. This is sufficient information for the client and server to independently perform compile-time type-checking and to generate appropriate calling sequences. However, an interface definition also contains other information that helps RPC reduce data storage and the amount of data transferred over the network. For example, an interface definition has information to indicate whether each argument is input, output, or both—only input arguments need be copied from client to server and only output arguments need be copied from server to client. Similarly, an interface definition also has information about type definitions, enumerated types, and defined constants that each side uses to manipulate data from RPC calls, making it unnecessary for both the client and the server to store this information separately. (See Figure 4.21 for an example of an interface definition.)

A server program that implements procedures in an interface is said to *export* the interface, and a client program that calls procedures from an interface is said to *import* the interface. When writing a distributed application, a programmer first writes an interface definition using the IDL. He or she can then write the client program that imports the interface and the server program that exports the interface. The interface definition is processed using an IDL compiler to generate components that can be combined with client and server programs, without making any changes to the existing compilers. In particular, from an interface definition, an IDL compiler generates a client stub procedure and a server stub procedure for each procedure in the interface, the appropriate marshaling and unmarshaling operations (described later in this chapter) in each stub procedure, and a header file that supports the data types in the interface definition. The header file is included in the source files of both the client and server programs, the client stub procedures are compiled and linked with the client program, and the server stub procedures are compiled and linked with the server program. An IDL compiler can be designed to process interface definitions for use with different languages, enabling clients and servers written in different languages, to communicate by using remote procedure calls.

4.6 RPC MESSAGES

Any remote procedure call involves a client process and a server process that are possibly located on different computers. The mode of interaction between the client and server is that the client asks the server to execute a remote procedure and the server returns the

result of execution of the concerned procedure to the client. Based on this mode of interaction, the two types of messages involved in the implementation of an RPC system are as follows:

1. *Call messages* that are sent by the client to the server for requesting execution of a particular remote procedure
2. *Reply messages* that are sent by the server to the client for returning the result of remote procedure execution

The protocol of the concerned RPC system defines the format of these two types of messages. Normally, an RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. Therefore an RPC protocol deals only with the specification and interpretation of these two types of messages.

4.6.1 Call Messages

Since a call message is used to request execution of a particular remote procedure, the two basic components necessary in a call message are as follows:

1. The identification information of the remote procedure to be executed
2. The arguments necessary for the execution of the procedure

In addition to these two fields, a call message normally has the following fields:

3. A message identification field that consists of a sequence number. This field is useful in two ways—for identifying lost messages and duplicate messages in case of system failures and for properly matching reply messages to outstanding call messages, especially in those cases where the replies of several outstanding call messages arrive out of order.
4. A message type field that is used to distinguish call messages from reply messages. For example, in an RPC system, this field may be set to 0 for all call messages and set to 1 for all reply messages.
5. A client identification field that may be used for two purposes—to allow the server of the RPC to identify the client to whom the reply message has to be returned and to allow the server to check the authentication of the client process for executing the concerned procedure.

Thus, a typical RPC call message format may be of the form shown in Figure 4.3.

4.6.2 Reply Messages

When the server of an RPC receives a call message from a client, it could be faced with one of the following conditions. In the list below, it is assumed for a particular condition that no problem was detected by the server for any of the previously listed conditions:

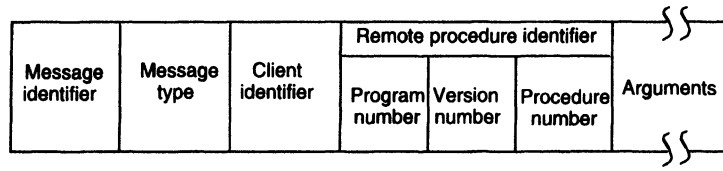


Fig. 4.3 A typical RPC call message format.

1. The server finds that the call message is not intelligible to it. This may happen when a call message violates the RPC protocol. Obviously the server will reject such calls.
2. The server detects by scanning the client's identifier field that the client is not authorized to use the service. The server will return an unsuccessful reply without bothering to make an attempt to execute the procedure.
3. The server finds that the remote program, version, or procedure number specified in the remote procedure identifier field of the call message is not available with it. Again the server will return an unsuccessful reply without bothering to make an attempt to execute the procedure.
4. If this stage is reached, an attempt will be made to execute the remote procedure specified in the call message. Therefore it may happen that the remote procedure is not able to decode the supplied arguments. This may happen due to an incompatible RPC interface being used by the client and server.
5. An exception condition (such as division by zero) occurs while executing the specified remote procedure.
6. The specified remote procedure is executed successfully.

Obviously, in the first five cases, an unsuccessful reply has to be sent to the client with the reason for failure in processing the request and a successful reply has to be sent in the sixth case with the result of procedure execution. Therefore the format of a successful reply message and an unsuccessful reply message is normally slightly different. A typical RPC reply message format for successful and unsuccessful replies may be of the form shown in Figure 4.4.

The message identifier field of a reply message is the same as that of its corresponding call message so that a reply message can be properly matched with its call message. The message type field is properly set to indicate that it is a reply message. For a successful reply, the reply status field is normally set to zero and is followed by the field containing the result of procedure execution. For an unsuccessful reply, the reply status field is either set to 1 or to a nonzero value to indicate failure. In the latter case, the value of the reply status field indicates the type of error. However, in either case, normally a short statement describing the reason for failure is placed in a separate field following the reply status field.

Since RPC protocols are generally independent of transport protocols, it is not possible for an RPC protocol designer to fix the maximum length of call and reply

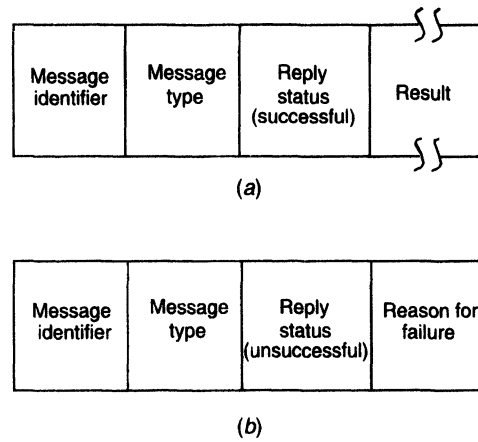


Fig. 4.4 A typical RPC reply message format: (*a*) a successful reply message format; (*b*) an unsuccessful reply message format.

messages. Therefore, for a distributed application to work for a group of transports, it is important for the distributed application developers to ensure that their RPC call and reply messages do not exceed the maximum length specified by any of the transports of the concerned group.

4.7 MARSHALING ARGUMENTS AND RESULTS

Implementation of remote procedure calls involves the transfer of arguments from the client process to the server process and the transfer of results from the server process to the client process. These arguments and results are basically language-level data structures (program objects), which are transferred in the form of message data between the two computers involved in the call. We have seen in the previous chapter that transfer of message data between two computers requires encoding and decoding of the message data. For RPCs this operation is known as *marshaling* and basically involves the following actions:

1. Taking the arguments (of a client process) or the result (of a server process) that will form the message data to be sent to the remote process.
2. Encoding the message data of step 1 above on the sender's computer. This encoding process involves the conversion of program objects into a stream form that is suitable for transmission and placing them into a message buffer.
3. Decoding of the message data on the receiver's computer. This decoding process involves the reconstruction of program objects from the message data that was received in stream form.

In order that encoding and decoding of an RPC message can be performed successfully, the order and the representation method (tagged or untagged) used to

marshal arguments and results must be known to both the client and the server of the RPC. This provides a degree of type safety between a client and a server because the server will not accept a call from a client until the client uses the same interface definition as the server. Type safety is of particular importance to servers since it allows them to survive against corrupt call requests.

The marshaling process must reflect the structure of all types of program objects used in the concerned language. These include primitive types, structured types, and user-defined types. Marshaling procedures may be classified into two groups:

1. Those provided as a part of the RPC software. Normally marshaling procedures for scalar data types, together with procedures to marshal compound types built from the scalar ones, fall in this group.

2. Those that are defined by the users of the RPC system. This group contains marshaling procedures for user-defined data types and data types that include pointers. For example, in Concurrent CLU, developed for use in the Cambridge Distributed Computer System [Bacon and Hamilton 1987], for user-defined types, the type definition must contain procedures for marshaling.

A good RPC system should always generate in-line marshaling code for every remote call so that the users are relieved of the burden of writing their own marshaling procedures. However, practically it is difficult to achieve this goal because of the unacceptable large amounts of code that may have to be generated for handling all possible data types.

4.8 SERVER MANAGEMENT

In RPC-based applications, two important issues that need to be considered for server management are server implementation and server creation.

4.8.1 Server Implementation

Based on the style of implementation used, servers may be of two types: stateful and stateless.

Stateful Servers

A stateful server maintains clients' state information from one remote procedure call to the next. That is, in case of two subsequent calls by a client to a stateful server, some state information pertaining to the service performed for the client as a result of the first call execution is stored by the server process. These clients' state information is subsequently used at the time of executing the second call.

For example, let us consider a server for byte-stream files that allows the following operations on files:

Open (*filename, mode*): This operation is used to open a file identified by *filename* in the specified *mode*. When the server executes this operation, it creates an entry for this file in a *file-table* that it uses for maintaining the file state information of all the open files. The file state information normally consists of the identifier of the file, the open mode, and the current position of a nonnegative integer pointer, called the *read-write pointer*. When a file is opened, its *read-write pointer* is set to zero and the server returns to the client a file identifier (*fid*), which is used by the client for subsequent accesses to that file.

Read (*fid, n, buffer*): This operation is used to get *n* bytes of data from the file identified by *fid* into the buffer named *buffer*. When the server executes this operation, it returns to the client *n* bytes of file data starting from the byte currently addressed by the *read-write pointer* and then increments the *read-write pointer* by *n*.

Write (*fid, n, buffer*): On execution of this operation, the server takes *n* bytes of data from the specified *buffer*, writes it into the file identified by *fid* at the byte position currently addressed by the *read-write pointer*, and then increments the *read-write pointer* by *n*.

Seek (*fid, position*): This operation causes the server to change the value of the *read-write pointer* of the file identified by *fid* to the new value specified as *position*.

Close (*fid*): This statement causes the server to delete from its *file-table* the file state information of the file identified by *fid*.

The file server mentioned above is stateful because it maintains the current state information for a file that has been opened for use by a client. Therefore, as shown in Figure 4.5, after opening a file, if a client makes two subsequent **Read** (*fid, 100, buf*) calls, the first call will return the first 100 bytes (bytes 0–99) and the second call will return the next 100 bytes (bytes 100–199).

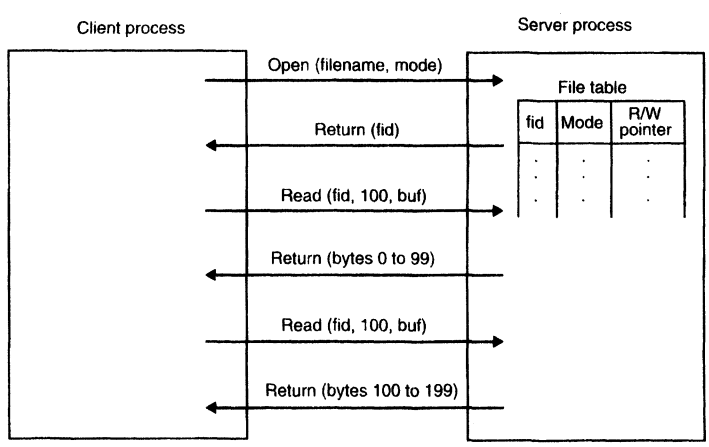


Fig. 4.5 An example of a stateful file server.

Stateless Servers

A stateless server does not maintain any client state information. Therefore every request from a client must be accompanied with all the necessary parameters to successfully carry out the desired operation. For example, a server for byte stream files that allows the following operations on files is stateless.

Read (*filename, position, n, buffer*): On execution of this operation, the server returns to the client *n* bytes of data of the file identified by *filename*. The returned data is placed in the buffer named *buffer*. The value of actual number of bytes read is also returned to the client. The position within the file from where to begin reading is specified as the *position* parameter.

Write (*filename, position, n, buffer*): When the server executes this operation, it takes *n* bytes of data from the specified *buffer* and writes it into the file identified by *filename*. The *position* parameter specifies the byte position within the file from where to start writing. The server returns to the client the actual number of bytes written.

As shown in Figure 4.6, this file server does not keep track of any file state information resulting from a previous operation. Therefore if a client wishes to have similar effect as that in Figure 4.5, the following two *Read* operations must be carried out:

Read (*filename, 0, 100, buf*)
Read (*filename, 100, 100, buf*)

Notice that in this case the client has to keep track of the file state information.

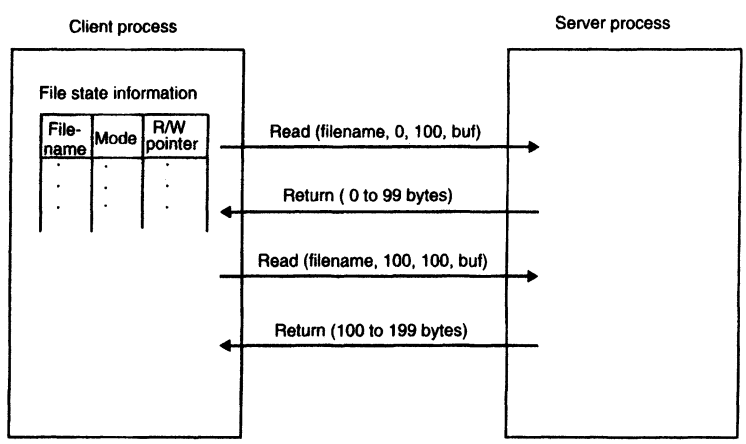


Fig. 4.6 An example of a stateless file server.

Why Stateless Servers?

