a set of standard translation functions, while another method, *auto generation of stubs*, is through IDL (Interface Definition Language). It consists of a list of procedure names and supported arguments and results. The programmer writes the RPC interface using IDL. The client imports the interface, while the server program exports the interface. An IDL compiler processes the interface definitions in different languages so that the client and the server can communicate using RPC. The entire RPC compilation cycle is explained in Figure 4-6.

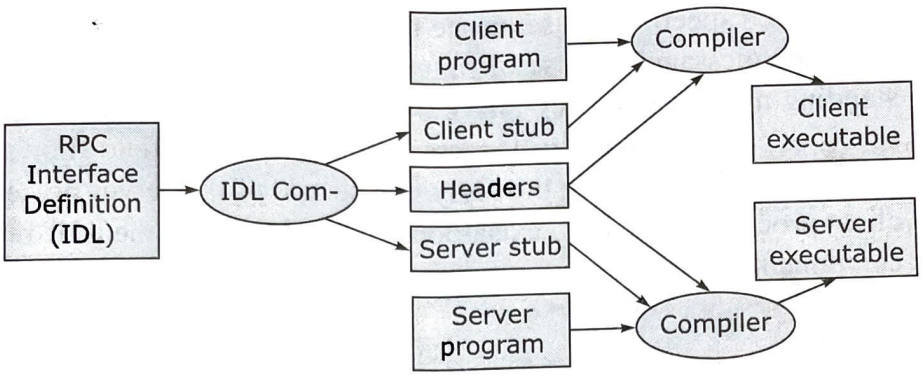

Figure 4-6 Steps for RPC compilation

In RPC, the caller process and the server process execute on different machines, and transparency is achieved using stubs. RPC consists of the following elements: client, client stub, RPC runtime, server stub, and server.

4.3 RPC Implementation

After describing the basic concepts of RPC, we now discuss the format of RPC messages and how RPCs can be implemented.

4.3.1 RPC Messages

During an RPC operation, the execution commences with the client making a request to execute the RPC. The server executes the RPC and returns the result to the client. Based on the mode of communication between the client and the server, the two types of RPC messages (Figure 4-7) are:

- *Request or call messages* from the client to the server, requesting RPC execution.
- *Reply messages* from the server to the client to return the RPC result.

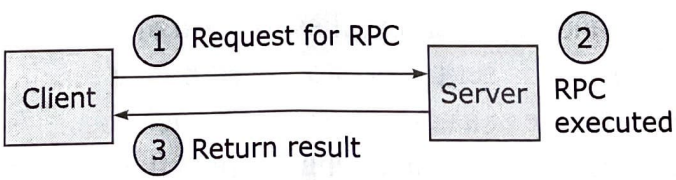

Figure 4-7 RPC messages

The RPC system protocol defines the message formats. This protocol is independent of the transport protocol, i.e. the client need not know how the message is transmitted from one process to another.

RPC call/request message

The basic function of this message is to request the server to execute an RPC by providing relevant details in the message itself. To execute an RPC, its details such as program, version, procedure number, arguments, are required. The call message has a message ID field which specifies the sequence number. It is useful for identifying lost messages or duplicate messages in case of failure; and for matching the reply message to the outstanding messages, if they arrive out of order. The message type field (either 1 or 0) specifies call (0) and reply (1) messages. The client ID field allows the server to identify the client before sending the reply. It also allows the server process to authenticate the client process before RPC execution. Figure 4-8 shows the RPC request message and its components.

