4.1 Introduction to Remote Communication

Distributed applications are difficult to build with IPC protocols tailored only for specific applications. Distributed systems use explicit message passing (send and receive primitives) for communication. The programmer handles these primitives and hence, this model fails to achieve *access transparency*, which is an essential feature of any distributed system. Hence, it was felt that there was a need for a generalized protocol which would serve the purpose. This chapter is concerned with the remote procedural call (RPC) and remote object invocation (RMI) programming models for distributed applications.

RPC is an extension of a local procedural call, which allows client programs to call server programs running on separate processes on remote machines. RPC packages the message more like a conventional program and is easy to use. The first half of the chapter focuses on the concepts, implementation, strengths, and weaknesses of RPC. The later part of the chapter discusses RMI. This technique allows objects in different processes running on different machines to communicate with one another. RMI is an extension of local method invocation, which allows objects in one process to invoke methods of an object in another process running locally or remotely. A later section of the chapter discusses the concepts, features, and operation of RMI followed by an example Java RMI.

4.1.1 Middleware

Middleware is a software which provides a programming model one level above the basic building block of processes and also provides message passing. This layer uses protocols like the request-reply protocol between processes to provide higher-level abstraction. Middleware provides location transparency and independence from the details of communication protocols, operating system, and hardware. RPC and RMI are the examples of middleware.

The various layers of a middleware are:

1. *Location transparency* The client need not know which server process is running its RPC – whether it is local or remote or where it is located. Similarly in RMI, the object making the invocation need not be aware of the object's location or whether the object invoked is local or remote.
2. *Communication protocols* The processes supporting middleware abstraction are independent of the lower-level transport protocols.
3. *Computer hardware* RPC uses techniques like external data representation to hide the heterogeneous nature of hardware architecture like *byte-ordering* from the user.
4. *Operating system* The higher layer abstraction provided by the middleware is independent of the underlying operating system.

5. *Use of programming languages* Middleware is designed using programming languages like Java, CORBA, or IDL (Interface Definition Language).

Figure 4-1 shows the role of middleware in remote communication.

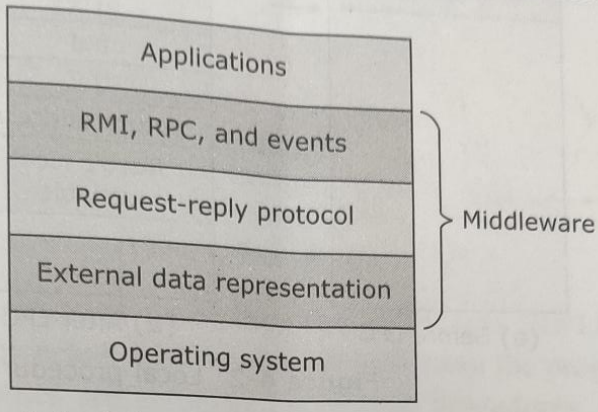

Figure 4-1 Role of middleware in remote communication

Middleware is the software that provides a programming model one level above the basic building block of processes and message passing.

4.2 Remote Procedural Call Basics

The RPC model is similar to the commonly used local procedure call (LPC) model that is used to transfer data and control within a program. We first describe the steps involved in LPC execution and then compare it with RPC. In the LPC model, the caller places arguments to a procedure in a specified location–memory or register. Then, the caller transfers the control to the procedure. The procedure executes in a new environment which consists of the arguments given in the calling instruction. After completion, the caller regains control, extracts the results of the procedure, and continues its execution. Figure 4-2 shows LPC execution.

Consider a simple LPC:

Count = read (fd, buf, nbytes)

where *fd* is an integer indicating a file, *buf* is an array of characters where data is read, and *nbytes* is the number of bytes to be read.

Figure 4-2 indicates the stack before the call is made from the main program. Parameters are pushed onto the stack in the reverse order. After the read execution is complete, it puts the return value in the register, removes the return address from the stack, and transfers the control back to the caller. It now takes the parameters from the stack. RPC works in a similar manner.

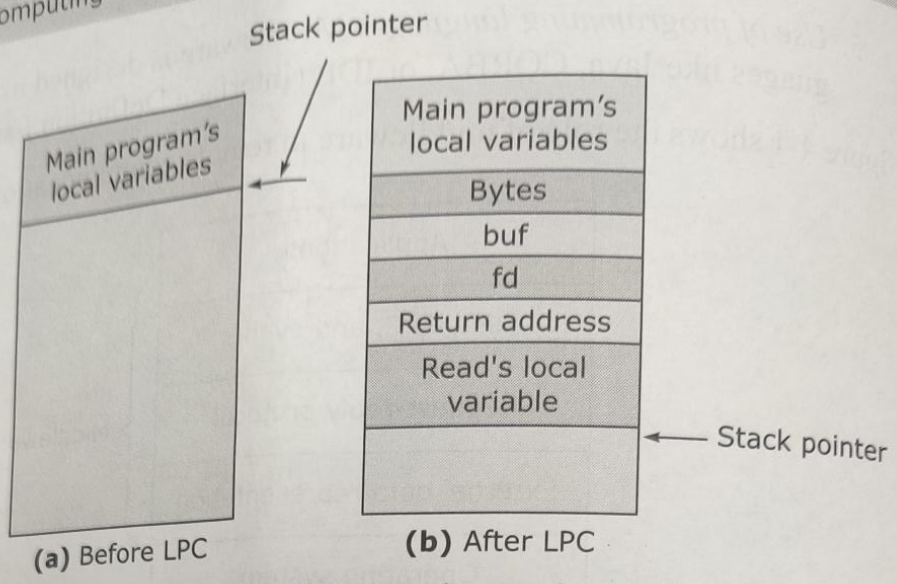

Figure 4-2 Local procedure call

4.2.1 Basic RPC Operation

In RPC, the caller process and the server process execute on different machines. First, the caller process sends a call message with procedure parameters to the server process. Then, the caller process suspends itself to wait for a reply message. A process on the server side extracts the procedure parameters, computes the results, and sends a reply message back to the caller process.

As seen in Figure 4-3, an RPC process on the client machine calls a procedure P(X, Y, Z) on the server at which point the calling process is suspended and the server initiates the execution of the called process. When the called process receives the parameters, X, Y, Z, it computes the result 'P' and sends a reply message to the calling process.

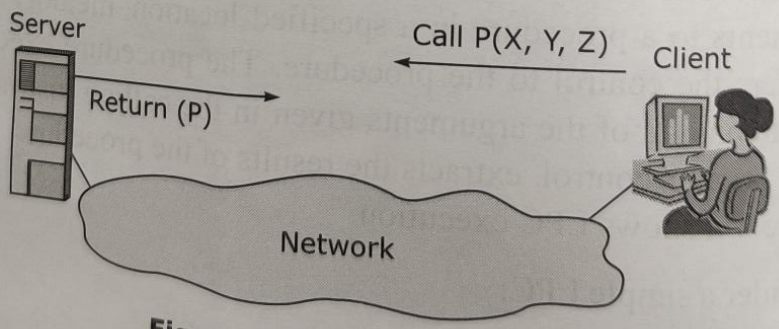

Figure 4-3 Basic RPC model

Figure 4-4 shows how only one process is active at a time. The client process is in the blocked state and may do other useful work while waiting for reply from the server. In RPC, the calling and the called procedures run on different machines and execute in different address spaces. Message passing carries out information exchange between the two processes and is transparent to the programmer. RPC is the popularly used communication mechanism for distributed applications because of its simple syntax and semantics.

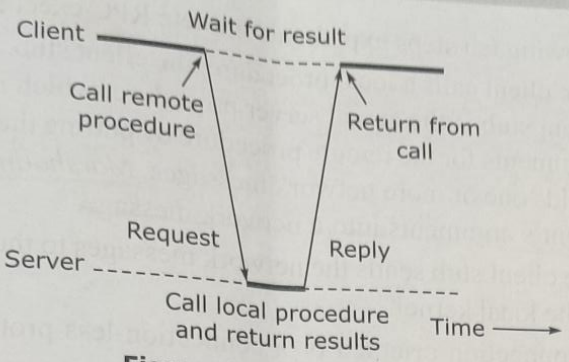

Figure 4-4 A typical RPC

To achieve transparency, designers have made RPC look like LPC, using the concept of *stubs* that hides the actual RPC implementation from the programs. Both the client and the server processes are associated with stub procedures. RPC implementation mechanism involves the following elements:

(a) Client
(b) Client stub
(c) RPC runtime
(d) Server stub
(e) Server

Figure 4-5 depicts the relation between the elements and the steps of a basic RPC operation. An instance of client, client stub, and RPC runtime executes on the client machine, while a similar separate instance executes on the server machine. The caller process is the client and the called process is the server.