From the description of stateful and stateless servers, readers might have observed that stateful servers provide an easier programming paradigm because they relieve the clients from the task of keeping track of state information. In addition, stateful servers are typically more efficient than stateless servers. Therefore, the obvious question that arises is why should stateless servers be used at all.

The use of stateless servers in many distributed applications is justified by the fact that stateless servers have a distinct advantage over stateful servers in the event of a failure. For example, with stateful servers, if a server crashes and then restarts, the state information that it was holding may be lost and the client process might continue its task unaware of the crash, producing inconsistent results. Similarly, when a client process crashes and then restarts its task, the server is left holding state information that is no longer valid but cannot easily be withdrawn. Therefore, the client of a stateful server must be properly designed to detect server crashes so that it can perform necessary error-handling activities. On the other hand, with stateless servers, a client has to only retry a request until the server responds; it does not need to know that the server has crashed or that the network temporarily went down. Therefore, stateless servers, which can be constructed around repeatable operations, make crash recovery very easy.

Both stateless and stateful servers have their own advantages and disadvantages. The choice of using a stateless or a stateful server is purely application dependent. Therefore, distributed application system designers must carefully examine the positive and negative aspects of both approaches for their applications before making a choice.

4.8.2 Server Creation Semantics

In RPC, the remote procedure to be executed as a result of a remote procedure call made by a client process lies in a server process that is totally independent of the client process. Independence here means that the client and server processes have separate lifetimes, they normally run on separate machines, and they have their own address spaces. Since a server process is independent of a client process that makes a remote procedure call to it, server processes may either be created and installed before their client processes or be created on a demand basis. Based on the time duration for which RPC servers survive, they may be classified as instance-per-call servers, instance-per-transaction/session servers, or persistent servers.

Instance-per-Call Servers

Servers belonging to this category exist only for the duration of a single call. A server of this type is created by RPCRuntime on the server machine only when a call message arrives. The server is deleted after the call has been executed.

This approach for server creation is not commonly used because of the following problems associated with it:

- The servers of this type are stateless because they are killed as soon as they have serviced the call for which they were created. Therefore, any state that has to be

preserved across server calls must be taken care of by either the client process or the supporting operating system. The involvement of the operating system in maintaining intercall state information will make the remote procedure calls expensive. On the other hand, if the intercall state information is maintained by the client process, the state information must be passed to and from the server with each call. This will lead to the loss of data abstraction across the client-server interface, which will ultimately result in loss of attractiveness of the RPC mechanism to the programmers.

- When a distributed application needs to successively invoke the same type of server several times, this approach appears more expensive, since resource (memory space to provide buffer space and control structures) allocation and deallocation has to be done many times. Therefore, the overhead involved in server creation and destruction dominates the cost of remote procedure calls.

Instance-per-Session Servers

Servers belonging to this category exist for the entire session for which a client and a server interact. Since a server of this type exists for the entire session, it can maintain intercall state information, and the overhead involved in server creation and destruction for a client-server session that involves a large number of calls is also minimized.

In this method, normally there is a server manager for each type of service. All these server managers are registered with the binding agent (binding agent mechanism for binding a client and a server is described later in this chapter). When a client contacts the binding agent, it specifies the type of service needed and the binding agent returns the address of the server manager of the desired type to the client. The client then contacts the concerned server manager, requesting it to create a server for it. The server manager then spawns a new server and passes back its address to the client. The client now directly interacts with this server for the entire session. This server is exclusively used by the client for which it was created and is destroyed when the client informs back to the server manager of the corresponding type that it no longer needs that server.

A server of this type can retain useful state information between calls and so can present a cleaner, more abstract interface to its clients. Note that a server of this type only services a single client and hence only has to manage a single set of state information.

Persistent Servers

A persistent server generally remains in existence indefinitely. Moreover, we saw that the servers of the previous two types cannot be shared by two or more clients because they are exclusively created for a particular client on demand. Unlike them, a persistent server is usually shared by many clients.

Servers of this type are usually created and installed before the clients that use them. Each server independently exports its service by registering itself with the binding agent. When a client contacts the binding agent for a particular type of service, the binding agent selects a server of that type either arbitrarily or based on some in-built policy (such as the

minimum number of clients currently bound to it) and returns the address of the selected server to the client. The client then directly interacts with that server.

Note that a persistent server may be simultaneously bound to several clients. In this case, the server interleaves requests from a number of clients and thus has to concurrently manage several sets of state information. If a persistent server is shared by multiple clients, the remote procedure that it offers must be designed so that interleaved or concurrent requests from different clients do not interfere with each other.

Persistent servers may also be used for improving the overall performance and reliability of the system. For this, several persistent servers that provide the same type of service may be installed on different machines to provide either load balancing or some measure of resilience to failure.

4.9 PARAMETER-PASSING SEMANTICS

The choice of parameter-passing semantics is crucial to the design of an RPC mechanism. The two choices are call-by-value and call-by-reference.

4.9.1 Call-by-Value

In the *call-by-value* method, all parameters are copied into a message that is transmitted from the client to the server through the intervening network. This poses no problems for simple compact types such as integers, counters, small arrays, and so on. However, passing larger data types such as multidimensional arrays, trees, and so on, can consume much time for transmission of data that may not be used. Therefore this method is not suitable for passing parameters involving voluminous data.

An argument in favor of the high cost incurred in passing large parameters by value is that it forces the users to be aware of the expense of remote procedure calls for large-parameter lists. In turn, the users are forced to carefully consider their design of the interface needed between client and server to minimize the passing of unnecessary data. Therefore, before choosing RPC parameter-passing semantics, it is important to carefully review and properly design the client-server interfaces so that parameters become more specific with minimal data being transmitted.

4.9.2 Call-by-Reference

Most RPC mechanisms use the call-by-value semantics for parameter passing because the client and the server exist in different address spaces, possibly even on different types of machines, so that passing pointers or passing parameters *by reference* is meaningless. However, a few RPC mechanisms do allow passing of parameters by reference in which pointers to the parameters are passed from the client to the server. These are usually closed systems, where a single address space is shared by all processes in the system. For example, distributed systems having distributed shared-memory mechanisms (described in Chapter 5) can allow passing of parameters by reference.

In an object-based system that uses the RPC mechanism for object invocation, the call-by-reference semantics is known as *call-by-object-reference*. This is because in an object-based system, the value of a variable is a reference to an object, so it is this reference (the object name) that is passed in an invocation.

Emerald [Black et al. 1986, 1987] designers observed that the use of a call-by-object-reference mechanism in distributed systems presents a potentially serious performance problem because on a remote invocation access by the remote operation to an argument is likely to cause an additional remote invocation. Therefore to avoid many remote references, Emerald supports a new parameter-passing mode that is known as *call-by-move*. In call-by-move, a parameter is passed by reference, as in the method of call-by-object-reference, but at the time of the call, the parameter object is moved to the destination node (site of the callee). Following the call, the argument object may either return to the caller's node or remain at the callee's node (these two modes are known as *call-by-visit* and *call-by-move*, respectively).

Obviously, the use of the call-by-move mode for parameter passing requires that the underlying system supports mobile objects that can be moved from one node to another. Emerald objects are mobile.

Notice that call-by-move does not change the parameter-passing semantics, which is still call-by-object-reference. Therefore call-by-move is basically convenient and optimizes performance. This is because call-by-move could be emulated as a two-step operation:

- First move each call-by-move parameter object to the invokee's node.
- Then invoke the object.

However, performing the moves separately would cause multiple messages to be sent across the network. Thus, providing call-by-move as a parameter-passing mode allows packaging of the argument objects in the same network packet as the invocation message, thereby reducing the network traffic and message count.

Although call-by-move reduces the cost of references made by the invokee, it increases the cost of the invocation itself. If the parameter object is mutable and shared, it also increases the cost of references by the invoker [Black et al. 1987].

4.10 CALL SEMANTICS

In RPC, the caller and the callee processes are possibly located on different nodes. Thus it is possible for either the caller or the callee node to fail independently and later to be restarted. In addition, failure of communication links between the caller and the callee nodes is also possible. Therefore, the normal functioning of an RPC may get disrupted due to one or more of the following reasons:

- The call message gets lost.
- The response message gets lost.

- The callee node crashes and is restarted.
- The caller node crashes and is restarted.

Some element of a caller's node that is involved in the RPC must contain necessary code to handle these failures. Obviously, the code for the caller's procedure should not be forced to deal with these failures. Therefore, the failure-handling code is generally a part of RPCRuntime. The call semantics of an RPC system that determines how often the remote procedure may be executed under fault conditions depends on this part of the RPCRuntime code. This part of the code may be designed to provide the flexibility to the application programmers to select from different possible call semantics supported by an RPC system. The different types of call semantics used in RPC systems are described below.

4.10.1 Possibly or May-Be Call Semantics

This is the weakest semantics and is not really appropriate to RPC but is mentioned here for completeness. In this method, to prevent the caller from waiting indefinitely for a response from the callee, a timeout mechanism is used. That is, the caller waits until a predetermined timeout period and then continues with its execution. Therefore the semantics does not guarantee anything about the receipt of the call message or the procedure execution by the caller. This semantics may be adequate for some applications in which the response message is not important for the caller and where the application operates within a local area network having a high probability of successful transmission of messages.

4.10.2 Last-One Call Semantics

This call semantics is similar to the one described in Section 3.9 and illustrated with an example in Figure 3.10. It uses the idea of retransmitting the call message based on timeouts until a response is received by the caller. That is, the calling of the remote procedure by the caller, the execution of the procedure by the callee, and the return of the result to the caller will eventually be repeated until the result of procedure execution is received by the caller. Clearly, the results of the last executed call are used by the caller, although earlier (abandoned) calls may have had side effects that survived the crash. Hence this semantics is called last-one semantics.

Last-one semantics can be easily achieved in the way described above when only two processors (nodes) are involved in the RPC. However, achieving last-one semantics in the presence of crashes turns out to be tricky for nested RPCs that involve more than two processors (nodes) [Bal et al. 1989]. For example, suppose process $P1$ of node $N1$ calls procedure $F1$ on node $N2$, which in turn calls procedure $F2$ on node $N3$. While the process on $N3$ is working on $F2$, node $N1$ crashes. Node $N1$'s processes will be restarted, and $P1$'s call to $F1$ will be repeated. The second invocation of $F1$ will again call procedure $F2$ on node $N3$. Unfortunately, node $N3$ is totally unaware of node $N1$'s crash. Therefore procedure $F2$ will be executed twice on node $N3$ and $N3$ may return the results of the two executions of $F2$ in any order, possibly violating last-one semantics.

The basic difficulty in achieving last-one semantics in such cases is caused by orphan calls. An *orphan call* is one whose parent (caller) has expired due to a node crash. To achieve last-one semantics, these orphan calls must be terminated before restarting the crashed processes. This is normally done either by waiting for them to finish or by tracking them down and killing them ("*orphan extermination*"). As this is not an easy job, other weaker semantics have been proposed for RPC.

4.10.3 Last-of-Many Call Semantics

This is similar to the last-one semantics except that the orphan calls are neglected [Bal et al. 1989]. A simple way to neglect orphan calls is to use call identifiers to uniquely identify each call. When a call is repeated, it is assigned a new call identifier. Each response message has the corresponding call identifier associated with it. A caller accepts a response only if the call identifier associated with it matches with the identifier of the most recently repeated call; otherwise it ignores the response message.

4.10.4 At-Least-Once Call Semantics

This is an even weaker call semantics than the last-of-many call semantics. It just guarantees that the call is executed one or more times but does not specify which results are returned to the caller. It can be implemented simply by using timeout-based retransmissions without caring for the orphan calls. That is, for nested calls, if there are any orphan calls, it takes the result of the first response message and ignores the others, whether or not the accepted response is from an orphan.

4.10.5 Exactly-Once Call Semantics

This is the strongest and the most desirable call semantics because it eliminates the possibility of a procedure being executed more than once no matter how many times a call is retransmitted. The last-one, last-of-many, and at-least-once call semantics cannot guarantee this. The main disadvantage of these cheap semantics is that they force the application programmer to design idempotent interfaces that guarantee that if a procedure is executed more than once with the same parameters, the same results and side effects will be produced. For example, let us consider the example given in [Wilbur and Bacarisse 1987] for reading and writing a record in a sequential file of fixed-length records. For reading successive records from such a file, a suitable procedure is

```
ReadNextRecord(Filename)
```

Ignoring initialization and end-of-file effects, each execution of this procedure will return the next record from the specified file. Obviously, this procedure is not idempotent because multiple execution of this procedure will return the successive records, which is not desirable for duplicate calls that are retransmitted due to the loss of response messages. This happens because in the implementation of this procedure, the server needs

to keep track of the current record position for each client that has opened the file for accessing. Therefore to design an idempotent interface for reading the next record from the file, it is important that each client keeps track of its own current record position and the server is made stateless, that is, no client state should be maintained on the server side. Based on this idea, an idempotent procedure for reading the next record from a sequential file is

```
ReadRecordN(Filename, N)
```

which returns the *N*th record from the specified file. In this case, the client has to correctly specify the value of *N* to get the desired record from the file.

However, not all nonidempotent interfaces can be so easily transformed to an idempotent form. For example, consider the following procedure for appending a new record to the same sequential file:

```
AppendRecord(Filename, Record)
```

It is clearly not idempotent since repeated execution will add further copies of the same record to the file. This interface may be converted into an idempotent interface by using the following two procedures instead of the one defined above:

```
GetLastRecordNo(Filename)
WriteRecordN(Filename, Record, N)
```

The first procedure returns the record number of the last record currently in the file, and the second procedure writes a record at a specified position in the file. Now, for appending a record, the client will have to use the following two procedures:

```
Last = GetLastRecordNo(Filename)
WriteRecordN(Filename, Record, Last)
```

For exactly-once semantics, the programmer is relieved of the burden of implementing the server procedure in an idempotent manner because the call semantics itself takes care of executing the procedure only once. As already described in Section 3.9 and illustrated with an example in Figure 3.12, the implementation of exactly-once call semantics is based on the use of timeouts, retransmissions, call identifiers with the same identifier for repeated calls, and a reply cache associated with the callee.