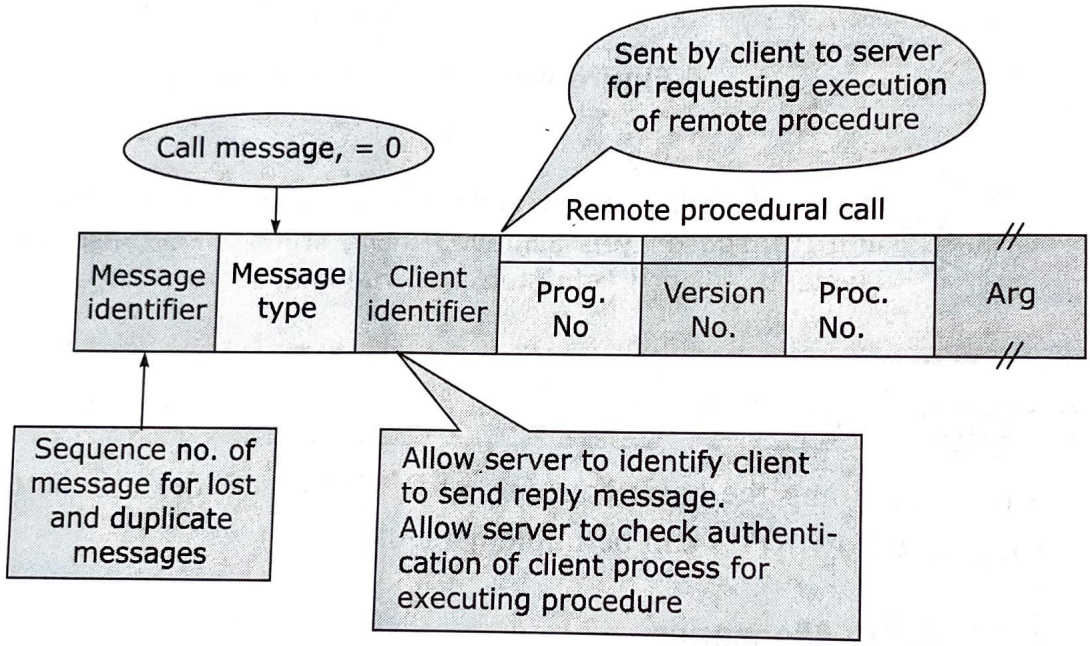

Figure 4-8 RPC call/request message format

RPC reply message

When the client requests the server to execute an RPC, the server receives the call message from the RPC. Table 4-1 shows the reply message conditions.

In conditions 1 to 5, the client receives an unsuccessful reply, specifying the reason for failure. The message ID fields of both the request and reply messages are identical to enable a proper match. As shown in Figure 4-9, the reply message is partially different for unsuccessful and successful replies. In case of a successful reply, the status field bit in the reply message is set to zero followed by the result field, while for an unsuccessful reply, it is set to one or a non- zero value, which specifies the type of error. A programmer can specify the RPC message length, since the RPC mechanism is independent of the

transport protocols. The distribution application designers are hence responsible for limiting the length of the message within the range specified by the network.

Table 4-1 RPC reply message conditions

Condition	Response from the server
▪ Server receives an unintelligible call message, probably because the call message has violated the RPC protocol.	▪ Rejects the call.
▪ Server receives the call messages with unauthorized client IDs, i.e. the client is prevented from making the RPC request.	▪ Return reply unsuccessful, and does not execute RPC.
▪ Server does not any receive procedure ID information from the message ID field—program number, version number, or ID.	▪ Return reply unsuccessful, and does not execute RPC.
▪ If all the above conditions are satisfied, the server executes the RPC, but may not be able to decode its arguments due to incompatible RPC interface.	▪ Return reply unsuccessful and does not execute RPC.
▪ Server executes the RPC, but an exception condition occurs.	▪ Return reply unsuccessful.
▪ Server executes the RPC successfully without any problems.	▪ RPC is successful and the server returns the result.