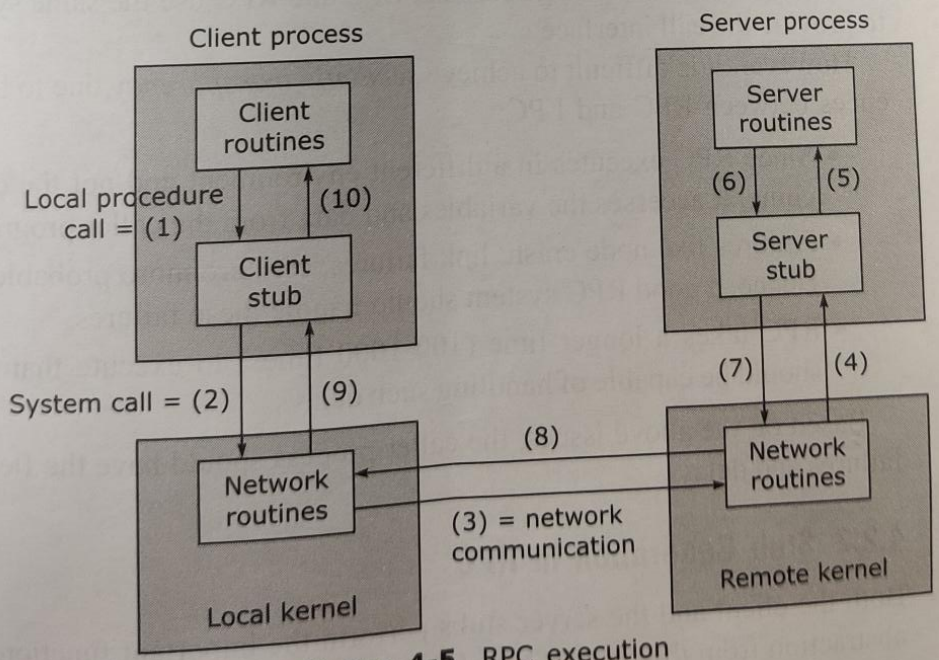

Figure 4-5 RPC execution

The following ten steps explain a complete RPC execution process:

1. The client calls a local procedure, the client stub. For the client, it appears as if the client stub is the actual server procedure, which it has called. A stub packages the arguments for the remote procedure by putting them into some standard format and builds one or more network messages. *Marshaling* is the process of packaging the client's arguments into a network message.
2. The client stub sends the network messages to the remote system via a system call to the local kernel.
3. A connection-oriented or a connection-less protocol transfers the network messages to the remote system.
4. The server stub procedure waits on the remote system for the client's request. It unmarshals the arguments from the network message and converts them.
5. The server stub executes a local procedure call to invoke the actual server function, passing it the arguments that it received from the client stub.
6. When the server procedure is completed, it sends the return values to the server stub.
7. The server stub converts the return values, marshals them into one or more network messages, and sends them back to the client stub.
8. The messages are transmitted across the network back to the client stub.
9. The client stub reads the network messages from the local kernel.
10. After converting the return values, the client stub returns to the client function.

For the client, it appears to be a normal procedure call with the results returned to the client. As far as the client is concerned, the remote server is accessed by executing an LPC and not by using the send-and-receive primitives. The client and the server stub hide the actual details of message passing. LPC and RPC use the same syntax and semantics to provide the call interface.

However, it is difficult to achieve *semantic transparency* due to the following differences between RPC and LPC:

- Since RPC executes in a different environment and not the caller process's machine, it accesses the variables and data from the caller program's address space.
- Failures like node crash, link failures, etc., are more probable in RPC than LPC. Hence, a good RPC system should handle these failures.
- RPC takes a longer time (100–1000 times) to execute than LPC. Hence, RPC should be capable of handling such delays.

Based on the above issues, the caller process should have the flexibility of handling failures and delays.

4.2.2 Stub Generation in RPC

Both the client and the server stubs perform the important function of hiding the RPC abstraction from the client. One of the methods, *manual generation of stubs*, is through

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.

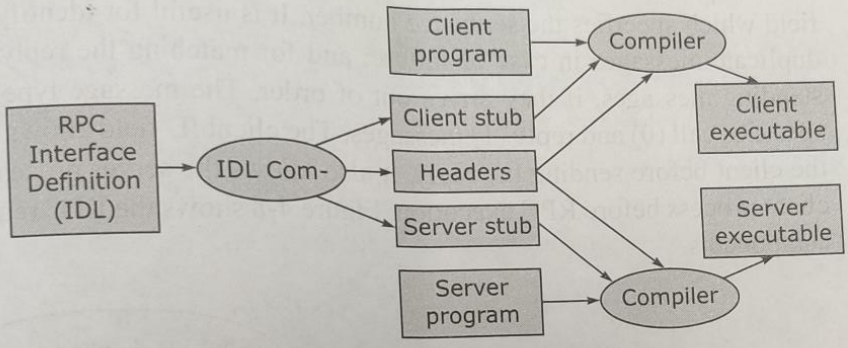

Figure 4-6 Steps for RPC compilation

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.

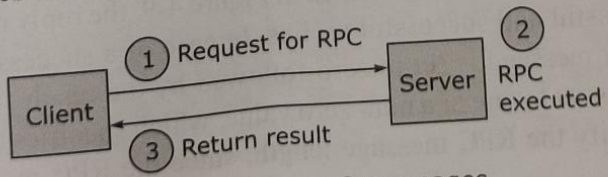

Figure 4-7 RPC messages

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.

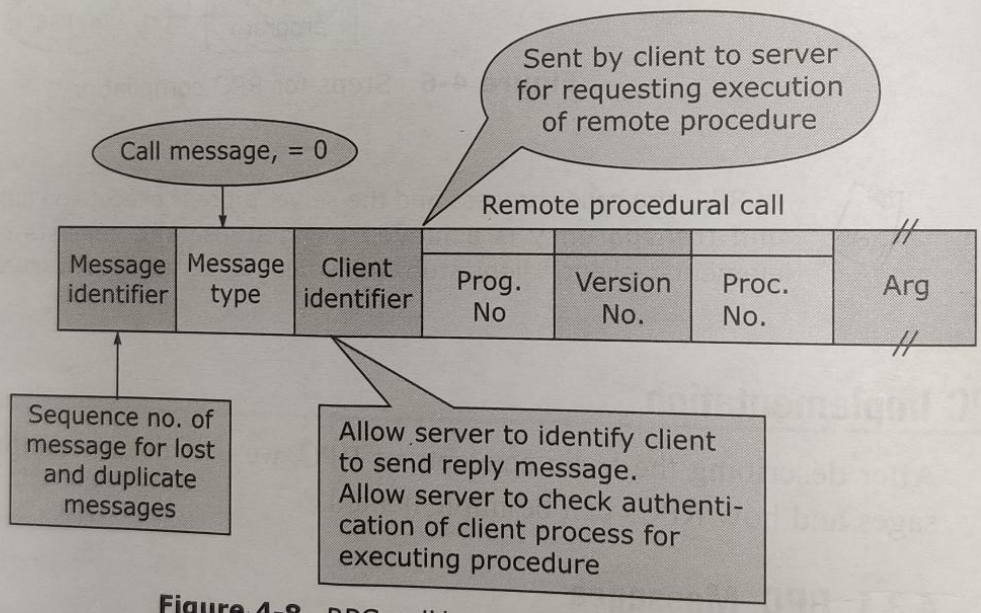

Figure 4-8 RPC call/request message format

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.

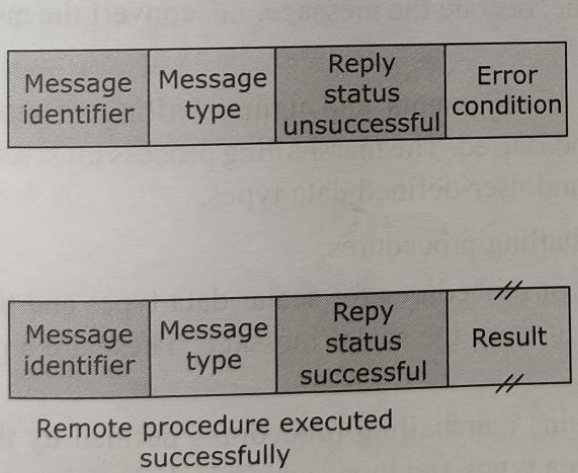

Figure 4-9 RPC reply message format

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.