4.11 COMMUNICATION PROTOCOLS FOR RPCs

Different systems, developed on the basis of remote procedure calls, have different IPC requirements. Based on the needs of different systems, several communication protocols have been proposed for use in RPCs. A brief description of these protocols is given below.

4.11.1 The Request Protocol

This protocol is also known as the *R* (request) protocol [Spector 1982]. It is used in RPCs in which the called procedure has nothing to return as the result of procedure execution and the client requires no confirmation that the procedure has been executed. Since no acknowledgment or reply message is involved in this protocol, only one message per call is transmitted (from client to server) (Fig. 4.7). The client normally proceeds immediately after sending the request message as there is no need to wait for a reply message. The protocol provides may-be call semantics and requires no retransmission of request messages.

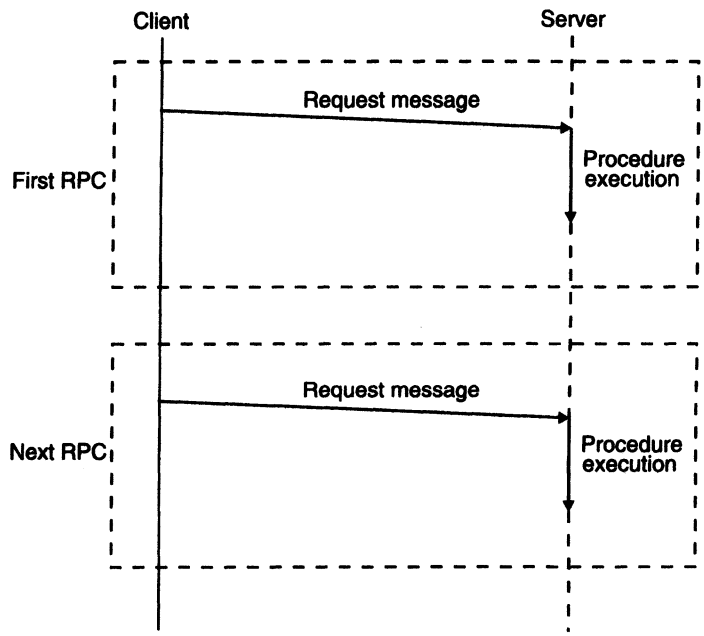


Fig. 4.7 The request (R) protocol.

An RPC that uses the R protocol is called *asynchronous RPC*. An asynchronous RPC helps in improving the combined performance of both the client and the server in those distributed applications in which the client does not need a reply to each request. Client performance is improved because the client is not blocked and can immediately continue to do other work after making the call. On the other hand, server performance is improved because the server need not generate and send any reply for the request. One such application is a distributed window system. A distributed window system, such as X-11 [Davison et al. 1992], is programmed as a server, and application programs wishing to display items in windows on a display screen are its clients. To display items in a window, a client normally sends many requests (each request containing a relatively small amount

of information for a small change in the displayed information) to the server one after another without waiting for a reply for each of these requests because it does not need replies for the requests.

Notice that for an asynchronous RPC, the RPCRuntime does not take responsibility for retrying a request in case of communication failure. This means that if an unreliable datagram transport protocol such as UDP is used for the RPC, the request message could be lost without the client's knowledge. Applications using asynchronous RPC with unreliable transport protocol must be prepared to handle this situation. However, if a reliable, connection-oriented transport protocol such as TCP is used for the RPC, there is no need to worry about retransmitting the request message because it is delivered reliably in this case.

Asynchronous RPCs with unreliable transport protocol are generally useful for implementing periodic update services. For example, a time server node in a distributed system may send time synchronization messages every T seconds to other nodes using the asynchronous RPC facility. In this case, even if a message is lost, the correct time is transmitted in the next message. Each node can keep track of the last time it received an update message to prevent it from missing too many update messages. A node that misses too many update messages can send a special request message to the time server node to get a reliable update after some maximum amount of time.

4.11.2 The Request/Reply Protocol

This protocol is also known as the *RR* (request/reply) protocol [Spector 1982]. It is useful for the design of systems involving simple RPCs. A *simple RPC* is one in which all the arguments as well as all the results fit in a single packet buffer and the duration of a call and the interval between calls are both short (less than the transmission time for a packet between the client and server) [Birrell and Nelson 1984]. The protocol is based on the idea of using implicit acknowledgment to eliminate explicit acknowledgment messages. Therefore in this protocol:

- A server's reply message is regarded as an acknowledgment of the client's request message.
- A subsequent call packet from a client is regarded as an acknowledgment of the server's reply message of the previous call made by that client.

The exchange of messages between a client and a server in the RR protocol is shown in Figure 4.8. Notice from the figure that the protocol involves the transmission of only two packets per call (one in each direction).

The RR protocol in its basic form does not possess failure-handling capabilities. Therefore to take care of lost messages, the timeouts-and-retries technique is normally used along with the RR protocol. In this technique, a client retransmits its request message if it does not receive the response message before a predetermined timeout period elapses. Obviously, if duplicate request messages are not filtered out, the RR protocol, compounded with this technique, provides at-least-once call semantics.

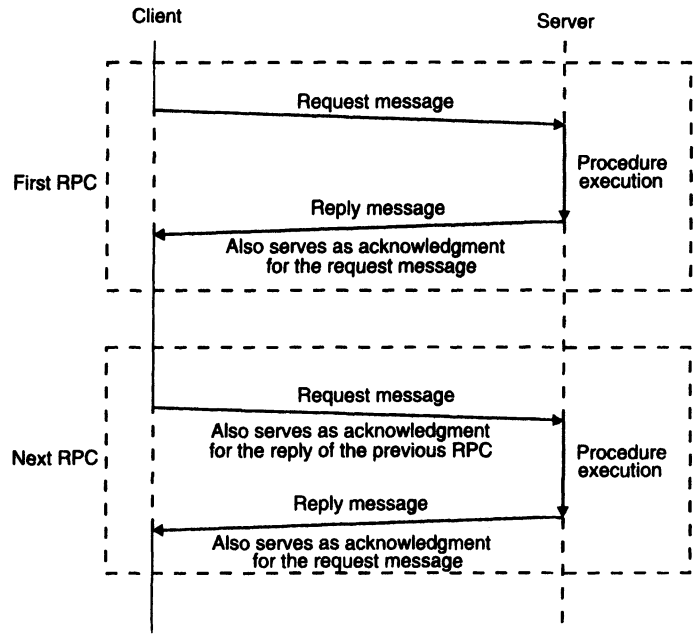


Fig. 4.8 The request/reply (RR) protocol.

However, servers can support exactly-once call semantics by keeping records of the replies in a reply cache that enables them to filter out duplicate request messages and to retransmit reply messages without the need to reprocess a request. The details of this technique were given in Section 3.9.

4.11.3 The Request/Reply/Acknowledge-Reply Protocol

This protocol is also known as the *RRA* (request/reply/acknowledge-reply) protocol [Spector 1982]. The implementation of exactly-once call semantics with RR protocol requires the server to maintain a record of the replies in its reply cache. In situations where a server has a large number of clients, this may result in servers needing to store large quantities of information. In some implementations, servers restrict the quantity of such data by discarding it after a limited period of time. However, this approach is not fully reliable because sometimes it may lead to the loss of those replies that have not yet been successfully delivered to their clients. To overcome this limitation of the RR protocol, the RRA protocol is used, which requires clients to acknowledge the receipt of reply messages. The server deletes an information from its reply cache only after receiving an acknowledgment for it from the client. As shown in Figure 4.9, the RRA protocol involves the transmission of three messages per call (two from the client to the server and one from the server to the client).

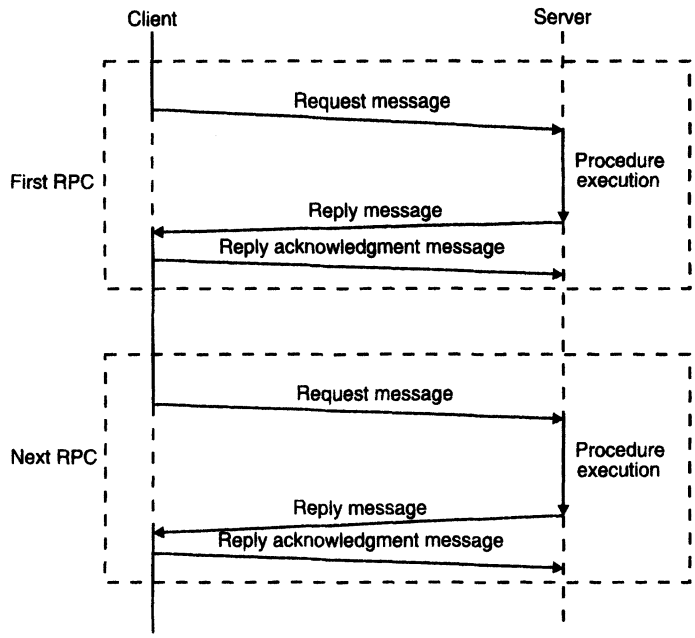


Fig. 4.9 The request/reply/acknowledge-reply (RRA) protocol.

In the RRA protocol, there is a possibility that the acknowledgment message may itself get lost. Therefore implementation of the RRA protocol requires that the unique message identifiers associated with request messages must be ordered. Each reply message contains the message identifier of the corresponding request message, and each acknowledgment message also contains the same message identifier. This helps in matching a reply with its corresponding request and an acknowledgment with its corresponding reply. A client acknowledges a reply message only if it has received the replies to all the requests previous to the request corresponding to this reply. Thus an acknowledgment message is interpreted as acknowledging the receipt of all reply messages corresponding to the request messages with lower message identifiers. Therefore the loss of an acknowledgment message is harmless.

4.12 COMPLICATED RPCs

Birrell and Nelson [1984] categorized the following two types of RPCs as complicated:

1. RPCs involving long-duration calls or large gaps between calls
2. RPCs involving arguments and/or results that are too large to fit in a single-datagram packet

Different protocols are used for handling these two types of complicated RPCs.

4.12.1 RPCs Involving Long-Duration Calls or Large Gaps between Calls

One of the following two methods may be used to handle complicated RPCs that belong to this category [Birrell and Nelson 1984]:

1. *Periodic probing of the server by the client.* In this method, after a client sends a request message to a server, it periodically sends a probe packet to the server, which the server is expected to acknowledge. This allows the client to detect a server's crash or communication link failures and to notify the corresponding user of an exception condition. The message identifier of the original request message is included in each probe packet. Therefore, if the original request is lost, in reply to a probe packet corresponding to that request message, the server intimates the client that the request message corresponding to the probe packet has not been received. Upon receipt of such a reply from the server, the client retransmits the original request.

2. *Periodic generation of an acknowledgment by the server.* In this method, if a server is not able to generate the next packet significantly sooner than the expected retransmission interval, it spontaneously generates an acknowledgment. Therefore for a long-duration call, the server may have to generate several acknowledgments, the number of acknowledgments being directly proportional to the duration of the call. If the client does not receive either the reply for its request or an acknowledgment from the server within a predetermined timeout period, it assumes that either the server has crashed or communication link failure has occurred. In this case, it notifies the concerned user of an exception condition.

4.12.2 RPCs Involving Long Messages

In some RPCs, the arguments and/or results are too large to fit in a single-datagram packet. For example, in a file server, quite large quantities of data may be transferred as input arguments to the *write* operation or as results to the *read* operation. A simple way to handle such an RPC is to use several physical RPCs for one logical RPC. Each physical RPC transfers an amount of data that fits in a single-datagram packet. This solution is inefficient due to a fixed amount of overhead involved with each RPC independent of the amount of data sent.

Another method of handling complicated RPCs of this category is to use multidatagram messages. In this method, a long RPC argument or result is fragmented and transmitted in multiple packets. To improve communication performance, a single acknowledgment packet is used for all the packets of a multidatagram message. In this case, the same approach that was described in Section 3.9 is used to keep track of lost and out-of-sequence packets of a multidatagram RPC message.

Some RPC systems are limited to small sizes. For example, the Sun Microsystem's RPC is limited to 8 kilobytes. Therefore, in these systems, an RPC involving messages larger than the allowed limit must be handled by breaking it up into several physical RPCs.

4.13 CLIENT-SERVER BINDING

It is necessary for a client (actually a client stub) to know the location of a server before a remote procedure call can take place between them. The process by which a client becomes associated with a server so that calls can take place is known as *binding*. From the application level's point of view, the model of binding is that servers "export" operations to register their willingness to provide service and clients' "import" operations, asking the RPCRuntime system to locate a server and establish any state that may be needed at each end [Bershad et al. 1987]. The client-server binding process involves proper handling of several issues:

1. How does a client specify a server to which it wants to get bound?
2. How does the binding process locate the specified server?
3. When is it proper to bind a client to a server?
4. Is it possible for a client to change a binding during execution?
5. Can a client be simultaneously bound to multiple servers that provide the same service?

These binding issues are described below.

4.13.1 Server Naming

The specification by a client of a server with which it wants to communicate is primarily a naming issue. For RPC, Birrell and Nelson [1984] proposed the use of interface names for this purpose. An *interface name* has two parts—a *type* and an *instance*. Type specifies the interface itself and instance specifies a server providing the services within that interface. For example, there may be an interface of type *file_server*, and there may be several instances of servers providing file service. When a client is not concerned with which particular server of an interface services its request, it need not specify the instance part of the interface name.