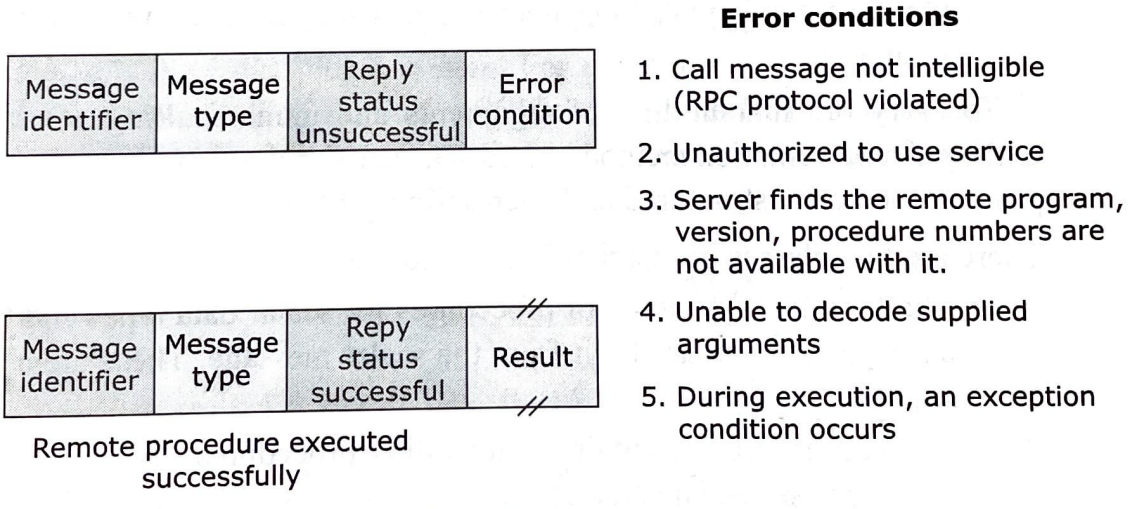

Figure 4-9 RPC reply message format

An RPC call/request message is sent by the client to the server to execute an RPC by providing its relevant details in the message itself. The RPC Reply message consists of the result or an error-code, based on the execution of the RPC by the server.

4.3.2 Parameter Passing Semantics

The function of the client sub is to take parameters from the process, pack them into a message, and send it to the server stub. The choice of a suitable parameter passing semantic is crucial to design an RPC mechanism. The various parameter passing semantics are: *call-by-value*, *call-by-reference*, and *call-by-copy/restore* semantics. We discuss each of these semantics in this section.

Call-by-value semantic

This semantic copies all the parameters into a message before transmitting them across the network. It works well for compact data types like integer, character, and arrays. The process of packing the parameters into a message is termed as *marshalling*. To make an RPC, the client sends the arguments and the server returns the result. The parameters/arguments/results are language-dependent data structures, transferred in the form of a message. As explained in the earlier chapter, the message passing process involves encoding and decoding of messages. Marshalling is a similar process carried out during RPC execution.

The marshalling process consists of the following steps:

- Identify the message data—arguments in the client or server process transmit the result to the remote process.
- Encode the data on the sender's machine by converting the program objects into a stream form and place them in a message buffer.
- On the receiving machine, decode the message, i.e. convert the message into program objects.

To carry out marshalling of arguments and unmarshalling of results, tagged or untagged representation method is used. The marshalling process must reflect all types of program objects—structured and user-defined data types.

There are two classes of marshalling procedures:

- The first group consists of procedures for scalar data types and those compound data types which are built from the scalar message. These are a part of the RPC software.
- The second group contains marshalling procedures defined by the RPC system users for user-defined data types and those which include pointers.

A good RPC system should generate its own marshalling code for every RPC. Therefore, the programmer will be relieved from doing this task. Practically, this is difficult because a vast amount of code is generated to cater to all data types. Let us take an example of a simple procedure 'add', which is remotely executed. As shown in Figure 4-10, the add procedure is executed in the client process. The client stub puts the following values in the message:

- Two parameters for execution
- Name or number of the procedure called in the message

The server stub receives the message, examines it, and makes the appropriate call. If the server supports other remote procedures, the server stub switches between multiple procedures. The actual call looks like a local call except that the parameters extracted from the message are sent by the client. After the server completes execution of the multiple RPCs, the server stub takes control. It packs the result into a message and sends it to the client stub, which unpacks it and returns the result to the client process.