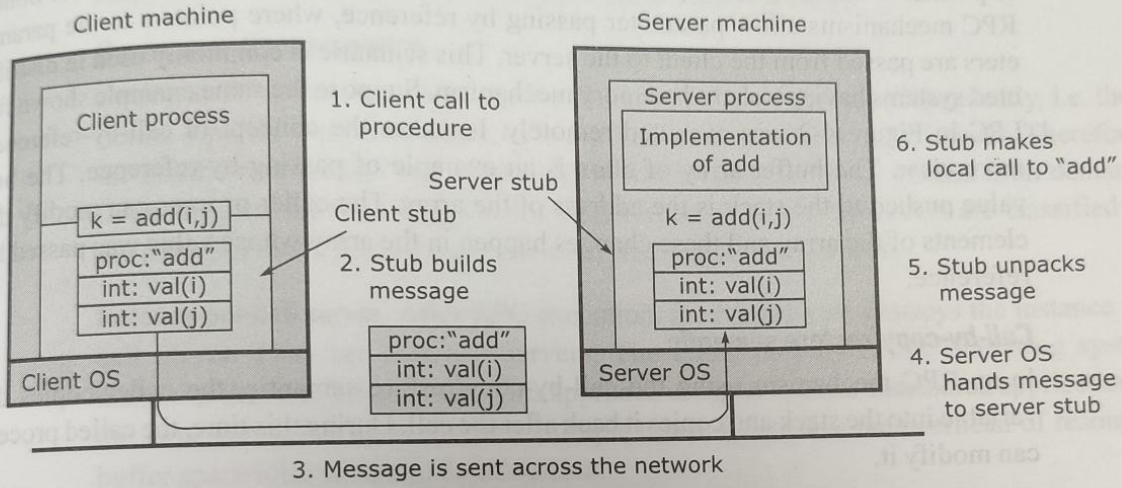

Figure 4-10 Example of call-by-value semantic

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.

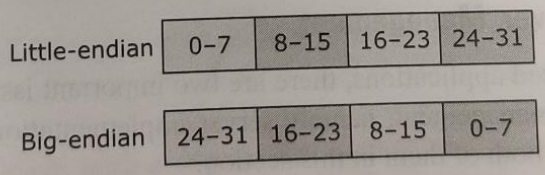

Figure 4-11 Data representations

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.

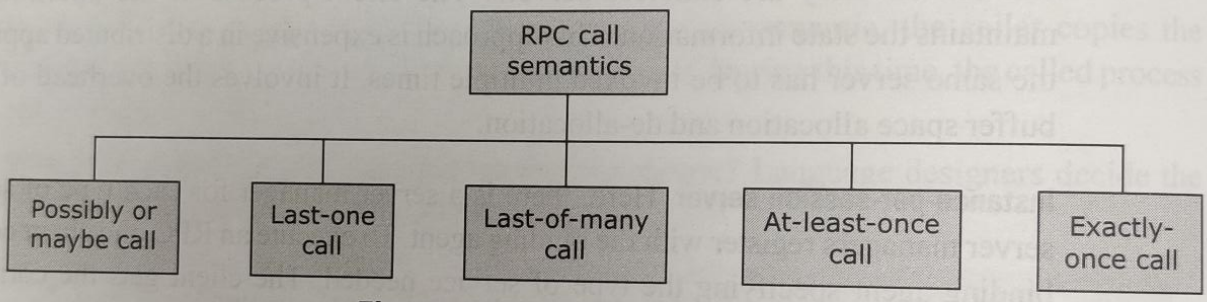

Figure 4-12 RPC call semantics

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.

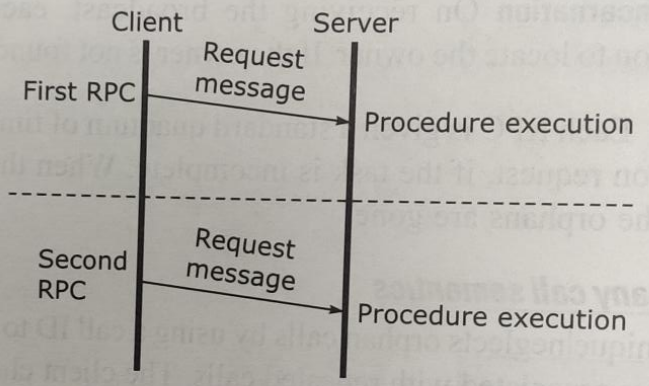

Figure 4-13 The request protocol

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.

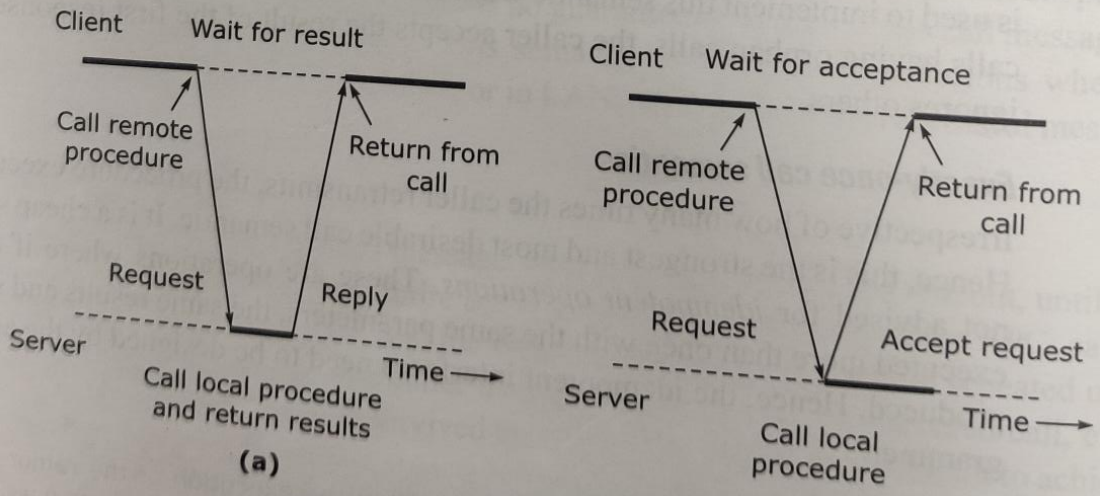

Figure 4-14 Asynchronous RPC

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.

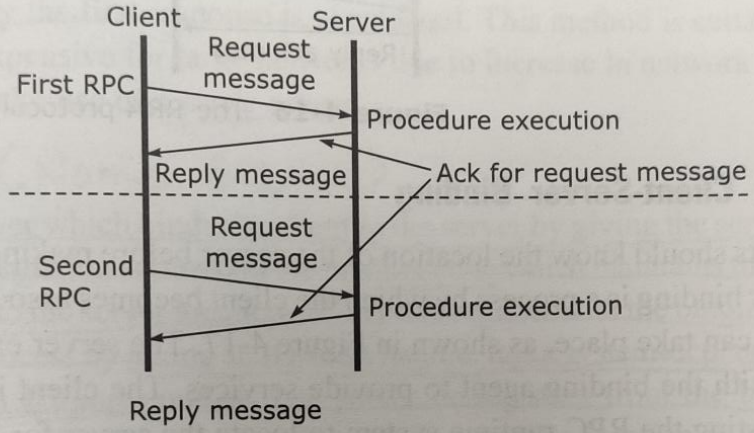

Figure 4-15 The RR protocol

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.

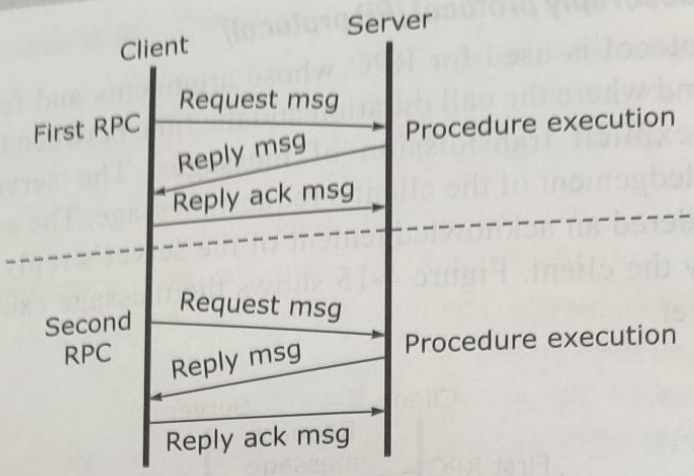

Figure 4-16 The RRA protocol

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.