The type part of an interface usually also has a version number field to distinguish between old and new versions of the interface that may have different sets of procedures or the same set of procedures with different parameters. It is inevitable in the course of distributed application programming that an application needs to be updated after a given version has been released. The use of a version number field allows old and new versions of a distributed application to coexist. One would hope that the new version of an interface would eventually replace all the old versions of the interface. However, experience has shown that it is always better to maintain backward compatibility with old versions of the software because someone might still be using one of the old versions.

According to Birrell and Nelson [1984], the interface name semantics are based on an arrangement between the exporter and the importer. Therefore, interface names are created by the users. They are not dictated by the RPC package. The RPC package only dictates the means by which an importer uses the interface name to locate an exporter.

4.13.2 Server Locating

The interface name of a server is its unique identifier. Thus when a client specifies the interface name of a server for making a remote procedure call, the server must be located before the client's request message can be sent to it. This is primarily a locating issue and any locating mechanism (locating mechanisms are described in Chapter 10) can be used for this purpose. The two most commonly used methods are as follows:

1. *Broadcasting.* In this method, a message to locate the desired server is broadcast to all the nodes from the client node. The nodes on which the desired server is located return a response message. Note that the desired server may be replicated on several nodes so the client node will receive a response from all these nodes. Normally, the first response that is received at the client's node is given to the client process and all subsequent responses are discarded.

This method is easy to implement and is suitable for use for small networks. However, the method is expensive for large networks because of the increase in message traffic due to the involvement of all the nodes in broadcast processing. Therefore the second method, which is based on the idea of using a name server, is generally used for large networks.

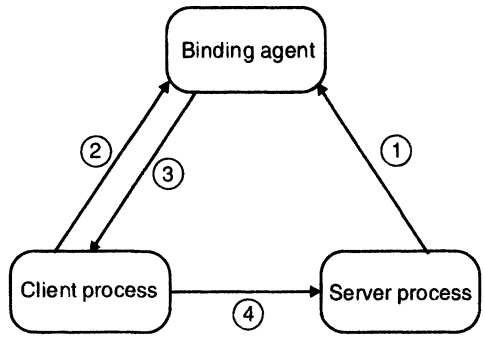
2. *Binding agent.* A binding agent is basically a name server used to bind a client to a server by providing the client with the location information of the desired server. In this method, a binding agent maintains a binding table, which is a mapping of a server's interface name to its locations. All servers register themselves with the binding agent as a part of their initialization process. To register with the binding agent, a server gives the binder its identification information and a handle used to locate it. The handle is system dependent and might be an Ethernet address, an IP address, an X.500 address, a process identifier containing a node number and port number, or something else. A server can also deregister with the binding agent when it is no longer prepared to offer service. The binding agent can also poll the servers periodically, automatically deregistering any server that fails to respond.

To locate a server, a client contacts the binding agent. If the server is registered with the binding agent, it returns the handle (location information) of the server to the client. The method is illustrated in Figure 4.10.

The binding agent's location is known to all nodes. This is accomplished by using either a fixed address for the binding agent that is known to all nodes or a broadcast message to locate the binding agent when a node is booted. In either case, when the binding agent is relocated, a message is sent to all nodes informing the new location of the binding agent.

A binding agent interface usually has three primitives: (a) *register* is used by a server to register itself with the binding agent, (b) *deregister* is used by a server to deregister itself with the binding agent, and (c) *lookup* is used by a client to locate a server.

The binding agent mechanism for locating servers has several advantages. First, the method can support multiple servers having the same interface type so that any of the available servers may be used to service a client's request. This helps to achieve a degree of fault tolerance. Second, since all bindings are done by the binding agent, when multiple



① The server registers itself with the binding agent.

② The client requests the binding agent for the server's location.

③ The binding agent returns the server's location information to the client.

④ The client calls the server.

Fig. 4.10 The binding agent mechanism for locating a server in case of RPC.

servers provide the same service, the clients can be spread evenly over the servers to balance the load. Third, the binding mechanism can be extended to allow servers to specify a list of users who may use its service, in which case the binding agent would refuse to bind those clients to the servers who are not authorized to use its service.

However, the binding agent mechanism has drawbacks. The overhead involved in binding clients to servers is large and becomes significant when many client processes are short lived. Moreover, in addition to any functional requirements, a binding agent must be robust against failures and should not become a performance bottleneck. Distributing the binding function among several binding agents and replicating information among them can satisfy both these criteria. Unfortunately, replication often involves extra overhead of keeping the multiple replicas consistent. Therefore, the functionality offered by many binding agents is lower than might be hoped for.

4.13.3 Binding Time

A client may be bound to a server at compile time, at link time, or at call time [Goscinski 1991].

Binding at Compile Time

In this method, the client and server modules are programmed as if they were intended to be linked together. For example, the server's network address can be compiled into the client code by the programmer and then it can be found by looking up the server's name in a file.

The method is extremely inflexible in the sense that if the server moves or the server is replicated or the interface changes, all client programs using the server will have to be found and recompiled. However, the method is useful in certain limited cases. For example, it may be used in an application whose configuration is expected to remain static for a fairly long time.

Binding at Link Time

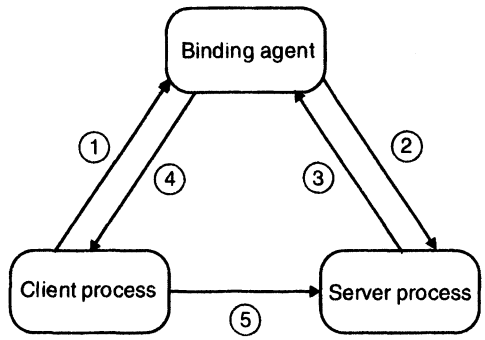
In this method, a server process exports its service by registering itself with the binding agent as part of its initialization process. A client then makes an import request to the binding agent for the service before making a call. The binding agent binds the client and the server by returning to the client the server's handle (details that are necessary for making a call to the server). Calls can take place once the client has received the server's handle. The server's handle is cached by the client to avoid contacting the binding agent for subsequent calls to be made to the same server. Due to the overhead involved in contacting the binding agent, this method is suitable for those situations in which a client calls a server several times once it is bound to it.

Binding at Call Time

In this method, a client is bound to a server at the time when it calls the server for the first time during its execution. A commonly used approach for binding at call time is the *indirect call* method. As shown in Figure 4.11, in this method, when a client calls a server for the first time, it passes the server's interface name and the arguments of the RPC call to the binding agent. The binding agent looks up the location of the target server in its binding table, and on behalf of the client it sends an RPC call message to the target server, including in it the arguments received from the client. When the target server returns the results to the binding agent, the binding agent returns this result to the client along with the target server's handle so that the client can subsequently call the target server directly.

4.13.4 Changing Bindings

The flexibility provided by a system to change bindings dynamically is very useful from a reliability point of view. Binding is a connection establishment between a client and a server. The client or server of a connection may wish to change the binding at some instance of time due to some change in the system state. For example, a client willing to get a request serviced by any one of the multiple servers for that service may be programmed to change a binding to another server of the same type when a call to the already connected server fails. Similarly, the server of a binding may want to alter the binding and connect the client to another server in situations such as when the service needs to move to another node or a new version of the server is installed. When a binding is altered by the concerned server, it is



① The client process passes the server's interface name and the arguments of the RPC call to the binding agent.

② The binding agent sends an RPC call message to the server, including in it the arguments received from the client.

③ The server returns the result of request processing to the binding agent.

④ The binding agent returns this result to the client along with the server's handle.

⑤ Subsequent calls are sent directly from the client process to the server process.

Fig. 4.11 Illustrating binding at call time by the method of indirect call.

important to ensure that any state data held by the server is no longer needed or can be duplicated in the replacement server. For example, when a file server has to be replaced with a new one, either it must be replaced when no files are open or the state of all the open files must be transferred from the old server to the new one as a part of the replacement process.

4.13.5 Multiple Simultaneous Bindings

In a system, a service may be provided by multiple servers. We have seen that, in general, a client is bound to a single server of the several servers of the same type. However, there may be situations when it is advantageous for a client to be bound simultaneously to all or multiple servers of the same type. Logically, a binding of this sort gives rise to multicast communication because when a call is made, all the servers bound to the client for that service will receive and process the call. For example, a client may wish to update multiple copies of a file that is replicated at several nodes. For this, the client can be bound simultaneously to file servers of all those nodes where a replica of the file is located.

4.14 EXCEPTION HANDLING

We saw in Figure 4.4 that when a remote procedure cannot be executed successfully, the server reports an error in the reply message. An RPC also fails when a client cannot contact the server of the RPC. An RPC system must have an effective exception-handling mechanism for reporting such failures to clients. One approach to do this is to define an exception condition for each possible error type and have the corresponding exception raised when an error of that type occurs, causing the exception-handling procedure to be called and automatically executed in the client's environment. This approach can be used with those programming languages that provide language constructs for exception handling. Some such programming languages are ADA, CLU [Liskov et al. 1981], and Modula-3 [Nelson 1991, Harbinson 1992]. In C language, signal handlers can be used for the purpose of exception handling.

However, not every language has an exception-handling mechanism. For example, Pascal does not have such a mechanism. RPC systems designed for use with such languages generally use the method provided in conventional operating systems for exception handling. One such method is to return a well-known value to the process, making a system call to indicate failure and to report the type of error by storing a suitable value in a variable in the environment of the calling program. For example, in UNIX the value −1 is used to indicate failure, and the type of error is reported in the global variable *errno*. In an RPC, a return value indicating an error is used both for errors due to failure to communicate with the server and errors reported in the reply message from the server. The details of the type of error is reported by storing a suitable value in a global variable in the client program. This approach suffers from two main drawbacks. First, it requires the client to test every return value. Second, it is not general enough because a return value used to indicate failure may be a perfectly legal value to be returned by a procedure. For example, if the value −1 is used to indicate failure, this value is also the return value of a procedure call with arguments −5 and 4 to a procedure for getting the sum of two numbers.

4.15 SECURITY

Some implementations of RPC include facilities for client and server authentication as well as for providing encryption-based security for calls. For example, in [Birrell and Nelson 1984], callers are given a guarantee of the identity of the callee, and vice versa, by using the authentication service of Grapevine [Birrell et al. 1982]. For full end-to-end encryption of calls and results, the federal data encryption standard [DES 1977] is used in [Birrell and Nelson 1984]. The encryption techniques provide protection from eavesdropping (and conceal patterns of data) and detect attempts at modification, replay, or creation of calls.

In other implementations of RPC that do not include security facilities, the arguments and results of RPC are readable by anyone monitoring communications between the caller and the callee. Therefore in this case, if security is desired, the user must implement his or her own authentication and data encryption mechanisms. When designing an application, the user should consider the following security issues related with the communication of messages:

- Is the authentication of the server by the client required?
- Is the authentication of the client by the server required when the result is returned?
- Is it all right if the arguments and results of the RPC are accessible to users other than the caller and the callee?

These and other security issues are described in detail in Chapter 11.

4.16 SOME SPECIAL TYPES OF RPCs

4.16.1 Callback RPC

In the usual RPC protocol, the caller and callee processes have a client-server relationship. Unlike this, the callback RPC facilitates a peer-to-peer paradigm among the participating processes. It allows a process to be both a client and a server.

Callback RPC facility is very useful in certain distributed applications. For example, remotely processed interactive applications that need user input from time to time or under special conditions for further processing require this type of facility. As shown in Figure 4.12, in such applications, the client process makes an RPC to the concerned server process, and during procedure execution for the client, the server process makes a callback RPC to the client process. The client process takes necessary action based on the server's request and returns a reply for the callback RPC to the server process. On receiving this reply, the server resumes the execution of the procedure and finally returns the result of the initial call to the client. Note that the server may make several callbacks to the client before returning the result of the initial call to the client process.

The ability for a server to call its client back is very important, and care is needed in the design of RPC protocols to ensure that it is possible. In particular, to provide callback RPC facility, the following are necessary:

- Providing the server with the client's handle
- Making the client process wait for the callback RPC
- Handling callback deadlocks

Commonly used methods to handle these issues are described below.

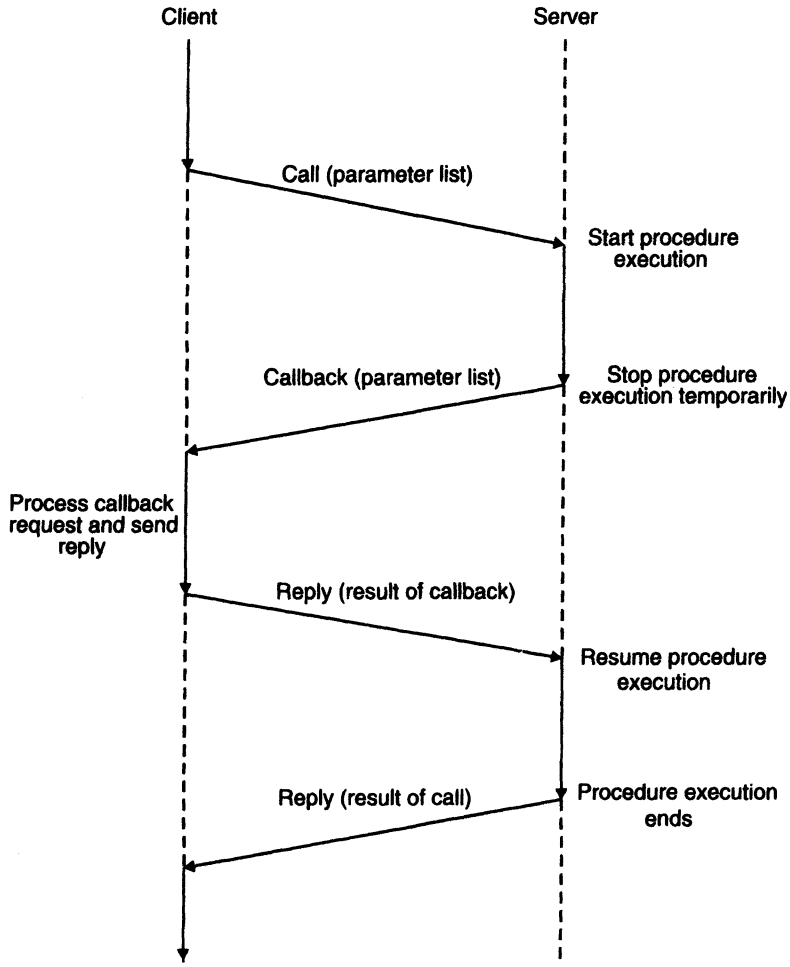


Fig. 4.12 The callback RPC.