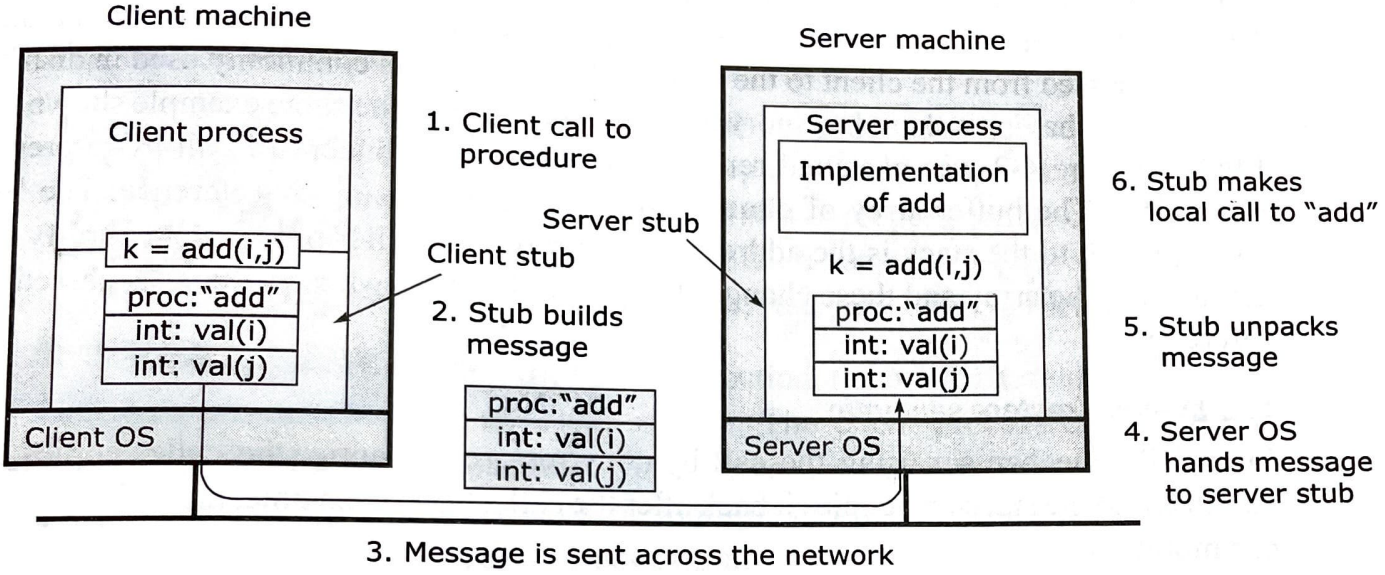

Figure 4-10 Example of call-by-value semantic

The RPC model works well if the client and the server machines are identical and the parameters and results are scalar types like integers, characters, or Boolean. This may not hold true in a large distributed system. For example, IBM mainframes use EBCDIC character code, while IBM PCs use ASCII code. Similarly, Intel Pentium machines use 'little-endian' format, while Sun SPARC machines use 'big-endian' format, as shown in Figure 4-11.

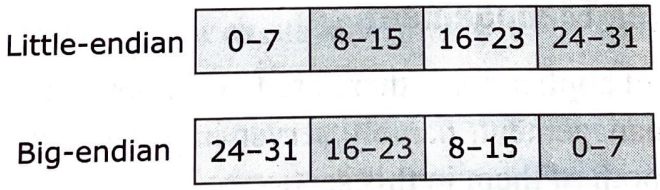

Figure 4-11 Data representations

In interprocess communication, messages are transferred byte by byte. Unequal byte-ordered machines might read the data in reverse order. Hence, in a distributed system with the above types of machines, the server may not understand the RPC parameters passed by the client. Suppose the example shown for LPC in Figure 4-2 was executed remotely. It makes the concept of call-by-value more clear. The integer value of *fd* and *nbytes* are passed by value and the caller-side values are unaffected.

Call-by-value semantic is required because the client and the server exist in different address spaces and may be running on different types of machines. It is ideal for closed

systems, where the client and the server share a single address space. However, this semantic is unsuitable for transmitting large voluminous data like multidimensional array, trees, etc., where part of the data may not be required on the server-side for RPC execution. The call-by-reference semantic is used to transmit this type of data.

Call-by-reference semantic

A pointer is meaningful only in the address space of a process where it is used. Some RPC mechanisms allow parameter passing by reference, where pointers to the parameters are passed from the client to the server. This semantic is commonly used in distributed systems having a shared memory mechanism. Suppose the same example shown for LPC in Figure 4-2 was executed remotely. It makes the concept of call-by-reference more clear. The buffer array of **char** is an example of passing-by-reference. The **buf** value pushed to the stack is the address of the array. The caller process can modify the elements of the array and these changes happen in the array whose value was passed by reference.

Call-by-copy/restore semantic

In an RPC mechanism using the call-by-copy/restore semantic, the caller copies the variable into the stack and copies it back after the call. During this time, the called process can modify it.

Which parameter passing semantic should we choose? Language designers decide the semantic based on the property of the language.

Call-by-value copies all parameters into a message before transmission. Call-by-reference passes pointers to the parameters that are passed from the client to the server, and call-by-copy/restore uses temporary storage which is accessed by both the client and the server.

4.3.3 Server Management

In RPC-based applications, there are two important issues which need to be considered for server management, namely server implementation and server creation semantics. We discuss both of them in this section.

Server implementation

We classify the servers as *stateless* and *stateful*, based on how they are implemented. A stateless server, as the name suggests, does not maintain the state information of RPC execution in the system. Hence, they include parameters for a successful operation in every request made by the client. On the other hand, in a stateful server, if the client makes multiple calls, the state information for all the calls is maintained by the server process. Subsequent calls execute with the help of earlier state information.