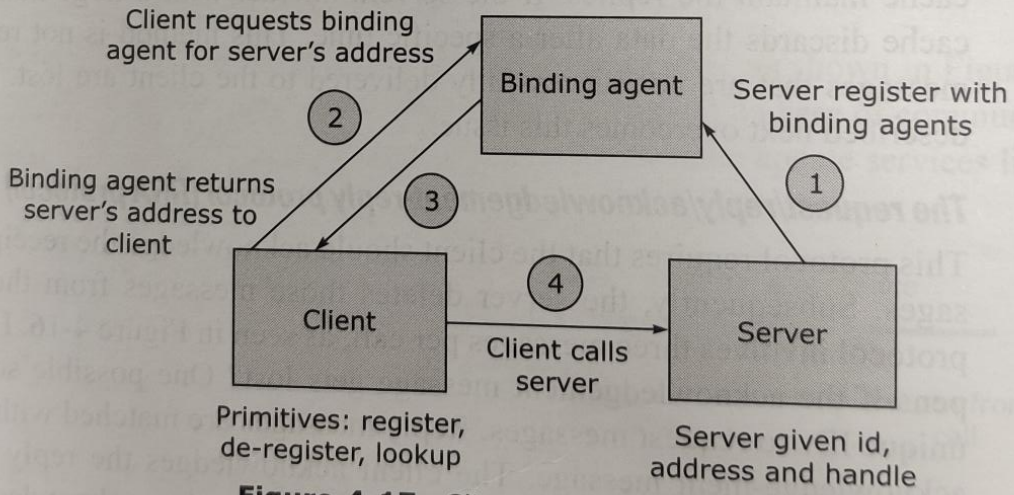

Figure 4-17 Client-server binding

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.

The main function of the RPC package is to locate the exporter. Binding is a process of establishing a connection between the client and the server.

Server locating mechanism

The interface name of the server is a unique ID. The server should be located prior to an RPC call, i.e. before the client makes a request for RPC. A common method for server location is broadcasting. In the broadcast method, the client sends a message to all nodes to locate the server. The node having the specified server sends a response message. If the server is replicated on many nodes, the response message is received from many servers, but only the first response is considered. This method is suitable for small networks, but is expensive for large networks due to increase in network traffic. The other method is to make use of a *name server*.

Binding agent (Name Server)

It is a name server which binds the client to the server by giving the server location to the client. This information is stored in the binding table which maintains the system-interface name mapping to the server location. All servers register with the binding agent during the initialization process by giving an ID and a handle. It is a system-dependent Ethernet or IP address/process ID/port number. The server de-registers from the binding agent, if it does not want to offer its services. The binding agent's location is broadcast to all the clients when it is relocated.

The primitives used to communicate with the binding agent are:

- 'Register' used by the server to register with the binding agent
- 'Deregister' used by the server to deregister with the binding agent
- 'Lookup' used by the client to locate the server

One of the advantages of the binding agent method is multiple server support, which provides *fault tolerance*. Any server can offer its services. The other advantages of this method are *load balancing* and *security*. The server holds the privilege of serving authorized users, and refusing service to other users. One of the drawbacks is the large overhead involved in binding a client to the server for short-lived processes. For large networks, the binding functions are distributed among multiple agents by information replication. However, this involves extra overhead to maintain consistency.

Types of bindings

Clients can be bound to a server in a variety of ways, such as *fixed binding* and *dynamic binding*. In fixed binding, the client knows the network address of the server. The client binds directly with the server and carries out RPC execution. This method fails if the server is moved or replicated. The other method is dynamic binding, which can be carried out in any of the following three ways: at compile time, link time, or call time.

Binding at compile time The client and the server modules are programmed as if they are to be linked together. As compared to fixed binding, the programmer adds the server

network address into the client code at compile time. The client then finds it by looking up the server name in a file. If the server moves or is replicated, it cannot be located. Programs have to be found and recompiled if the server location is changed or the server is replicated. This method is ideal for clients and servers having static configuration.

Binding at link time The server exports its service by registering itself with the binding agent during the initialization process. Before making a call, the client makes an import request to the binding agent. It binds the client and the server and returns the server handle to the client, which makes the call to the server. For the next set of calls to the same server, the server's handle is cached to avoid contacting the binding agent. This method works well for applications where the client needs to call the server several times once it is bound.

Binding at call time Client is bound to the server when it calls the server for the first time during execution. The client passes the server's interface name and RPC arguments to the binding agent. The binding agent locates the target server and sends an RPC call message with arguments to it. The server completes the RPC execution and sends the result back to the binding agent. It then returns the results and the target server's handle to the client. During subsequent calls, the client calls the server directly.

Dynamic binding is useful from the reliability point of view. Binding is the process of establishing a connection between the client and the server. Sometimes, the client and the server may want to change the connection, in case a server migrates or replicates. Whenever the binding is changed, the state data held by the server is destroyed if not needed later; else it is duplicated in the migrated node. Generally, a client is bound to a single server. However, there are some cases where the client can be bound to multiple servers of the same type. This call results in multicast communication because the call message is sent to all the servers, which are bound to the client. For example, when a file needs to be replicated at several nodes, multiple servers are involved.

RPC binding is a process by which the client becomes associated with the server so that an RPC call can take place. A few issues involved in this process are server naming and server locating. Binding can be classified as fixed binding and static binding.

4.5 Other RPC Issues

In this section, we focus on other issues in RPC implementation not covered so far, such as exception handing and security, failure handling, optimizing RPC execution, and various types of complicated RPCs.

4.5.1 Exception Handling and Security

In case an RPC does not execute successfully, the server reports an error in the reply message. Hence, RPC should have an effective exception handling mechanism to report

failures to clients. The various types of failures like server or client node failure, communication link failure, etc. are discussed in the earlier section. A systematic method of exception handling is to define the exception conditions for each type of error. An error raises the corresponding flag and the particular procedure is automatically executed on the client side. Programming languages like ADA and CLU use this method, which supports exception handling. What if the language does not support exception handling? In such a case, the local operating system needs to take care of these exceptions.

Some RPCs include client-server authentication and encryption techniques for calls. Full end-to-end encryption of calls and results uses the federal Data Encryption Standard (DES). This encryption technique is used to prevent tapping of data, detect replay, and execution of calls. Else, the user has the flexibility to implement his own security (authentication and data encryption) mechanism as desired. While designing an application, the user needs to consider the following issues related to security of message communication:

- Is the authentication of the server by the client required?
- Is authentication of the client by the server required when the result is returned as a reply message?
- Should users other than the caller and the called be allowed access to the RPC arguments and results?

These and other aspects of security are discussed in detail in Chapter 10.

4.5.2 RPC in Heterogeneous Environment

The design of distributed applications using RPC should take care of the heterogeneous nature of the system. Typically, more portable the application, the better it is. While designing an RPC system for a heterogeneous system, we consider three types of heterogeneity: data representation, transport protocol, and control protocol.

Data representation

Machines with different architectures may use different representations, as discussed in Section 4.3.2. For example, Intel uses the little-endian format which numbers the bytes from right to left, while Sparc uses the big-endian format. Some machines may use 2's complement notation for storing numbers. Also, floating point representations may vary from machine to machine. Hence, an RPC system designed for a heterogeneous environment should take care of the differences in data representations of the client and server machines.

Transport protocol

RPC systems must be independent of the underlying network transport protocol for better portability of applications. This will allow distributed applications using RPC systems to run on different networks which use different protocols.

Control protocol

To enable applications to be portable, the RPC system must be independent of the underlying network control protocol which defines the control information in each transport packet used to track the state of the call.

One of the simplest approaches to deal with the issue of heterogeneity is to delay the choices of data representations, transport protocol, and the control protocol until bind time. In conventional RPC systems, these decisions are made at the design stage. Hence the binding mechanism of RPC system for a heterogeneous environment is richer, since it includes a mechanism for determining data conversion software, the transport protocol, and the control protocol to be used between a specific client and server. This mechanism will return the correct procedures to the stubs as result parameters of the binding call. These binding mechanisms are transparent to the users. The application programs never directly access the component structures of the binding mechanism; they deal with bindings as atomic types and acquire and discard them through RPC system calls. A few RPC systems designed to support heterogeneous environments are HCS (Heterogeneous Computer Systems), HRPC, and Firefly RPC.

4.5.3 Failure Handling

The main goal of an RPC system is to project the RPC execution as LPC. RPC works well as long as there is no failure, but that is an ideal situation. The various failures which can occur during RPC execution are discussed below.

Client cannot find the server It is possible that the client cannot locate a suitable server to provide services or make a specific call. In addition, if a new version of the interface is installed, the stubs are generated again, but the server may still include old binaries. When the client makes a request to the binding agent, it receives a failure code.