Providing the Server with the Client's Handle

The server must have the client's handle to call the client back. The client's handle uniquely identifies the client process and provides enough information to the server for making a call to it. Typically, the client process uses a transient program number for the callback service and exports the callback service by registering its program number with the binding agent. The program number is then sent as a part of the RPC request to the server. To make a callback RPC, the server initiates a normal RPC request to the client using the given program number. Instead of having the client just send the server the program number, it could also send its handle, such as the port number. The client's handle could then be used by the server to directly communicate

with the client and would save an RPC to the binding agent to get the client's handle.

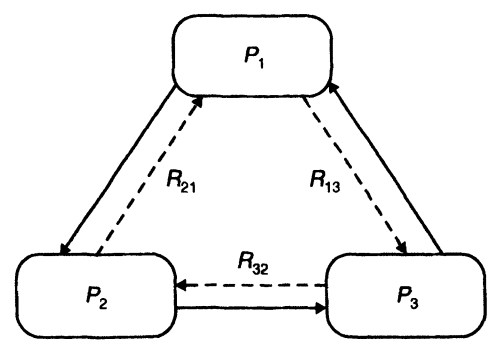
Making the Client Process Wait for the Callback RPC

The client process must be waiting for the callback so that it can process the incoming RPC request from the server and also to ensure that a callback RPC from the server is not mistaken to be the reply of the RPC call made by the client process. To wait for the callback, a client process normally makes a call to a *svc-routine*. The *svc-routine* waits until it receives a request and then dispatches the request to the appropriate procedure.

Handling Callback Deadlocks

In callback RPC, since a process may play the role of either a client or a server, callback deadlocks can occur. For example, consider the most simple case in which a process P_1 makes an RPC call to a process P_2 and waits for a reply from P_2. In the meantime, process P_2 makes an RPC call to another process P_3 and waits for a reply from P_3. In the meantime, process P_3 makes an RPC call to process P_1 and waits for a reply from P_1. But P_1 cannot process P_3's request until its request to P_2 has been satisfied, and P_2 cannot process P_1's request until its request to P_3 has been satisfied, and P_3 cannot process P_2's request until its request to P_1 has been satisfied. As shown in Figure 4.13, a situation now exists where P_1 is waiting for a reply from P_2, which is waiting for a reply from P_3, which is waiting for a reply from P_1. The result is that none of the three processes can have their request satisfied, and hence all three will continue to wait indefinitely. In effect, a callback deadlock has occurred due to the interdependencies of the three processes.

While using a callback RPC, care must be taken to handle callback deadlock situations. Various methods for handling deadlocks are described in Chapter 6.



P_1 is waiting for R_{21} (reply from P_2 to P_1)
P_2 is waiting for R_{32} (reply from P_3 to P_2)
P_3 is waiting for R_{13} (reply from P_1 to P_3)

Fig. 4.13 An example of a callback deadlock in case of callback RPC mechanism.

4.16.2 Broadcast RPC

The RPC-based IPC is normally of the one-to-one type, involving a single client process and a single server process. However, we have seen in the previous chapter that for performance reasons several highly parallel distributed applications require the communication system to provide the facility of broadcast and multicast communication. The RPC-based IPC mechanisms normally support broadcast RPC facility for such applications. In broadcast RPC, a client's request is broadcast on the network and is processed by all the servers that have the concerned procedure for processing that request. The client waits for and receives numerous replies.

A broadcast RPC mechanism may use one of the following two methods for broadcasting a client's request:

1. The client has to use a special broadcast primitive to indicate that the request message has to be broadcasted. The request is sent to the binding agent, which forwards the request to all the servers registered with it. Note that in this method, since all broadcast RPC messages are sent to the binding agent, only services that register themselves with their binding agent are accessible via the broadcast RPC mechanism.

2. The second method is to declare broadcast ports. A network port of each node is connected to a broadcast port. A network port of a node is a queuing point on that node for broadcast messages. The client of the broadcast RPC first obtains a binding for a broadcast port and then broadcasts the RPC message by sending the message to this port. Note that the same primitive may be used for both unicast and broadcast RPCs. Moreover, unlike the first method, this method also has the flexibility of being used for multicast RPC in which the RPC message is sent only to a subset of the available servers. For this, the port declaration mechanism should have the flexibility to associate only a subset of the available servers to a newly declared multicast port.

Since a broadcast RPC message is sent to all the nodes of a network, a reply is expected from each node. As already described in the previous chapter, depending on the degree of reliability desired, the client process may wait for zero, one, *m*-out-of-*n*, or all the replies. In some implementations, servers that support broadcast RPC typically respond only when the request is successfully processed and are silent in the face of errors. Such systems normally use some type of timeout-based retransmission mechanism for improving the reliability of the broadcast RPC protocol. For example, in SunOS, the broadcast RPC protocol transmits the broadcast and waits for 4 seconds before retransmitting the request. It then waits for 6 seconds before retransmitting the request and continues to increment the amount of time to wait by 2 seconds until the timeout period becomes greater than 14 seconds. Therefore, in the worst case, the request is broadcast six times and the total wait time is 54 seconds ($4+6+8+10+12+14$). In SunOS, the broadcast RPC uses unreliable, packet-based protocol for broadcasting the request, and so the routine retransmits the broadcast requests by default. Increasing the amount of time between retransmissions is known as a *back-off algorithm*. The use of a back-off algorithm for timeout-based retransmissions helps in reducing the load on the physical network and computers involved.

4.16.3 Batch-Mode RPC

Batch-mode RPC is used to queue separate RPC requests in a transmission buffer on the client side and then send them over the network in one batch to the server. This helps in the following two ways:

1. It reduces the overhead involved in sending each RPC request independently to the server and waiting for a response for each request.
2. Applications requiring higher call rates (50–100 remote calls per second) may not be feasible with most RPC implementations. Such applications can be accommodated with the use of batch-mode RPC.

However, batch-mode RPC can be used only with those applications in which a client has many RPC requests to send to a server and the client does not need any reply for a sequence of requests. Therefore, the requests are queued on the client side, and the entire queue of requests is flushed to the server when one of the following conditions becomes true:

1. A predetermined interval elapses.
2. A predetermined number of requests have been queued.
3. The amount of batched data exceeds the buffer size.
4. A call is made to one of the server's procedures for which a result is expected. From a programming standpoint, the semantics of such a call (nonqueueing RPC request) should be such that the server can distinguish it from the queued requests and send a reply for it to the client.

The flushing out of queued requests in cases 1, 2, and 3 happens independent of a nonqueuing RPC request and is not noticeable by the client.

Obviously, the queued messages should be sent reliably. Hence, a batch-mode RPC mechanism requires reliable transports such as TCP. Moreover, although the batch-mode optimization retains syntactic transparency, it may produce obscure timing-related effects where other clients are accessing the server simultaneously.

4.17 RPC IN HETEROGENEOUS ENVIRONMENTS

Heterogeneity is an important issue in the design of any distributed application because typically the more portable an application, the better. When designing an RPC system for a heterogeneous environment, the three common types of heterogeneity that need to be considered are as follows:

1. *Data representation.* Machines having different architectures may use different data representations. For example, integers may be represented with the most significant byte at the low-byte address in one machine architecture and at the high-byte address in another machine architecture. Similarly, integers may be represented in 1's complement

notation in one machine architecture and in 2's complement notation in another machine architecture. Floating-point representations may also vary between two different machine architectures. Therefore, an RPC system for a heterogeneous environment must be designed to take care of such differences in data representations between the architectures of client and server machines of a procedure call.

2. *Transport protocol.* For better portability of applications, an RPC system must be independent of the underlying network transport protocol. This will allow distributed applications using the RPC system to be run on different networks that use different transport protocols.

3. *Control protocol.* For better portability of applications, an RPC system must also be independent of the underlying network control protocol that defines control information in each transport packet to track the state of a call.

The most commonly used approach to deal with these types of heterogeneity while designing an RPC system for a heterogeneous environment is to delay the choices of data representation, transport protocol, and control protocol until bind time. In conventional RPC systems, all these decisions are made when the RPC system is designed. That is, the binding mechanism of an RPC system for a heterogeneous environment is considerably richer in information than the binding mechanism used by a conventional RPC system. It includes mechanisms for determining which data conversion software (if any conversion is needed), which transport protocol, and which control protocol should be used between a specific client and server and returns the correct procedures to the stubs as result parameters of the binding call. These binding mechanism details are transparent to the users. That is, application programs never directly access the component structures of the binding mechanism; they deal with bindings only as atomic types and acquire and discard them via the calls of the RPC system.

Some RPC systems designed for heterogeneous environments are the HCS (Heterogeneous Computer Systems) RPC (called HRPC) [Bershad et al. 1987], the DCE SRC (System Research Center) Firefly RPC [Schroeder and Burrows 1990], Matchmaker [Jones et al. 1985], and Horus [Gibbons 1987].

4.18 LIGHTWEIGHT RPC

The *Lightweight Remote Procedure Call* (*LRPC*) was introduced by Bershad et al. [1990] and integrated into the Taos operating system of the DEC SRC Firefly microprocessor workstation. The description below is based on the material in their paper [Bershad et al. 1990].

As mentioned in Chapter 1, based on the size of the kernel, operating systems may be broadly classified into two categories—monolithic-kernel operating systems and microkernel operating systems. Monolithic-kernel operating systems have a large, monolithic kernel that is insulated from user programs by simple hardware boundaries. On the other hand, in microkernel operating systems, a small kernel provides only primitive operations and most of the services are provided by user-level servers. The servers are

usually implemented as processes and can be programmed separately. Each server forms a component of the operating system and usually has its own address space. As compared to the monolithic-kernel approach, in this approach services are provided less efficiently because the various components of the operating system have to use some form of IPC to communicate with each other. The advantages of this approach include simplicity and flexibility. Due to modular structure, microkernel operating systems are simple and easy to design, implement, and maintain.

In the microkernel approach, when different components of the operating system have their own address spaces, the address space of each component is said to form a *domain*, and messages are used for all interdomain communication. In this case, the communication traffic in operating systems are of two types [Bershad et al. 1990]:

1. *Cross-domain*, which involves communication between domains on the same machine
2. *Cross-machine*, which involves communication between domains located on separate machines

The LRPC is a communication facility designed and optimized for cross-domain communications.

Although conventional RPC systems can be used for both cross-domain and cross-machine communications, Bershad et al. observed that the use of conventional RPC systems for cross-domain communications, which dominate cross-machine communications, incurs an unnecessarily high cost. This cost leads system designers to coalesce weakly related components of microkernel operating systems into a single domain, trading safety and performance. Therefore, the basic advantages of using the microkernel approach are not fully exploited. Based on these observations, Bershad et al. designed the LRPC facility for cross-domain communications, which has better performance than conventional RPC systems. Nonetheless, LRPC is safe and transparent and represents a viable communication alternative for microkernel operating systems.

To achieve better performance than conventional RPC systems, the four techniques described below are used by LRPC.

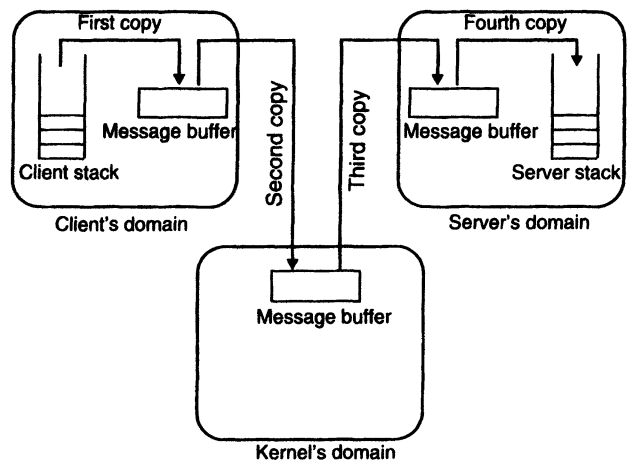
4.18.1 Simple Control Transfer

Whenever possible, LRPC uses a control transfer mechanism that is simpler than that used in conventional RPC systems. For example, it uses a special threads scheduling mechanism, called handoff scheduling (details of the threads and handoff scheduling mechanism are given in Chapter 8), for direct context switch from the client thread to the server thread of an LRPC. In this mechanism, when a client calls a server's procedure, it provides the server with an argument stack and its own thread of execution. The call causes a trap to the kernel. The kernel validates the caller, creates a call linkage, and dispatches the client's thread directly to the server domain, causing the server to start executing immediately. When the called procedure completes, control and results return through the kernel back to the point of the client's call. In contrast to this, in conventional RPC implementations, context switching between the client and server threads of an RPC is slow because the client thread and the server thread are fixed in their own domains, signaling one another at a rendezvous, and the

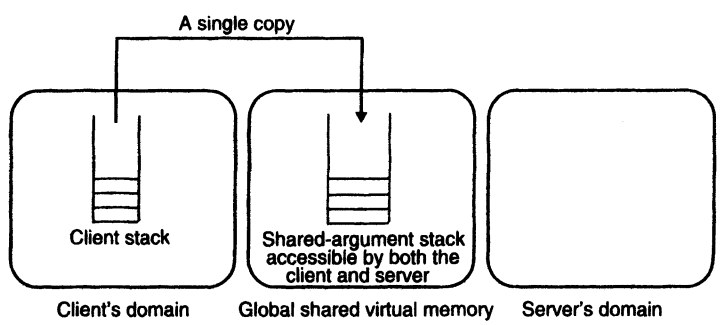
scheduler must manipulate system data structures to block the client's thread and then select one of the server's threads for execution.