A stateful server is easy to restart on failure. This type of server relieves the client from maintaining the state information. In the event of a failure, if the stateful server crashes and restarts, all the earlier state information is lost. The client, being unaware of this problem, will produce inconsistent results. Similarly, if the client crashes, the server may keep on holding unwanted state information. However, stateless servers have a distinct advantage in case of failures, because the client keeps retrying until the server responds. Hence, stateless servers make crash recovery easy.

Server creation semantics

During an RPC call, the client and the server processes execute independently, i.e. they run on separate machines which have separate address spaces and lifetimes. Therefore, the middleware creates and installs the server process earlier or creates it on demand. Depending on the time duration for which a server is active, servers are classified as *instance-per-call*, *instance-per-session*, and *persistent servers*.

Instance-per-call server After RPC execution, the middleware destroys the instance per call server. They are stateless servers. The client process or the operating system maintains the state information. This approach is expensive in a distributed application, if the same server has to be invoked multiple times. It involves the overhead of resource-buffer space allocation and de-allocation.

Instance-per-session server Here, there is a server manager for each type of service. All server managers register with the binding agent. To execute an RPC, the client contacts the binding agent specifying the type of service needed. The client gets the corresponding server manager address. Then, the client contacts the server manager with a request to create a server. The client and the server interact directly for the entire session. When the client informs the server that the RPC session is complete, the server is destroyed. The server can retain information between calls, but it serves only a single client.

Persistent server This type of server exists indefinitely and is sharable among the clients in the distributed system. A persistent server is created and installed before the client uses it. Each server exports the services and registers it with the binding agent. The client contacts the binding agent with a request for service. The binding agent selects and locates the server and returns the address of the selected server to the client. Now the client interacts directly with the server. This type of server is bound to multiple clients simultaneously, servicing interleaved requests.

Servers can be classified as stateful (maintains all information and assists in easy recovery in event of a crash) and stateless (does not maintain state information). Depending on the time duration for which a server is active, servers are classified as instance-per-call, instance-per-session or persistent servers.

4.4 RPC Communication

In RPC, the caller and the called processes need to communicate with each other, since they are located on different machines. In this section, we consider three important aspects of RPC communication: RPC call semantics, various communication protocols, and client-server binding in RPC.

4.4.1 RPC Call Semantics

In RPC, the caller and the called process are located on different nodes. The caller or the called node can fail independently and may restart later. Additionally, the communication link may fail, leading to disconnection between the called and the caller node. The RPC execution is disturbed, leading to loss of the call or response message and/or the caller or called node crashes. Hence, the RPC runtime system should have an integral failure handling code.

The call semantics define how often a remote procedure is executed under fault conditions. These are as shown in Figure 4-12.

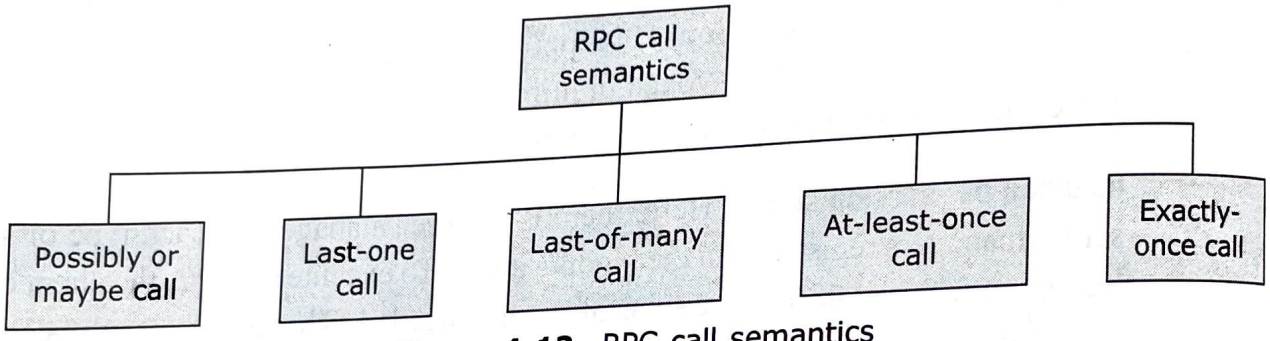

Figure 4-12 RPC call semantics

Possibly or may-be call semantics

This is one of the weakest semantics. The caller waits for a predetermined timeout period and continues with its execution. There is no guarantee of receipt of the call message or procedure execution by the caller. This semantic is ideal for applications where a response is not important for the caller, or in LANs with guaranteed successful message transmission.