Request from client to the server is lost The caller kernel starts a timer while sending the request. The kernel resends the message if the timer expires, and it receives no reply or acknowledgement. If this retransmission is done repeatedly, the client understands that the server is down, leading to the first type of failure: Cannot Locate Server.

Reply from server to the client is lost The sender timer times out and the message is retransmitted even if the reply is received. However, the server may end up executing the request again, if it had already received the first request and executed it. This could result in problems in case of non-idempotent operations, which produce side effects if executed repeatedly. One of the solutions is to structure the request as an idempotent operation. The other solution is to assign a sequence number to the message while sending it. The server identifies the sequence number and does not execute the message again.

Server crashes after getting the request The server could crash before or after executing a request, but before sending the reply. In the first case, the server informs the failure to the client while in the latter case, it retransmits the request. To overcome the failures, various server call semantics are used which are described in the earlier section.

Client crashes after sending the request What happens if the client crashes before it can receive a reply from the server? These calls are called orphan calls and we have already described them in the earlier section.

4.5.4 RPC Optimization

Similar to any system design, performance plays an important role in the design of a distributed system. There are various optimizations which are possible and can be adopted to achieve better performance of distributed applications using RPC. These are: providing concurrent access to multiple servers, serving multiple requests simultaneously, reducing per-call workload of server, using reply cache for idempotent RPCs, selection of timeout value, and the design of RPC protocol specification, as shown in Figure 4-18. Performance of any RPC execution can be significantly improved by choosing an appropriate optimization technique.

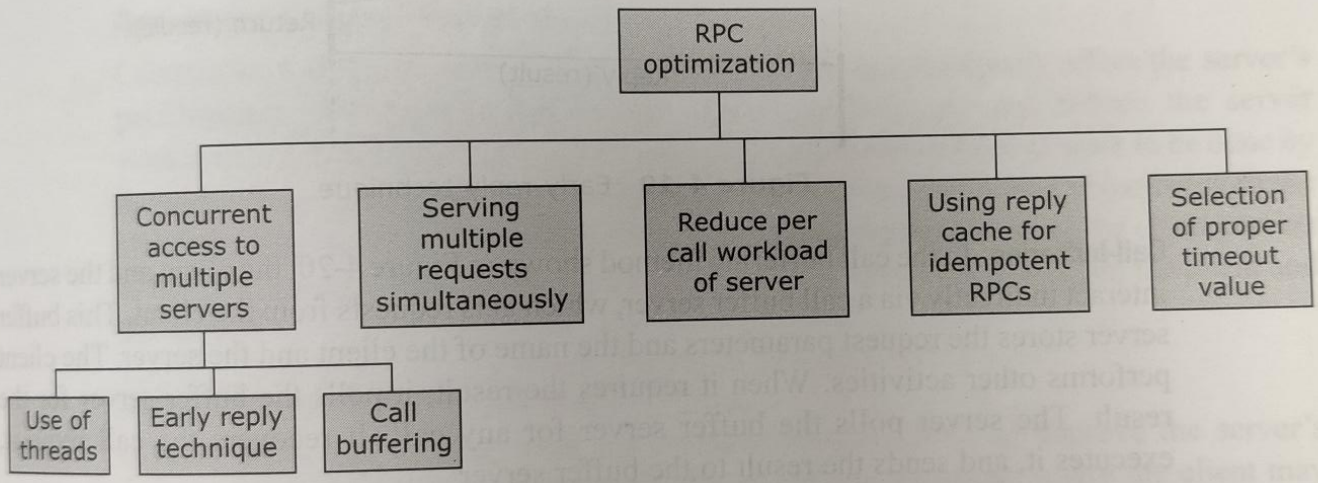

Figure 4-18 Techniques for RPC optimization

Concurrent access to multiple servers

One benefit of RPC is its synchronization property, and many distributed systems benefit from concurrent access to multiple servers. Any one of the following approaches can be used to provide this facility: use of threads, early reply technique, and call buffering approach.

Use of threads Threads can be used to implement a client process (discussed in detail in Chapter 6). Each thread can make an independent call to different servers. The underlying protocol should be capable of providing correct routing of responses.

Early reply technique To improve performance and throughput, some applications use concurrent access to multiple servers. One method uses threads to implement RPCs to different servers. The other method is the early reply technique where the call is split into two RPC calls. RPC 1 passes parameters to the server, which returns a reply-tag. The tag is sent to the server with the second call to match the call with the correct result. The client

decides the time between the calls and does other activity during this period. The disadvantage of this method is that the server holds the result of the call until the client makes a request for it. This technique is depicted in Figure 4-19.

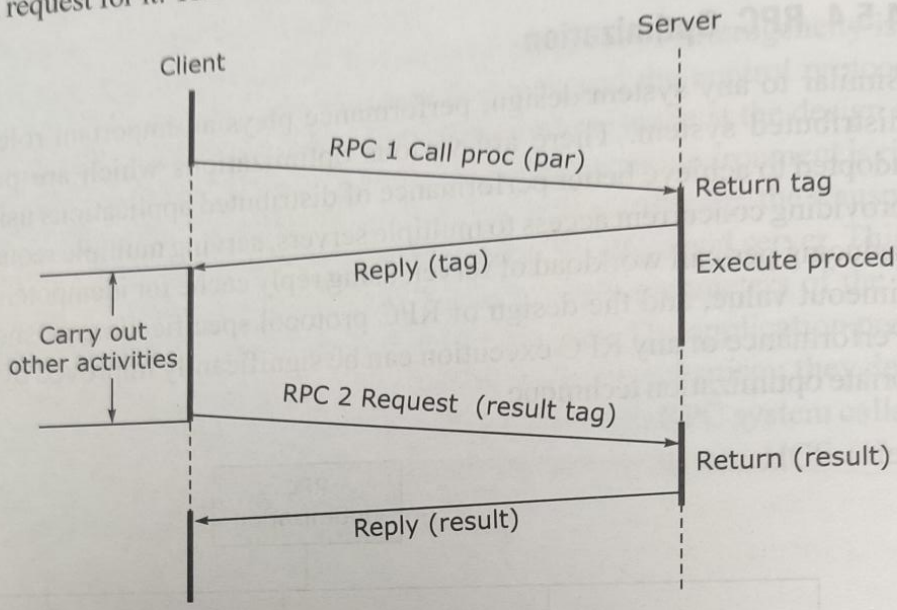

Figure 4-19 Early reply technique

Call-buffering In the call buffering method shown in Figure 4-20, the client and the server interact indirectly via a call buffer server, which gets requests from the client. This buffer server stores the request parameters and the name of the client and the server. The client performs other activities. When it requires the result, it polls the buffer server for the result. The server polls the buffer server for any call. It recovers the call request, executes it, and sends the result to the buffer server.

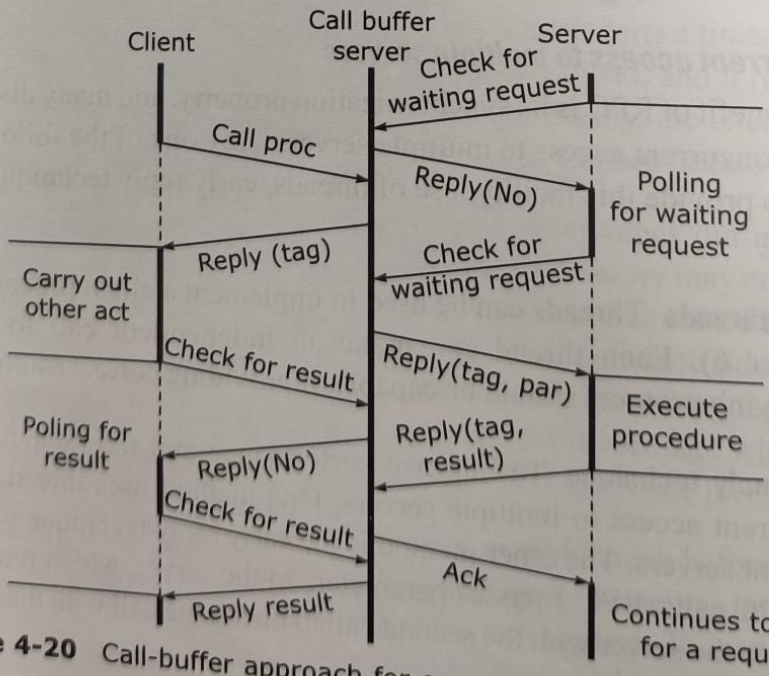

Figure 4-20 Call-buffer approach for concurrent access to multiple servers

Serving multiple requests simultaneously

There are two types of delays which can occur in any RPC system:

- Delay caused when a server waits for a resource which is temporarily unavailable. For example, during RPC execution, a server may wait for a file which is locked by another user.
- Delay caused when a server calls a remote function which involves a large amount of computation to complete execution or involves a large transmission delay.