4.18.2 Simple Data Transfer

In an RPC, arguments and results need to be passed between the client and server domains in the form of messages. As compared to traditional RPC systems, LRPC reduces the cost of data transfer by performing fewer copies of the data during its transfer from one domain to another. For example, let us consider the path taken by a procedure's argument during a traditional cross-domain RPC. As shown in Figure 4.14(*a*), an argument in this case normally has to be copied four times:



(*a*)



(*b*)

Fig. 4.14 Data transfer mechanisms in traditional cross-domain RPC and LRPC. (*a*) The path taken by a procedure's argument during a traditional cross-domain RPC involves four copy operations. (*b*) The path taken by a procedure's argument during LRPC involves a single-copy operation.

1. From the client's stack to the RPC message
2. From the message in the client domain to the message in the kernel domain
3. From the message in the kernel domain to the message in the server domain
4. From the message in the server domain to the server's stack

To simplify this data transfer operation, LRPC uses a shared-argument stack that is accessible to both the client and the server. Therefore, as shown in Figure 4.14(*b*), the same argument in an LRPC can be copied only once—from the client's stack to the shared-argument stack. The server uses the argument from the argument stack. Pairwise allocation of argument stacks enables LRPC to provide a private channel between the client and server and also allows the copying of parameters and results as many times as are necessary to ensure correct and safe operation.

4.18.3 Simple Stubs

The distinction between cross-domain and cross-machine calls is usually made transparent to the stubs by lower levels of the RPC system. This results in an interface and execution path that are general but infrequently needed.

The use of a simple model of control and data transfer in LRPC facilitates the generation of highly optimized stubs. Every procedure has a call stub in the client's domain and an entry stub in the server's domain. A three-layered communication protocol is defined for each procedure in an LRPC interface:

1. End to end, described by the calling conventions of the programming language and architecture
2. Stub to stub, implemented by the stubs themselves
3. Domain to domain, implemented by the kernel

To reduce the cost of interlayer crossings, LRPC stubs blur the boundaries between the protocol layers. For example, at the time of transfer of control, the kernel associates execution stacks with the initial call frame expected by the called server's procedure and directly invokes the corresponding procedure's entry in the server's domain. No intermediate message examination or dispatching is done, and the server stub starts executing the procedure by directly branching to the procedure's first instruction. Notice that with this arrangement a simple LRPC needs only one formal procedure call (into the client stub) and two returns (one out of the server procedure and one out of the client stub).

4.18.4 Design for Concurrency

When the node of the client and server processes of an LRPC has multiple processors with a shared memory, special mechanisms are used to achieve higher call throughput and lower call latency than is possible on a single-processor node. Throughput is increased by avoiding needless lock contention by minimizing the use of shared-data structures on the

critical domain transfer path. On the other hand, latency is reduced by reducing context-switching overhead by caching domains on idle processors. This is basically a generalization of the idea of decreasing operating system latency by caching recently blocked threads on idle processors to reduce wake-up latency. Instead of threads, LRPC caches domains so that any thread that needs to run in the context of an idle domain can do so quickly, not just the thread that ran there most recently.

Based on the performance evaluation made by Bershad et al. [1990], it was found that LRPC achieves a factor-of-three performance improvement over more traditional approaches. Thus LRPC reduces the cost of cross-domain communication to nearly the lower bound imposed by conventional hardware.

4.19 OPTIMIZATIONS FOR BETTER PERFORMANCE

As with any software design, performance is an issue in the design of a distributed application. The description of LRPC shows some optimizations that may be adopted for better performance of distributed applications using RPC. Some other optimizations that may also have significant payoff when adopted for designing RPC-based distributed applications are described below.

4.19.1 Concurrent Access to Multiple Servers

Although one of the benefits of RPC is its synchronization property, many distributed applications can benefit from concurrent access to multiple servers. One of the following three approaches may be used for providing this facility:

1. The use of threads (described in Chapter 8) in the implementation of a client process where each thread can independently make remote procedure calls to different servers. This method requires that the addressing in the underlying protocol is rich enough to provide correct routing of responses.

2. Another method is the use of the early reply approach [Wilbur and Bacarisse 1987]. As shown in Figure 4.15, in this method a call is split into two separate RPC calls, one passing the parameters to the server and the other requesting the result. In reply to the first call, the server returns a tag that is sent back with the second call to match the call with the correct result. The client decides the time delay between the two calls and carries out other activities during this period, possibly making several other RPC calls. A drawback of this method is that the server must hold the result of a call until the client makes a request for it. Therefore, if the request for results is delayed, it may cause congestion or unnecessary overhead at the server.

3. The third approach, known as the call buffering approach, was proposed by Gimson [1985]. In this method, clients and servers do not interact directly with each other. They interact indirectly via a call buffer server. To make an RPC call, a client sends its call request to the call buffer server, where the request parameters together with the name of the server and the client are buffered. The client can then perform other activities until it

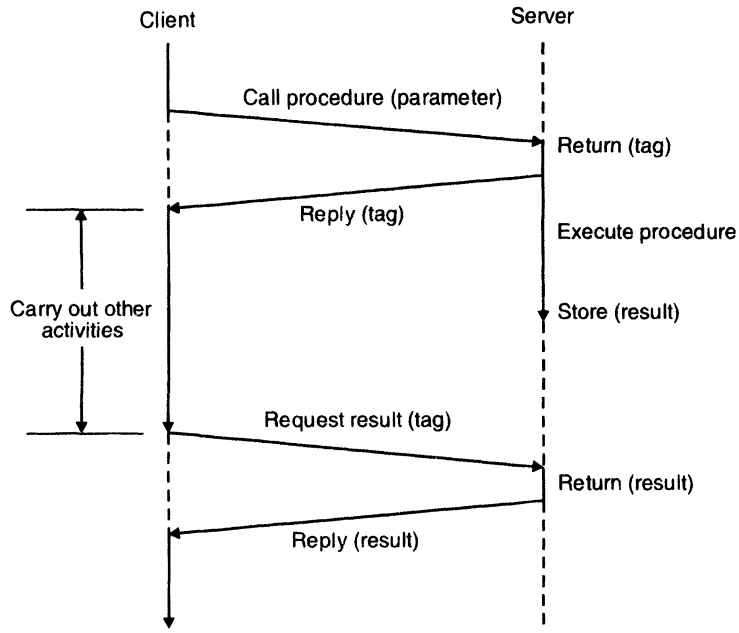


Fig. 4.15 The early reply approach for providing the facility of concurrent access to multiple servers.

needs the result of the RPC call. When the client reaches a state in which it needs the result, it periodically polls the call buffer server to see if the result of the call is available, and if so, it recovers the result. On the server side, when a server is free, it periodically polls the call buffer server to see if there is any call for it. If so, it recovers the call request, executes it, and makes a call back to the call buffer server to return the result of execution to the call buffer server. The method is illustrated in Figure 4.16.

A variant of this approach is used in the Mercury communication system developed at MIT [Liskov and Shrira 1988] for supporting asynchronous RPCs. The Mercury communication system has a new data type called *promise* that is created during an RPC call and is given a type corresponding to those of the results and exceptions of the remote procedure. When the results arrive, they are stored in the appropriate promise, from where the caller claims the results at a time suitable to it. Therefore, after making a call, a caller can continue with other work and subsequently pick up the results of the call from the appropriate promise.

A promise is in one of two states—blocked or ready. It is in a blocked state from the time of creation to the time the results of the call arrive, whereupon it enters the ready state. A promise in the ready state is immutable.

Two operations (*ready* and *claim*) are provided to allow a caller to check the status of the promise for the call and to claim the results of the call from it. The *ready* operation is used to test the status (blocked/ready) of the promise. It returns true or false according to whether the promise is ready or blocked. The *claim* operation is

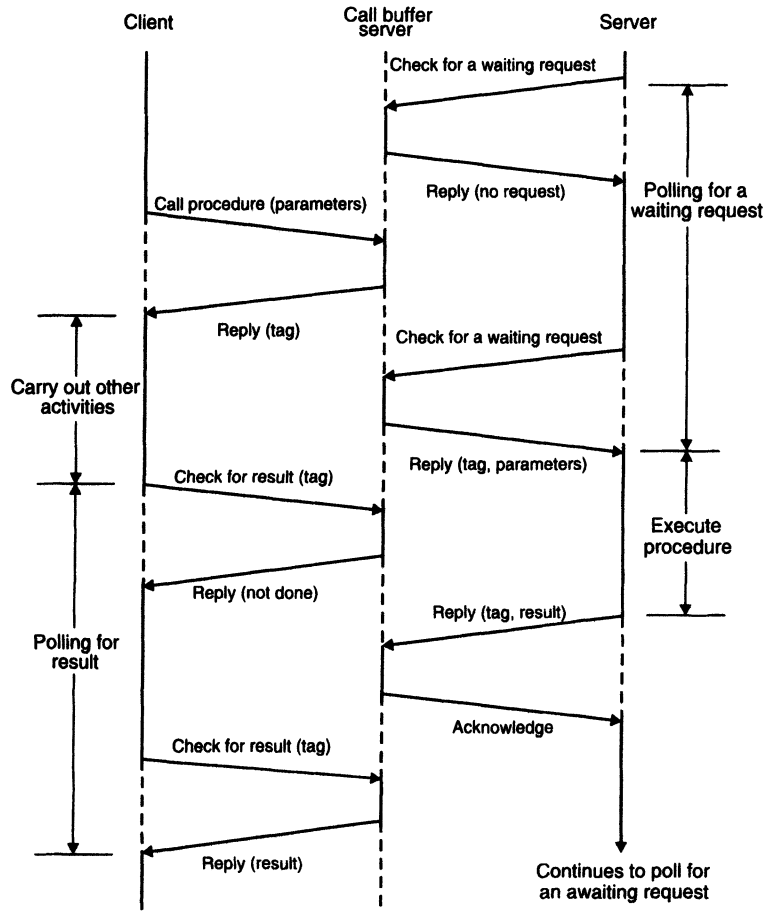


Fig. 4.16 The call buffering approach for providing the facility of concurrent access to multiple servers.

used to obtain the results of the call from the promise. The *claim* operation blocks the caller until the promise is ready, whereupon it returns the results of the call. Therefore, if the caller wants to continue with other work until the promise becomes ready, it can periodically check the status of the promise by using the *ready* operation.

4.19.2 Serving Multiple Requests Simultaneously

The following types of delays are commonly encountered in RPC systems:

1. Delay caused while a server waits for a resource that is temporarily unavailable. For example, during the course of a call execution, a server might have to wait for accessing a shared file that is currently locked elsewhere.

2. A delay can occur when a server calls a remote function that involves a considerable amount of computation to complete or involves a considerable transmission delay.

For better performance, good RPC implementations must have mechanisms to allow the server to accept and process other requests, instead of being idle while waiting for the completion of some operation. This requires that a server be designed in such a manner that it can service multiple requests simultaneously. One method to achieve this is to use the approach of a multiple-threaded server with dynamic threads creation facility for server implementation (details of this approach are given in Chapter 8).

4.19.3 Reducing Per-Call Workload of Servers

Numerous client requests can quickly affect a server's performance when the server has to do a lot of processing for each request. Thus, to improve the overall performance of an RPC system, it is important to keep the requests short and the amount of work required by a server for each request low. One way of accomplishing this improvement is to use stateless servers and let the clients keep track of the progression of their requests sent to the servers. This approach sounds reasonable because, in most cases, the client portion of an application is really in charge of the flow of information between a client and a server.

4.19.4 Reply Caching of Idempotent Remote Procedures

The use of a reply cache to achieve exactly-once semantics in nonidempotent remote procedures has already been described. However, a reply cache can also be associated with idempotent remote procedures for improving a server's performance when it is heavily loaded. When client requests to a server arrive at a rate faster than the server can process the requests, a backlog develops, and eventually client requests start timing out and the clients resend the requests, making the problem worse. In such a situation, the reply cache helps because the server has to process a request only once. If a client resends its request, the server just sends the cached reply.

4.19.5 Proper Selection of Timeout Values

To deal with failure problems, timeout-based retransmissions are necessary in distributed applications. An important issue here is how to choose the timeout value. A "too small" timeout value will cause timers to expire too often, resulting in unnecessary retransmissions. On the other hand, a "too large" timeout value will cause a needlessly long delay in the event that a message is actually lost. In RPC systems, servers are likely to take varying amounts of time to service individual requests, depending on factors such as server load, network routing, and network congestion. If clients continue to retry sending those requests for which replies have not yet been received, the server loading and network congestion problem will become worse. To prevent this situation, proper selection of timeout values is important. One method to handle this issue is to use some sort of back-off strategy of exponentially increasing timeout values.

4.19.6 Proper Design of RPC Protocol Specification

For better performance, the protocol specification of an RPC system must be properly designed so as to minimize the amount of data that has to be sent over the network and the frequency at which it is sent. Reducing the amount of data to be transferred helps in two ways: It requires less time to encode and decode the data and it requires less time to transmit the data over the network. Several existing RPC systems use TCP/IP or UDP/IP as the basic protocol because they are easy to use and fit in well with existing UNIX systems and networks such as the Internet. This makes it straightforward to write clients and servers that run on UNIX systems and standard networks. However, the use of a standard general-purpose protocol for RPC generally leads to poor performance because general-purpose protocols have many features to deal with different problems in different situations. For example, packets in the IP suite (to which TCP/IP and UDP/IP belong) have in total 13 header fields, of which only 3 are useful for an RPC—the source and destination addresses and the packet length. However, several of these header fields, such as those dealing with fragmentation and checksum, have to be filled in by the sender and verified by the receiver to make them legal IP packets. Some of these fields, such as the checksum field, are time consuming to compute. Therefore, for better performance, an RPC system should use a specialized RPC protocol. Of course, a new protocol for this purpose has to be designed from scratch, implemented, tested, and embedded in existing systems, so it requires considerably more work.