Last-one call semantics

This semantic retransmits the call message, based on a predetermined timeout, until the caller receives the response. The entire set of actions like caller calling the RPC, called process executing the procedure, and the caller receiving the results, are repeated until the caller receives the result. The caller uses the results of the last executed call, even though the earlier calls may have survived the crash. This semantic is difficult to achieve in case of *orphan calls*.

Orphan calls are calls whose caller has expired due to a node crash. No parent is waiting for such calls, resulting in unwanted computation. These calls waste CPU cycles,

lock files, and may tie up resources. If the client reboots, the computation repeats, leading to rework. To achieve last-one call semantics, the various techniques used are as follows:

Extermination The client maintains a log on the disk, before it sends a call. On reboot, the log is checked and the orphan process is killed. A 'Write to the disk' operation is an overhead in terms of cost and resources. What happens if RPCs themselves call other RPCs and create grand orphans? More write operations are required. This technique is expensive because of the cost of writing every record to the disk.

Reincarnation Time is divided into sequential numbered units called *epochs*. A new epoch is started on reboot and broadcasted to all machines. The remote computations are located and killed. Some orphans may survive in partitioned networks. If the replies contain obsolete epoch numbers, we can detect the orphan calls.

Gentle reincarnation On receiving the broadcast, each machine checks for remote computation to locate the owner. If the owner is not found, then the computation is killed.

Expiration Each RPC is given a standard quantum of time to do the task. The quantum is extended on request, if the task is incomplete. When the server reboots after time T or crashes, the orphans are gone.

Last-of-many call semantics

This technique neglects orphan calls by using a call ID to uniquely identify each call. New call IDs are associated with repeated calls. The client checks the message ID and accepts the response, if the call ID matches the recently repeated call, else it ignores the message.

At-least-once call semantics

As compared to last-of-many call semantics, the call is executed one or more times, but does not specify which results the caller gets. Message timeout based on retransmission is used to implement this semantic without bothering about orphan calls. In case of nested calls having orphan calls, the caller accepts the result of the first response message, but ignores others.

Exactly-once call semantics

Irrespective of how many times the caller retransmits, the procedure executes only once. Hence, this is the strongest and most desirable call semantic. It is a cheap semantic and is not advised for *idempotent operations*. These are operations where if a procedure is executed more than once with the same parameters, the same results and side effects are produced. Hence, the idempotent interfaces need to be designed by the application programmer.

RPC call semantics define how often the execution of the remote procedure takes place under fault conditions. They can be classified as: possibly or maybe call, Last-one call, last-of-many call, and exactly-once call semantics.

4.4.2 RPC Communication Protocols

On the basis of RPCs, different systems have different IPC requirements. Several communication protocols are explained here which cater to the needs of different systems. These are: request protocol, request/reply protocol, and request/reply/acknowledge-reply protocol.

The request protocol (R protocol)

RPC uses this protocol in applications where the client procedure has nothing to return as the result of procedure execution, as shown in Figure 4-13. Client requires no confirmation that the procedure is complete. It helps in improving the client and server performance, because the client is not blocked waiting for result and the server does not need to send a reply message.

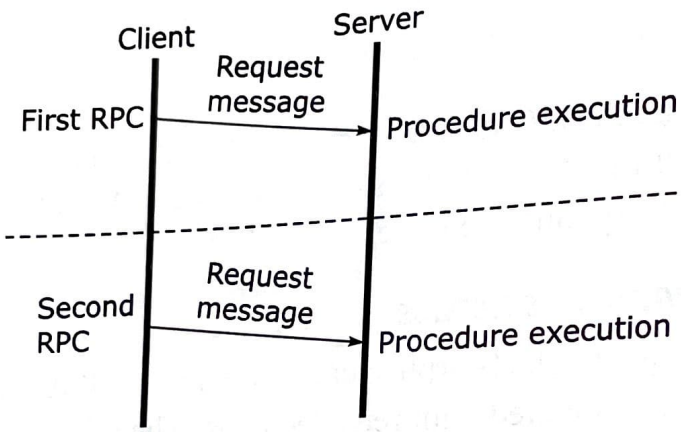

Figure 4-13 The request protocol

An RPC using this protocol is termed *asynchronous RPC*, as shown in Figure 4-14. The RPC runtime is not responsible for retrying the request, in case of communication failure. These protocols are useful for implementing periodic update services like synchronizing time.