A good RPC system must have schemes to avoid being idle when waiting for completion of some operations. Servers should be able to service multiple requests simultaneously. RPC systems need to handle the delays caused by servers waiting for a resource. The server should be designed in such a way that it can service multiple requests simultaneously. One way to achieve this is to use a multi-threaded server approach with a dynamic thread creation facility for server implementation.

Reduce per-call workload of server

Catering to a large number of requests from clients can drastically affect the server's performance. To improve the overall server performance and reduce the server workload, it is beneficial to keep the requests short and the amount of work to be done by the server low for each request. Stateless servers help to achieve this objective with the client tracking the requests made to the server. This is reasonable, since the client portion of the application is actually in charge of the flow of information between the client and the server.

Using reply cache for idempotent RPCs

A reply cache can also be associated with idempotent RPC to improve the server's performance when it is heavily loaded. There may be situations where the client may send in requests faster than the server can process. This results in a backlog and the client requests a timeout and the client resends the requests, making the situation worse. The reply cache is helpful in such a situation because the server processes the request only once. In case the client resends the request, the server sends the cached reply.

Selection of the timeout value

Timeout-based retransmissions are necessary to deal with failures in distributed applications. The choice of the timer value is an important issue. A too-small timer will expire soon resulting in unnecessary retransmissions, while a too large timer causes delays and messages may be lost. In RPC systems, servers may take varying amounts of time to service individual requests. The time required for execution depends on various factors like server load, network traffic, and routing policy. All these decide the time taken by the server to service a particular request. If the client continues to retry sending the requests for which the replies are not received, the server load will increase leading to network congestion. Hence, an appropriate selection of timeout value is important. A backing algorithm with exponentially increasing timeout values is used to handle this issue.

Design of RPC protocol specification

The protocol specification of an RPC system should be designed so as to minimize the amount of data sent across the network and the frequency of transmission of requests. There are two advantages of reducing the data to be transferred: (i) less time is required for encoding and decoding the data and (ii) it requires less time to transfer this data across the network. Many existing RPC systems use TCP/IP or UDP/IP as the basic protocol, since they are easy to use and fit well with existing networks like the Internet.

RPC optimization techniques include the concurrent access to multiple servers using early reply technique or call buffering, serving multiple requests simultaneously, reducing call workload of server, using reply cache for idempotent RPCs, selection of timeout values, and design of RPC protocol specification.

4.5.5 Complicated and Special RPCs

After understanding the working, implementation, and issues in designing a simple RPC, we now focus on a few complicated and special types of RPCs. We also discuss a few complicated RPCs like those with long duration calls or with gaps between calls and RPCs with long messages. We also discuss special RPCs like the callback RPC, broadcast RPC, and the batch mode RPC.

Complicated RPCs

Typical types of complicated RPCs are those involving long duration calls (or gaps between calls) and those involving long messages.

RPCs with long duration calls or with gaps between calls The protocols used to handle these complicated RPCs are: client probing the server periodically, and server generating acknowledgements periodically.

- In the first method, the client probes the server periodically. The client sends a request to the server and later probes the server, which has to respond with an acknowledgement. The client can detect a node or a link failure and raise a flag for an exception condition. The message ID is included in the probe message. If the request message is lost, the server sends the information regarding the loss of message, and the client retransmits the original request.
- In the second method, the server generates acknowledgements periodically. The client uses this method in case the server cannot generate the result within the retransmission interval. The number of acknowledgements is proportional to the number of calls. If the client does not receive an acknowledgement within a predetermined timeout period, it understands that the server has crashed or there is a communication link failure. In that case, the client generates an exception condition.

RPCs with long messages In some RPCs, the arguments and/or results are too large to fit into a single datagram packet. A typical example is that of a file server where a large quantity of data may be transferred as input argument to a write operation or as results to a read operation. If arguments and/or results do not fit into one message, the client sends multiple datagram messages. The client sends many physical RPCs for one logical RPC. Each physical RPC can transfer one datagram message. The server transmits long arguments or results by dividing this long message into multiple packets. Only one acknowledgement is sent for all multi-datagram messages, leading to performance improvement. For example, in Sun systems, the RPC size is limited to 8 kb. So in these systems, an RPC involving a message size greater than 8 kb is handled by being broken into several physical RPCs.

Special types of RPC

The RPC protocol between the caller and the called process obeys the client-server relationship. However, there are some special RPCs like callback, broadcast, and batch mode RPCs that perform special calls as described below.

Callback RPC This RPC allows a process to be both a client and a server. The client process makes an RPC to a server process. During an RPC call execution (this is RPC 1 execution), the server makes a callback to the client process. It now acts as a server (this is RPC 2 in execution) and accepts the call request, executes it, and sends the callback reply to the server process. Once the server receives the reply, it resumes execution of the RPC (RPC 1) and on completion, returns the result of the initial call to the client. It is possible that in some cases, the server can make several callbacks. Figure 4-21 shows a callback RPC.

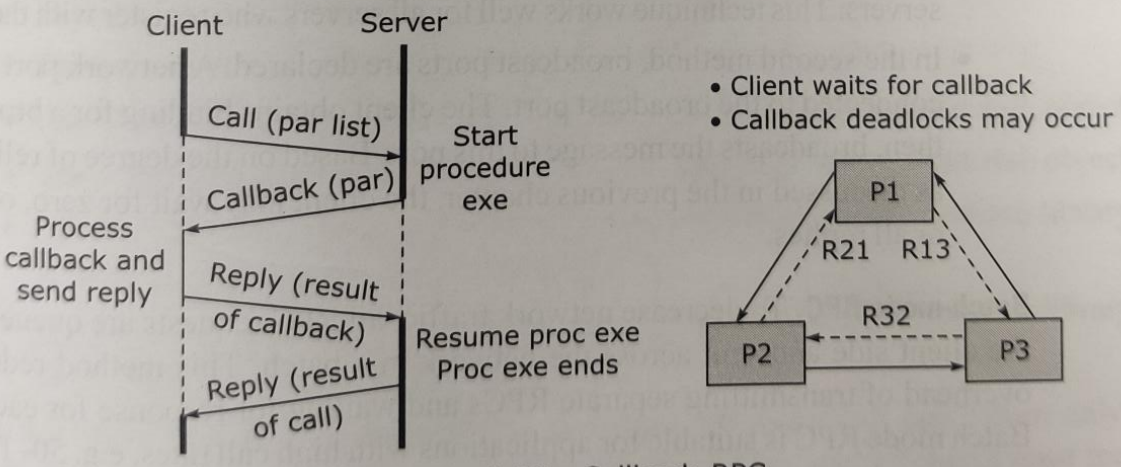

Figure 4-21 Callback RPC

To enable the server to callback the client, here are some of the necessary requirements:

- Client handle is provided to the server.
- Client process should wait for callback RPC.
- Call back deadlocks should be handled.

We describe each of these issues in this section.

Client handle is provided to the server The client handle identifies the client process so that the server can call the client. It could be a program number with which the client registers with the binding agent. The RPC request to the server includes the program number along with the handle-port number. Now, the server can directly communicate with the client.

Client process should wait for callback RPC The client process should give time to the server to process the request. The client itself will take sometime to process the request received from the server. It should not mistake the RPC call for a reply to its own request.

Handle callback deadlocks There is a possibility that deadlocks can occur because a process acts both as a client and a server. As shown in Figure 4-21,

- P1 makes an RPC call to P2 and waits for a reply from P2.
- P2 makes an RPC call to P3 and waits for a reply from P3.
- P3 makes an RPC call to P1 and waits for a reply from P1.

A cycle is complete if all the three processes end up waiting for each other, resulting in a deadlock state. We will discuss how to tackle this condition in subsequent chapters.

Broadcast RPC In this type of RPC, the client broadcasts the request on the network. All the servers having that particular process accept and execute the call request. The client receives multiple replies. The client uses two different techniques for broadcasting the RPC call.

- In the first method, the client uses a special broadcast primitive in the RPC call message. The request is sent to the binding agent which forwards it to the registered servers. This technique works well for all servers who register with the binding agent.
- In the second method, broadcast ports are declared. A network port of each node is connected to the broadcast port. The client obtains binding for a broadcast port and then, broadcasts the message to this port. Based on the degree of reliability desired, as discussed in the previous chapter, the client may wait for zero, one, *m*-out-of-*n*, or all replies.