4.20 CASE STUDIES: SUN RPC, DCE RPC

Many RPC systems have been built and are in use today. Notable ones include the Cedar RPC system [Birrell and Nelson 1984], Courier in the Xerox NS family of protocols [Xerox Corporation 1981], the Eden system [Almes et al. 1985], the CMU Spice system [Jones et al. 1985], Sun RPC [Sun Microsystems 1985], Argus [Liskov and Scheifler 1983], Arjuna [Shrivastava et al. 1991], the research system built at HP Laboratories [Gibbons 1987], NobelNet's EZ RPC [Smith 1994], Open Software Foundation's (OSF's) DCE RPC [Rosenberry et al. 1992], which is a descendent of Apollo's Network Computing Architecture (NCA), and the HRPC system developed at the University of Washington [Bershad et al. 1987]. Of these, the best known UNIX RPC system is the Sun RPC. Therefore, the Sun RPC will be described in this section as a case study. In addition, due to the policy used in this book to describe DCE components as case studies, the DCE RPC will also be briefly described.

4.20.1 Sun RPC

Stub Generation

Sun RPC uses the automatic stub generation approach, although users have the flexibility of writing the stubs manually. An application's interface definition is written in an IDL called RPC Language (RPCL). RPCL is an extension of the Sun XDR

language that was originally designed for specifying external data representations. As an example, the interface definition of the stateless file service, described in Section 4.8.1, is given in Figure 4.17. As shown in the figure, an interface definition contains a program number (which is 0 x 20000000 in our example) and a version number of the service (which is 1 in our example), the procedures supported by the service (in our example READ and WRITE), the input and output parameters along with their types for each procedure, and the supporting type definitions. The three values program number (STATELESS_FS_PROG), version number (STATELESS_FS_VERS), and a procedure number (READ or WRITE) uniquely identify a remote procedure. The READ and WRITE procedures are given numbers 1 and 2, respectively. The number 0 is reserved for a null procedure that is automatically generated and is intended to be used to test whether a server is available. Interface definition file names have an extension *.x*. (for example, *StatelessFS.x*).

```
/* Interface definition for a stateless file service (StatelessFS)
   in file StatelessFS.x */
const FILE_NAME_SIZE = 16
const BUFFER_SIZE = 1024

typedef string FileName<FILE_NAME_SIZE>;
typedef long Position;
typedef long Nbytes;

struct Data {
        long n;
        char buffer[BUFFER_SIZE];
};

struct readargs {
        FileName        filename;
        Position        position;
        Nbytes          n;
};

struct writeargs {
        FileName        filename;
        Position        position;
        Data            data;
};

program STATELESS_FS_PROG {
        version STATELESS_FS_VERS {
                        Data        READ (readargs) = 1;
                        Nbytes      WRITE (writeargs) = 2;
        } = 1;
} = 0x20000000;
```

Fig. 4.17 Interface definition for a stateless file service written in RPCL of Sun RPC.

The IDL compiler is called *rpcgen* in Sun RPC. From an interface definition file, *rpcgen* generates the following:

1. A header file that contains definitions of common constants and types defined in the interface definition file. It also contains external declarations for all XDR marshaling and unmarshaling procedures that are automatically generated. The name of the header file is formed by taking the base name of the input file to *rpcgen* and adding a *.h* suffix (for example, *StatelessFS.h*). This file is manually included in client and server program files and automatically included in client stub, server stub, and XDR filters files using *#include*.

2. An XDR filters file that contains XDR marshaling and unmarshaling procedures. These procedures are used by the client and server stub procedures. The name of this file is formed by taking the base name of the input file to *rpcgen* and adding a *_xdr.c* suffix (for example, *StatelessFS_xdr.c*).

3. A client stub file that contains one stub procedure for each procedure defined in the interface definition file. A client stub procedure name is the name of the procedure given in the interface definition, converted to lowercase and with an underscore and the version number appended. For instance, in our example, the client stub procedure names for *READ* and *WRITE* procedures will be *read_1* and *write_1*, respectively. The name of the client stub file is formed by taking the base name of the input file to *rpcgen* and adding a *_clnt.c* suffix (for example, *StatelessFS_clnt.c*).

4. A server stub file that contains the *main* routine, the *dispatch* routine, and one stub procedure for each procedure defined in the interface definition file plus a null procedure.

The *main* routine creates the transport handles and registers the service. The default is to register the program on both the UDP and TCP transports. However, a user can select which transport to use with a command-line option to *rpcgen*.

The *dispatch* routine dispatches incoming remote procedure calls to the appropriate procedure. The name used for the dispatch routine is formed by mapping the program name to lowercase characters and appending an underscore followed by the version number (for example, *stateless_fs_prog_1*).

The name of the server stub file is formed by taking the base name of the input file to *rpcgen* and adding a *_svc.c* suffix (for example, *StatelessFS_svc.c*).

Now using the files generated by *rpcgen*, an RPC application is created in the following manner:

1. The application programmer manually writes the client program and server program for the application. The skeletons of these two programs for our example application of stateless file service are given Figures 4.18 and 4.19, respectively. Notice that the remote procedure names used in these two programs are those of the stub procedures (*read_1* and *write_1*).

```
/* A skeleton of client source program for the stateless file service in file client.c */
#include <stdio.h>
#include <rpc/rpc.h>
#include "StatelessFS.h"

main (argc, argv)
        int argc;
        char **argv;
{
        CLIENT          *client_handle;
        char            *server_host_name = "paris";
        readargs        read_args;
        writeargs       write_args;
        Data            *read_result;
        Nbytes          *write_result;

        client_handle = clnt_create (server_host_name, STATELESS_FS_PROG,
                        STATELESS_FS_VERS, "udp");
                        /* Get a client handle. Creates socket */
        if (client_handle == NULL) {
                        clnt_pcreateerror (server_host_name);
                        return (1); /* Cannot contact server */
        };

        /* Prepare parameters and make an RPC to read procedure */
        read_args.filename = "example";
        read_args.position = 0;
        read_args.n = 500;
        read_result = read_1 (&read_args, client_handle);
        . . .
        . . .
        . . .

        /* Prepare parameters and make an RPC to write procedure */
        write_args.filename = "example";
        write_args.position = 501;
        write_args.data.n = 100;
        /* Statements for putting 100 bytes of data in &write_args.data.buffer */
        write_result = write_1 (&write_args, client_handle);
        . . .
        . . .
        . . .
        clnt_destroy (client_handle);
        /* Destroy the client handle when done. Closes socket */
}
```

Fig. 4.18 A skeleton of client source program for the stateless file service of Figure 4.17.

```
/* A skeleton of server source program for the stateless file service in file server.c */
#include <stdio.h>
#include <rpc/rpc.h>
#include "StatelessFS.h"

/* READ PROCEDURE */
Data    *read_1 (args) /* Input parameters as a single argument */
        readargs        *args;
{
        static Data result; /* Must be declared as static */

        /* Statements for reading args.n bytes from the file args.filename starting
        from position args.position, and for putting the data read in &result.buffer
        and the actual number of bytes read in result.n */

        return (&result); /* Return the result as a single argument */
}

/* WRITE PROCEDURE */
Nbytes  *write_1 (args) /* Input parameters as a single argument */
        writeargs       *args;
{
        static Nbytes result; /* Must be declared as static */

        /* Statements for writing args.data.n bytes of data from the buffer
        &args.data.buffer into the file args.filename starting at position
        args.position */

        /* Statement for putting the actual number of bytes written in result */

        return (&result);
}
```

Fig. 4.19 A skeleton of server source program for the stateless file service of Figure 4.17.

2. The client program file is compiled to get a client object file.
3. The server program file is compiled to get a server object file.
4. The client stub file and the XDR filters file are compiled to get a client stub object file.
5. The server stub file and the XDR filters file are compiled to get a server stub object file.
6. The client object file, the client stub object file, and the client-side RPCRuntime library are linked together to get the client executable file.
7. The server object file, the server stub object file, and the server-side RPCRuntime library are linked together to get the server executable file.

The entire process is summarized in Figure 4.20.

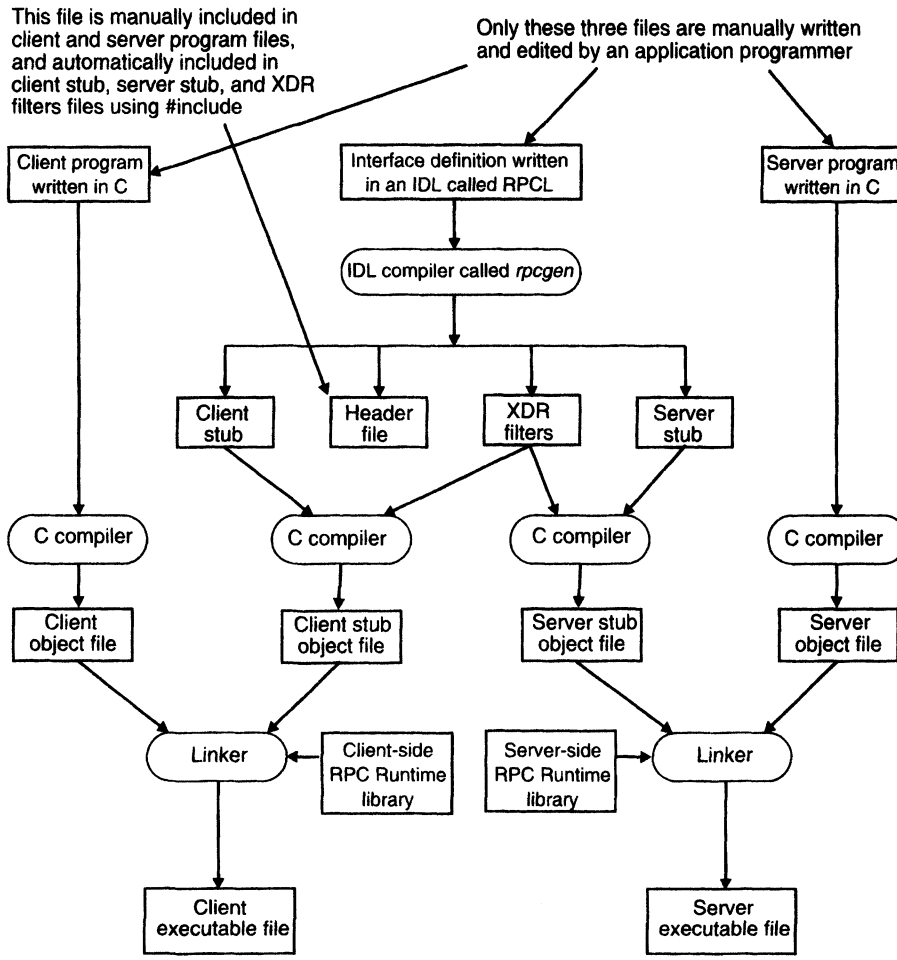


Fig. 4.20 The steps in creating an RPC application in Sun RPC.

Procedure Arguments

In Sun RPC, a remote procedure can accept only one argument and return only one result. Therefore, procedures requiring multiple parameters as input or as output must include them as components of a single structure. This is the reason why the structures *Data* (used as a single output argument to the *READ* procedure), *readargs* (used as a single input argument to the *READ* procedure), and *writeargs* (used as a single input argument to the *WRITE* procedure) have been defined in our example of Figures 4.17–4.19. If a remote procedure does not take an argument, a NULL pointer must still be passed as an argument to the remote procedure. Therefore, a Sun RPC call always has two arguments—the first is a pointer to the single argument of the remote procedure and the second is a pointer to a client handle (see the calls for *read_1* and *write_1* in Fig. 4.18). On the other hand, a

return argument of a procedure is a pointer to the single result. The returned result must be declared as a static variable in the server program because otherwise the value of the returned result becomes undefined when the procedure returns (see the return argument *result* in Fig. 4.19).

Marshaling Arguments and Results

We have seen that Sun RPC allows arbitrary data structures to be passed as arguments and results. Since significant data representation differences can exist between the client computer and the server computer, these data structures are converted to eXternal Data Representation (XDR) and back using marshaling procedures. The marshaling procedures to be used are specified by the user and may be either built-in procedures supplied in the RPCRuntime library or user-defined procedures defined in terms of the built-in procedures. The RPCRuntime library has procedures for marshaling integers of all sizes, characters, strings, reals, and enumerated types.

Since XDR encoding and decoding always occur, even between a client and server of the same architecture, unnecessary overhead is added to the network service for those applications in which XDR encoding and decoding are not needed. In such cases, user-defined marshaling procedures can be utilized. That is, users can write their own marshaling procedures verifying that the architectures of the client and the server machines are the same and, if so, use the data without conversion. If they are not the same, the correct XDR procedures can be invoked.

Call Semantics

Sun RPC supports at-least-once semantics. After sending a request message, the RPCRuntime library waits for a timeout period for the server to reply before retransmitting the request. The number of retries is the total time to wait divided by the timeout period. The total time to wait and the timeout period have default values of 25 and 5 seconds, respectively. These default values can be set to different values by the users. Eventually, if no reply is received from the server within the total time to wait, the RPCRuntime library returns a timeout error.