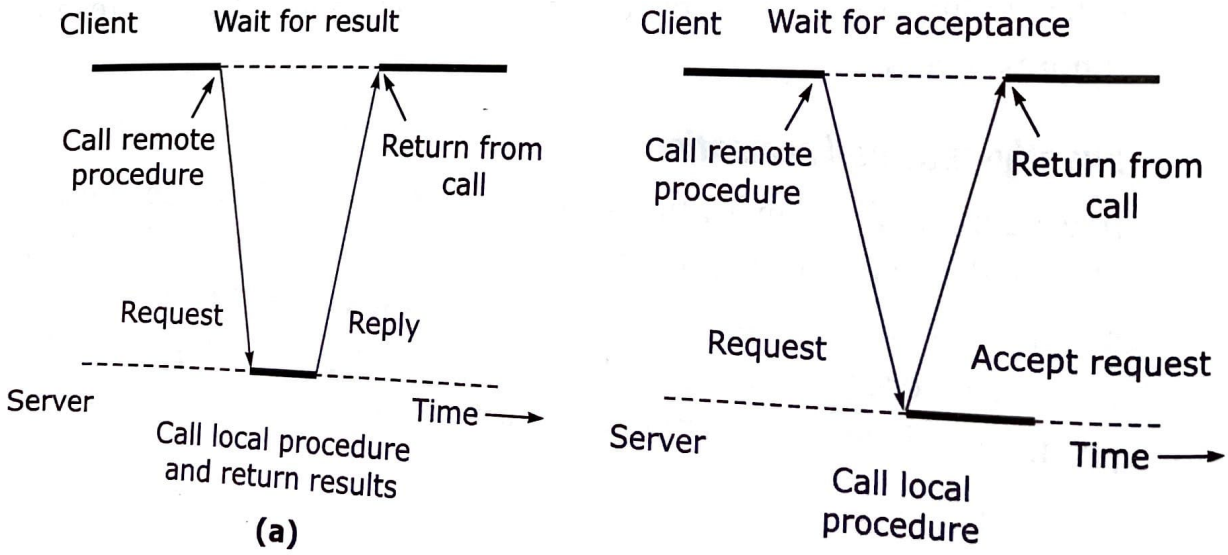

Figure 4-14 Asynchronous RPC

The request/reply protocol (RR protocol)

This protocol is used for RPC whose arguments and results fit into a single packet buffer and where the call duration and the time between the call is short. This protocol avoids explicit transmission of messages. The server's reply message acts as acknowledgement of the client's request message. The subsequent call from the client is considered an acknowledgement of the server's reply message of the previous call made by the client. Figure 4-15 shows the message exchange between the client and the server.

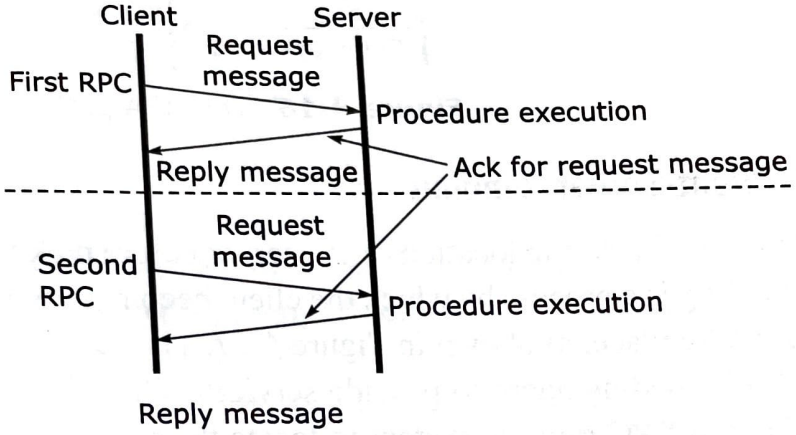

Figure 4-15 The RR protocol

The protocol uses timeouts and retries for handling failures, but it requires that the cache maintain the replies. If the servers interact with a large number of clients, the cache discards the data after a specific time. This method is not reliable, because the messages that are not successfully delivered to the client are lost. The RRA protocol described next overcomes this issue.

The request/reply/acknowledgement-reply protocol (RRA protocol)

This protocol requires that the client should acknowledge the receipt of the reply messages. Subsequently, the server deletes those messages from the cache. The RRA protocol involves three messages per call, as seen in Figure 4-16. However, what happens if the acknowledgement message gets lost? One possible solution is to assign unique IDs to request messages. Reply messages are matched with the corresponding acknowledge-ment message. The client acknowledges the reply message only if it receives the replies to all earlier requests. The loss of an acknowledgement message is immaterial, since an acknowledgement corresponds to the receipt of all reply messages (with lower IDs).