Batch-mode RPC To decrease network traffic, all RPC requests are queued in a buffer on the client side and sent across the network in a batch. This method reduces the traffic overhead of transmitting separate RPCs and waiting for response for each one of them. Batch mode RPC is suitable for applications with high call rates, e.g. 50–100 remote calls per second. It is also ideal when no reply is expected for a sequence of requests. Requests are queued on the server side and the queue is flushed to the server after a predetermined time interval, or the predetermined requests are queued when the amount of batch data is greater than the buffer size. The queue flush operation is independent of the request and is transparent to the client. The batch mode RPC uses reliable transport mechanism.

4.6 Case Study: Sun RPC

Many RPC systems have been built and are in use today, such as Sun, Cedar, Courier, and Argus. Of these, the best known UNIX RPC system is Sun RPC which we briefly describe in this section.

Sun RPC uses automatic stub generation and also provides the users with the flexibility to write stubs manually. An application's interface definition is written in an IDL called RPCL, an extension of Sun XDR. It uses the Rpcgen compiler which generates the following:

- *A header file* containing definitions of common constants and types defined in the interface definition file, external declarations for marshalling and unmarshalling procedures which are automatically generated (denoted by **.h** suffix)
- *An XDR filter file* containing XDR marshaling and unmarshaling procedures which are used by client and server stub procedures (denoted by _xdr.c)
- *A client stub file* containing one stub procedure for each procedure defined in IDL
- *A server stub file* containing the main routine (creates transport handles and registers the service), the dispatch routine (dispatches incoming remote procedure calls to appropriate procedure), and a stub procedure for each procedure defined in the interface definition file plus a null procedure.

The RPC application is created using the files generated by the Rpcgen compiler, based on the following steps:

- The application programmer manually writes the client program as well as the server program
- The client program file is compiled to get a client object file
- The server program file is compiled to get a server object file
- The client stub file and the XDR filters are complied to get a client stub object file
- Server stub file and the XDR filters file are compiled to get a client stub object file
- Client object file, client stub object file, and client stub RPC runtime library are linked together to get a client executable file
- Server object file, server stub object file, and server stub RPC runtime library are linked together to get a server executable file

A Sun RPC remote procedure can accept only one argument and return only one result. Hence, procedures requiring multiple parameters as input or output must include them as components of a single structure. To handle differences in data representations, data structures are converted to XDR and back, using marshaling procedures. The RPC runtime library has procedures for marshaling integers of all sizes, characters, strings, reals, and enumerated types. Sun RPC supports 'at least once' semantics. After sending a request message, the RPC runtime library waits for a user-defined timeout period for

the server to reply before retransmitting the request. Each node uses a local binding agent called *portmapper* which maintains a database mapping of local services and their port numbers. The server, on startup, registers its program number, version number, and port number with the local port mapper. The client is supposed to find the port number of the server which supports the remote procedure.

The server-side error handling procedures which process the detected errors send a reply to the client indicating the detected error. The client-side error handling procedures provide flexibility to choose the error reporting mechanism. Sun RPC supports UNIX style authentication (restrict service access to certain set of users) and DES style authentication (secure RPC which uses DES technique). Sun RPC also supports asynchronous, callback, broadcast, and batch mode RPC. A few disadvantages of Sun RPC include location transparency, no general specification of procedure arguments and results, transport dependency, transport protocol being limited to either UDP or TCP, and finally, no support for network-wide binding service.

4.7 Remote Method Invocation Basics

Object-oriented models are efficient for developing non-distributed applications. The object interface hides the internal structure from the outside world. Therefore, it is easy to replace objects by maintaining the same interface. In distributed systems, RPC handles IPC. The principle of RPC can be applied to handle objects, i.e. invocation of remote objects. With this concept, objects in different procedures can communicate with each other through RMI or Remote Object Invocation. It is an extension of local method invocation which allows objects in one process to invoke methods of an object in another process on the same machine.

We first introduce the basic concept of distributed objects and then explain how they communicate with each other. In the next section, we explain the RMI process and the binding of clients to objects. The later sections describe the parameter passing techniques of RMI, followed by various types of RMI and a case study of Java RMI.

4.7.1 Distributed Object Concepts

Objects consist of a set of data and its methods. Objects encapsulate data called the *state*. An object invokes another object by invoking its methods, i.e. passing arguments and parameters. Methods are various operations performed on data, which can be availed through interface. Method invocation accesses or modifies the state of the object. Users can avail the methods through the object's interface.

Multiple interfaces can also implement objects. Objects are efficiently used in a distributed system by being accessed only through their methods remotely.

Objects in other processes can invoke methods that only belong to its remote interface. As shown in Figure 4-22, local objects are invoked in the remote interface and in other methods implemented by the remote object.

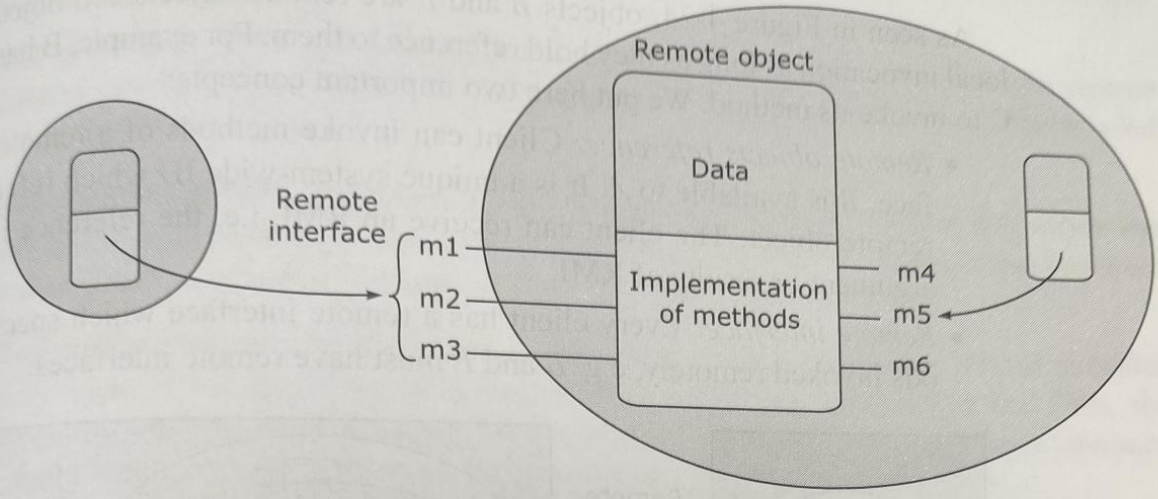

Figure 4-22 Remote object and remote interface

Figure 4-23 shows how processes across the network share objects. A distributed system uses the client-server architecture. The server manages the objects and clients invoke the methods – called the RMI. The RMI technique sends the request as a message to the server which executes the method of the object and returns the result message to the client. The state of the object is accessed only by the method of the object, i.e. unauthorized methods are not allowed to act on the state of the object.

We now take an example to explain the RMI operation. The RMI process is similar to Local Method Invocation (LMI). In terms of remote execution, it is similar to RPC with the difference that RPC works with procedures, while method invocation works with objects and methods. Each process contains a collection of objects; some of these can receive both local and remote invocations, while others can receive only local invocations. Method invocation between objects in the same process is called *local invocation,* while those between different processes are called *remote invocations*.

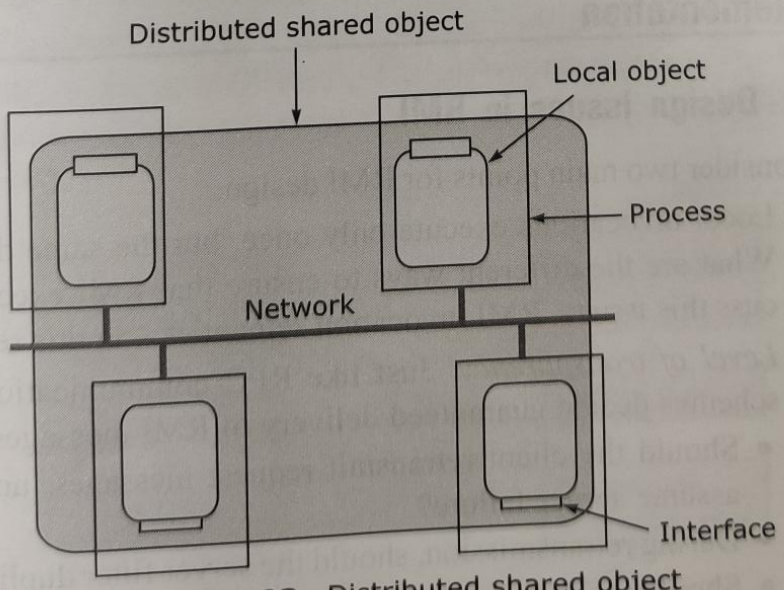

Figure 4-23 Distributed shared object

As seen in Figure 4-24, objects B and F are remote objects. All objects can receive local invocation as long as they hold reference to them. For example, B has a reference to C to invoke its method. We put here two important concepts:

- *Remote objects reference:* Client can invoke methods of a remote objects interface. B is available to A. It is a unique system-wide ID which refers to a specific remote object. The client can receive an RMI, i.e. the reference is passed as an argument or results of RMI.
- *Remote interface:* Every client has a remote interface which specifies the methods invoked remotely, e.g. B and F must have remote interfaces.