Client-Server Binding

Sun RPC does not have a networkwide binding service for client-server binding. Instead, each node has a local binding agent called *portmapper* that maintains a database of mapping of all local services (as already mentioned, each service is identified by its program number and version number) and their port numbers. The portmapper runs at a well-known port number on every node.

When a server starts up, it registers its program number, version number, and port number with the local portmapper. When a client wants to do an RPC, it must first find out the port number of the server that supports the remote procedure. For this, the client makes a remote request to the portmapper at the server's host, specifying the program number and version number (see *clnt_create* part of Fig. 4.18). This means that a client

must specify the host name of the server when it imports a service interface. In effect, this means that Sun RPC has no location transparency.

The procedure *clnt_create* is used by a client to import a service interface. It returns a client handle that contains the necessary information for communicating with the corresponding server port, such as the socket descriptor and socket address. The client handle is used by the client to directly communicate with the server when making subsequent RPCs to procedures of the service interface (see RPCs made to *read_1* and *write_1* procedures in Fig. 4.18).

Exception Handling

The RPCRuntime library of Sun RPC has several procedures for processing detected errors. The server-side error-handling procedures typically send a reply message back to the client side, indicating the detected error. However, the client-side error-handling procedures provide the flexibility to choose the error-reporting mechanism. That is, errors may be reported to users either by printing error messages to *stderr* or by returning strings containing error messages to clients.

Security

Sun RPC supports the following three types of authentication (often referred to as *flavors*):

1. *No authentication.* This is the default type. In this case, no attempt is made by the server to check a client's authenticity before executing the requested procedure. Consequently, clients do not pass any authentication parameters in request messages.

2. *UNIX-style authentication.* This style is used to restrict access to a service to a certain set of users. In this case, the *uid* and *gid* of the user running the client program are passed in every request message, and based on this authentication information, the server decides whether to execute the requested procedure or not.

3. *DES-style authentication.* Data Encryption Standard (DES) is an encryption technique described in Chapter 11. In DES-style authentication, each user has a globally unique name called *netname*. The *netname* of the user running the client program is passed in encrypted form in every request message. On the server side, the encrypted *netname* is first decrypted and then the server uses the information in *netname* to decide whether to execute the requested procedure or not.

The DES-style authentication is recommended for users who need more security than UNIX-style authentication. RPCs using DES-style authentication are also referred to as *secure RPC*.

Clients have the flexibility to select any of the above three authentication flavors for an RPC. The type of authentication can be specified when a client handle is created. It is possible to use a different authentication mechanism for different remote procedures within a distributed application by setting the authentication type to the flavor desired before doing the RPC.

The authentication mechanism of Sun RPC is open ended in the sense that in addition to the three authentication types mentioned above users are free to invent and use new authentication types.

Special Types of RPCs

Sun RPC provides support for asynchronous RPC, callback RPC, broadcast RPC, and batch-mode RPC.

Asynchronous RPC is accomplished by setting the timeout value of an RPC to zero and writing the server such that no reply is generated for the request.

To facilitate callback RPC, the client registers the callback service using a transient program number with the local portmapper. The program number is then sent as part of the RPC request to the server. The server initiates a normal RPC request to the client using the given program number when it is ready to do the callback RPC.

A broadcast RPC is directed to the portmapper of all nodes. Each node's portmapper then passes it on to the local service with the given program name. The client picks up any replies one by one.

Batch-mode RPC is accomplished by batching of client calls that require no reply and then sending them in a pipeline to the server over TCP/IP.

Critiques of Sun RPC

In spite of its popularity, some of the criticisms normally made against Sun RPC are as follows:

1. Sun RPC lacks location transparency because a client has to specify the host name of the server when it imports a service interface.
2. The interface definition language of Sun RPC does not allow a general specification of procedure arguments and results. It allows only a single argument and a single result. This requirement forces multiple arguments or return values to be packaged as a single structure.
3. Sun RPC is not transport independent and the transport protocol is limited to either UDP or TCP. However, a transport-independent version of Sun RPC, known as TI-RPC (transport-independent RPC), has been developed by Sun-Soft, Inc. TI-RPC provides a simple and consistent way in which transports can be dynamically selected depending upon user preference and the availability of the transport. Details of TI-RPC can be found in [Khanna 1994].
4. In UDP, Sun RPC messages are limited to 8 kilobytes in length.
5. Sun RPC supports only at-least-once call semantics, which may not be acceptable for some applications.
6. Sun RPC does not have a networkwide client-server binding service.
7. We saw in Section 4.18 that threads can be used in the implementation of a client or a server process for better performance of an RPC-based application. Sun RPC does not include any integrated facility for threads in the client or server, although Sun OS has a separate threads package.

4.20.2 DCE RPC

The DCE RPC is one of the most fundamental components of DCE because it is the basis for all communication in DCE. It is derived from the Network Computing System (NCS) developed by Apollo (now part of Hewlett-Packard).

The DCE RPC also uses the automatic stub generation approach. An application's interface definition is written in IDL. As an example, the interface definition of the stateless file service of Figure 4.17 is rewritten in Figure 4.21 in IDL. Notice that, unlike Sun RPC, DCE RPC IDL allows a completely general specification of procedure arguments and results. As shown in the figure, each interface is uniquely identified by a universally unique identifier (UUID) that is a 128-bit binary number represented in the IDL file as an ASCII string in hexadecimal. The uniqueness of each UUID is ensured by incorporating in it the timestamp and the location of creation. A UUID as well as a template for the interface definition is produced by using the *uuidgen* utility. Therefore, to create the interface definition file for a service, the first step is to call the *uuidgen* program. The automatically generated template file is then manually edited to define the constants and the procedure interfaces of the service.

When the IDL file is complete, it is compiled using the IDL compiler to generate the client and server stubs and a header file. The client and server programs are then manually written for an application. Finally, the same steps as that of Figure 4.20 are used to get the client and server executable files.

```
[uuid (b20a1705-3c26-12d8-8ea3-04163a0dcefz)
 version (1.0)]

interface stateless_fs
{
        const long FILE_NAME_SIZE = 16
        const long BUFFER_SIZE = 1024
        typedef char FileName[FILE_NAME_SIZE];
        typedef char Buffer[BUFFER_SIZE];

        void read (
                [in] FileName           filename;
                [in] long               position;
                [in, out] long          nbytes;
                [out] Buffer            buffer;
        );

        void write (
                [in] FileName           filename;
                [in] long               position;
                [in, out] long          nbytes;
                [in] Buffer             buffer;
        );
}
```

Fig. 4.21 Interface definition of the stateless file service of Figure 4.17 written in the IDL of DCE RPC.

The default call semantics of a remote procedure in DCE RPC is at-most-once semantics. However, for procedures that are of idempotent nature, this rather strict call semantics is not necessary. Therefore, DCE RPC provides the flexibility to application programmers to indicate as part of a procedure's IDL definition that it is idempotent. In this case, error recovery is done via a simple retransmission strategy rather than the more complex protocol used to implement at-most-once semantics.

The DCE RPC has a networkwide binding service for client-server binding that is based on its directory service (the details of the DCE directory service are given in Chapter 10). For the description here, it is sufficient to know that every cell in a DCE system has a component called Cell Directory Service (CDS), which controls the naming environment used within a cell. Moreover, on each DCE server node runs a daemon process called *rpcd* (RPC daemon) that maintains a database of (server, endpoint) entries. An *endpoint* is a process address (such as the TCP/IP port number) of a server on its machine.

When an application server initializes, it asks the operating system for an endpoint. It then registers this endpoint with its local rpcd. At the time of initialization, the server also registers its host address with the CDS of its cell.

When a client makes its first RPC involving the server, the client stub first gets the server's host address by interacting with the server(s) of the CDS, making a request to find it a host running an instance of the server. It then interacts with the rpcd (an rpcd has a well-known endpoint on every host) of the server's host to get the endpoint of the server. The RPC can take place once the server's endpoint is known. Note that this lookup is not needed on subsequent RPCs made to the same server.

The steps described above are used for client-server binding when the client and the server belong to the same cell. A client can also do an RPC with a server that belongs to another cell. In this case, the process of getting the server's host address also involves Global Directory Service (GDS), which controls the global naming environment outside (between) cells (for details see Chapter 10).

The DCE RPC also provides broadcast facility. To use this facility, a remote procedure has to be given the broadcast attribute in its defining IDL file. When a procedure with this attribute is called, the request is sent to all servers of the requested interface. All the servers receiving the request respond, but only the first response is returned to the caller; the others are discarded by the RPCRuntime library.

4.21 SUMMARY

Remote Procedure Call (RPC) is a special case of the general message-passing model of IPC that has become a widely accepted IPC mechanism in distributed computing systems. Its popularity is due to its simple call syntax, familiar procedure call semantics, ease of use, generality, efficiency, and specification of a well-defined interface. Ideal transparency of RPC means that remote procedure calls are indistinguishable from local procedure calls. However, this is usually only partially achievable.

In the implementation of an RPC mechanism, five pieces of programs are involved: the client, the client stub, the RPCRuntime, the server stub, and the server. The purpose

of the client and server stubs is to manipulate the data contained in a call or reply message so that it is suitable for transmission over the network or for use by the receiving process. On the other hand, the RPCRuntime provides network services in a transparent manner.

The two types of messages involved in the implementation of an RPC system are call messages and reply messages. Call messages are sent by the client to the server for requesting the execution of a remote procedure, and reply messages are sent by the server to the client for returning the result of remote procedure execution. The process of encoding and decoding of the data of these RPC messages is known as marshaling.

Servers of an RPC-based application may either be stateful or stateless. Moreover, depending on the time duration for which an RPC server survives, servers may be of three types—instance-per-call servers, instance-per-transaction/session servers, and persistent servers. The choice of a particular type of server depends on the needs of the application being designed.

The two choices of parameter-passing semantics in the design of an RPC mechanism are call-by-value and call-by-reference. Most RPC mechanisms use the call-by-value semantics because the client and server processes exist in different address spaces.

The call semantics of an RPC mechanism determines how often the remote procedure may be executed under fault conditions. The different types of call semantics used in RPC mechanisms are possibly or may be, last one, last of many, at least once, and exactly once. Of these, the exactly-once call semantics is the strongest and most desirable.

Based on their IPC needs, different systems use one of the following communication protocols for RPC: the request (R) protocol, the request/reply (RR) protocol, and the request/reply/acknowledge-reply (RRA) protocol. In addition to these, special communication protocols are used for handling complicated RPCs that involve long-duration calls or large gaps between calls or whose arguments and/or results are too large to fit in a single-datagram packet.

Client-server binding is necessary for a remote procedure call to take place. The general model used for binding is that servers export operations to register their willingness to provide service and clients import operations when they need some service. A client may be bound to a server at compile time, at link time, or at call time.

Some special types of RPCs operate in a manner different from the usual RPC protocol. For example, asynchronous RPC provides a one-way message facility from client to server, callback RPC facilitates a peer-to-peer paradigm instead of a client-server relationship among the participating processes, broadcast RPC provides the facility of broadcast and multicast communication instead of one-to-one communication, and batch-mode RPC allows the batching of client requests, which is a type of asynchronous mode of communication, instead of the usual synchronous mode of communication.

Unlike the conventional RPC systems, in which most of the implementation decisions are made when the RPC system is designed, the choices of transport protocol, data representation, and control protocol are delayed until bind time in an RPC system designed for a heterogeneous environment. For this, the binding facility used by such an RPC system is made considerably richer in information than the binding used by conventional RPC systems.

Bershad et al. [1990] proposed the use of Lightweight Remote Procedure Call (LRPC), which is a communication facility designed and optimized for cross-domain communications in microkernel operating systems. For achieving better performance than conventional RPC systems, LRPC uses the following four techniques: simple control transfer, simple data transfer, simple stubs, and design for concurrency.

Some optimizations that may be used to improve the performance of distributed applications that use an RPC facility are concurrent access to multiple servers, serving multiple requests simultaneously, reducing per call workload of servers, reply caching of idempotent remote procedures, proper selection of timeout values, and proper design of RPC protocol specification.

EXERCISES

4.1. What was the primary motivation behind the development of the RPC facility? How does an RPC facility make the job of distributed applications programmers simpler?

4.2. What are the main similarities and differences between the RPC model and the ordinary procedure call model?

4.3. In the conventional procedure call model, the caller and the callee procedures often use global variables to communicate with each other. Explain why such global variables are not used in the RPC model.

4.4. In RPC, the called procedure may be on the same computer as the calling procedure or it may be on a different computer. Explain why the term *remote procedure call* is used even when the called procedure is on the same computer as the calling procedure.

4.5. What are the main issues in designing a transparent RPC mechanism? Is it possible to achieve complete transparency of an RPC mechanism? If no, explain why. If yes, explain how.

4.6. Achieving complete transparency of an RPC mechanism that allows the caller and callee processes to be on different computers is nearly impossible due to the involvement of the network in message communication between the two processes. Suppose an RPC mechanism is to be designed in which the caller and callee processes are always on the same computer. Is it possible to achieve complete transparency of this RPC mechanism? Give reasons for your answer.

4.7. What is a "stub"? How are stubs generated? Explain how the use of stubs helps in making an RPC mechanism transparent.

4.8. A server is designed to perform simple integer arithmetic operations (addition, substraction, multiplication, and division). Clients interact with this server by using an RPC mechanism. Describe the contents of the call and reply messages of this RPC application, explaining the purpose of each component. In case of an error, such as division by zero or arithmetic overflow, the server must suitably inform the client about the type of error.

4.9. Write marshaling procedures for both tagged and untagged representations for marshaling the message contents of the RPC application of Exercise 4.8.

4.10. A user-defined program object is a structure consisting of the following basic data types in that order: a Boolean, an integer, a long integer, and a fixed-length character string of eight characters. Write marshaling procedures for both tagged and untagged representations for this program object. Assume that the RPC software provides marshaling of the basic data types.