RPC communication protocols can be classified as R-protocol, RR protocol, or RRA protocol, depending on the number of messages involved in the communication between the client and the server for RPC execution.

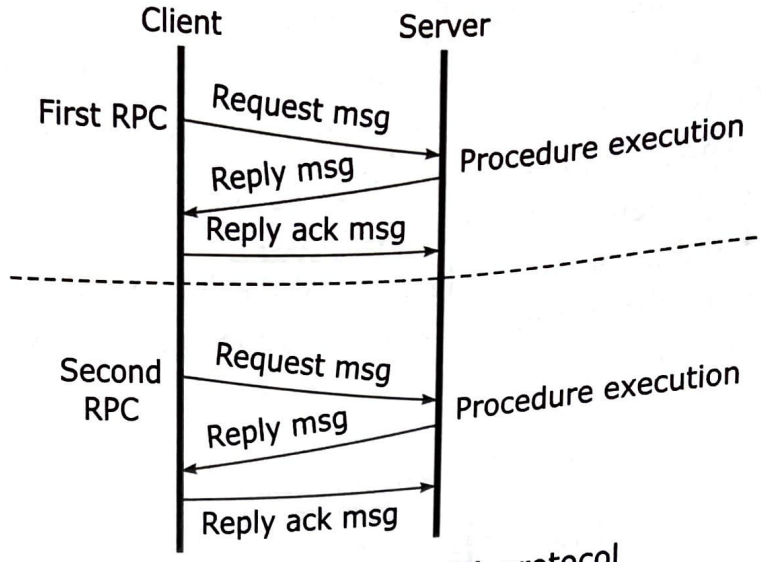

Figure 4-16 The RRA protocol

4.4.3 Client-Server Binding

Clients should know the location of the server before making a request for RPC. Client-server binding is a process by which the client becomes associated with the server so that a call can take place, as shown in Figure 4-17. The server exports operations and registers with the binding agent to provide services. The client imports these operations by requesting the RPC runtime system to locate the server for RPC execution. The client-server binding process deals with issues like server naming, server locating, binding time, and multiple simultaneous bindings. We explain each of these issues in this section.

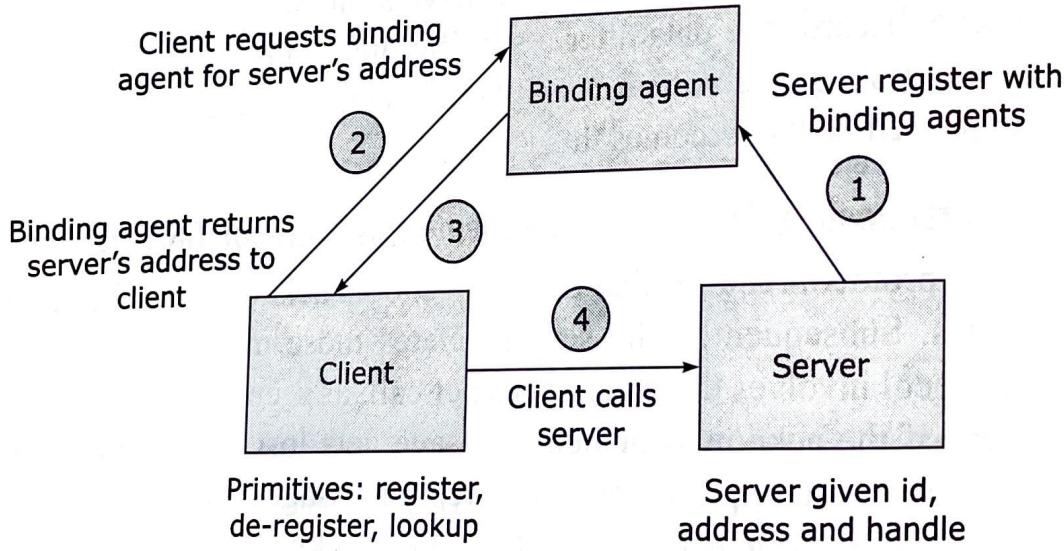

Figure 4-17 Client-server binding

Server naming

The client names a server with which it wants to communicate, by specifying its interface name. The *type* part of the interface name specifies the interface itself and its version number. The latter is used to distinguish between old and new versions of the interface. The instance specifies the server providing the service within the interface and can have multiple instances. Interface names are created by the user and not by the RPC package.