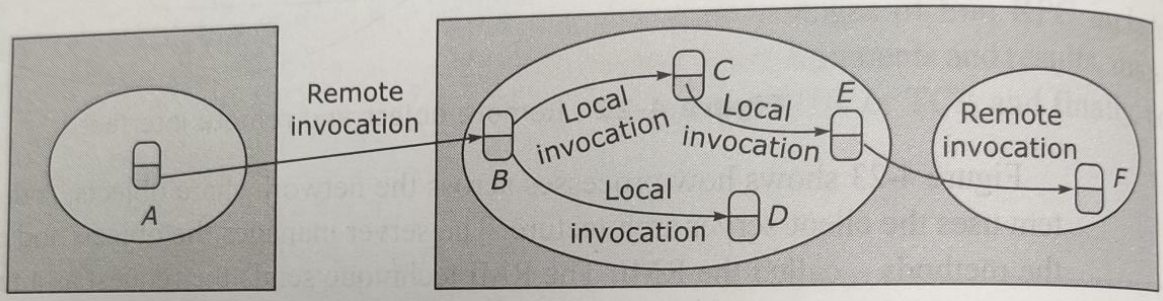

Figure 4-24 RMI and LMI

The heterogeneous nature of distributed systems adheres to different data formats at distributed sites. This heterogeneity is transparent to the client because methods are used to access remote objects. Objects that can receive remote invocations are called remote objects.

RMI allows the invocation of a remote object on another machine by calling its methods.

4.8 RMI Implementation

4.8.1 Design Issues in RMI

We consider two main points for RMI design:

(a) Local invocations execute only once, but the same does not hold true for RMI. What are the different ways to ensure that RMI executes exactly once? We discuss this issue – RMI invocation semantics – in this section.

(b) *Level of transparency:* Just like RPC communication protocols, the following schemes decide guaranteed delivery of RMI messages:

- Should the client retransmit request messages, until it receives the reply or assume server failure?
- During retransmission, should the server filter duplicate requests?
- Should the server cache the results and use them during retransmission?

RMI invocation semantics

Maybe semantics With this semantic, the client may not know whether the remote method is executed once or not at all. This semantic is useful in applications where failed invocations are acceptable.

At-least-once semantics The invoker receives the result, implying that the server has executed the method at least once, or an exception informing that no result was received. Retransmission of request messages is used to achieve this semantic.

At-most-once semantics The client receives the result, implying that the method executes exactly once, or an exception informing that no result was received so far. Thus, the method executes once or not at all. This semantic is achieved by using fault tolerance measures. Table 4-2 describes various invocation semantics.

Table 4-2 Invocation semantics

Fault-tolerance measures			Invocation semantics
Retransmit request message	Duplicate filtering	Re-execute procedure of retransmit reply	
No	Not applicable	Not applicable	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

Choosing a mix of the above schemes provides RMI reliability. The RMI flow is depicted in Figure 4-27. Normally, LMI uses exactly-once semantics.

Level of transparency

This involves hiding, marshalling, message passing, locating, and contacting the remote object for the client. RMI is more prone to failures than LMI. In spite of choosing reliable semantics, it is still possible that the client will not receive the reply. The reason could be failure of network, node, or RMI itself. The difference between the local and remote method invocation should be implemented in the interface. A typical RMI flow diagram is shown in Figure 4-25.

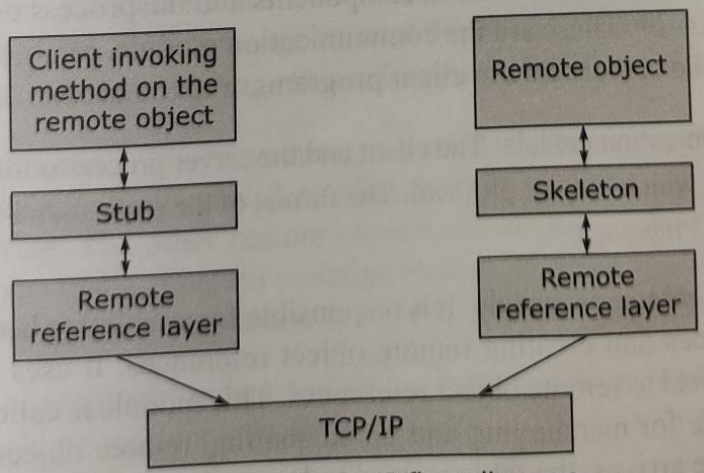

Figure 4-25 RMI flow diagram

As shown in Figure 4-26, when the client binds to the distributed object, the object interface called the *proxy* is loaded in the client address space. A proxy is similar to the client-server stub of RPC. It marshals the RMI request into a message before sending it to a server and unmarshals the reply message of method invocation when it arrives at the client site.

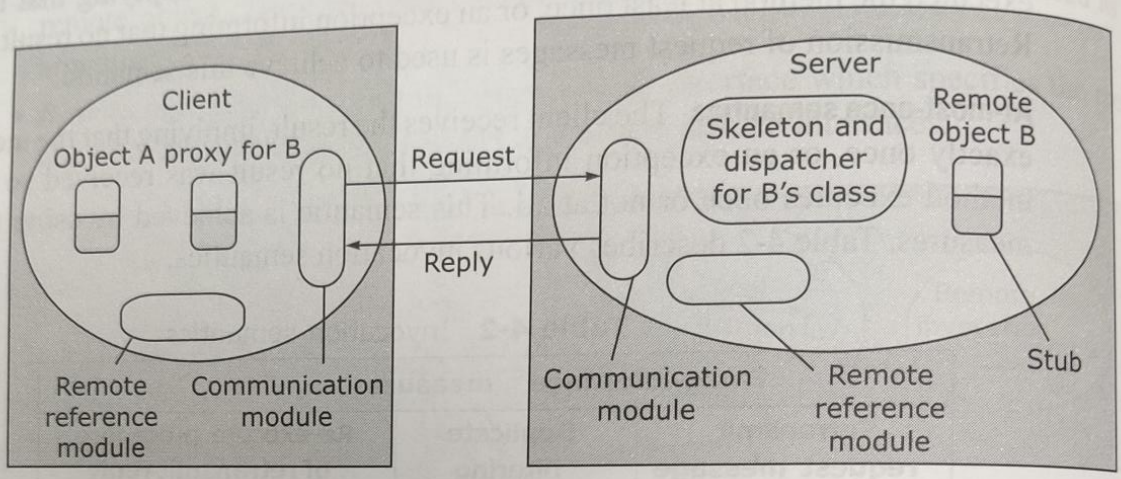

Figure 4-26 RMI components

The server-side machine invokes the object and offers the same interface as if it resides on the client machine. The other function of the server stub-skeleton is to marshal replies, convert them into a message, and send it to the client-side proxy. The object state resides on a single machine, while the interfaces are available on another machine. Hence, a distributed object implies that the state and interface are distributed across the machines, but it is transparent to the user by hiding behind the object's interface.

The design issues in RMI include RMI reliability semantics like maybe invocation, at-least-once, and at-most-once semantics. The next issue is the level of transparency that decides the guaranteed delivery of messages.

4.8.2 RMI Execution

We now discuss the various components and the process of RMI execution, as shown in Figure 4-27. These are the communication module, remote reference module, RMI software, the server and the client programs, and the binder.

Communication module The client and the server processes form a part of the communication module which use RR protocol. The format of the request-and-reply message is very similar to the RPC message.

Remote reference module It is responsible for translating between local and remote object references and creating remote object references. It uses a remote object table which maps local to remote object references. This module is called by the components of RMI software for marshalling and unmarshalling remote object references. When a request message arrives, the table is used to locate the object to be invoked.

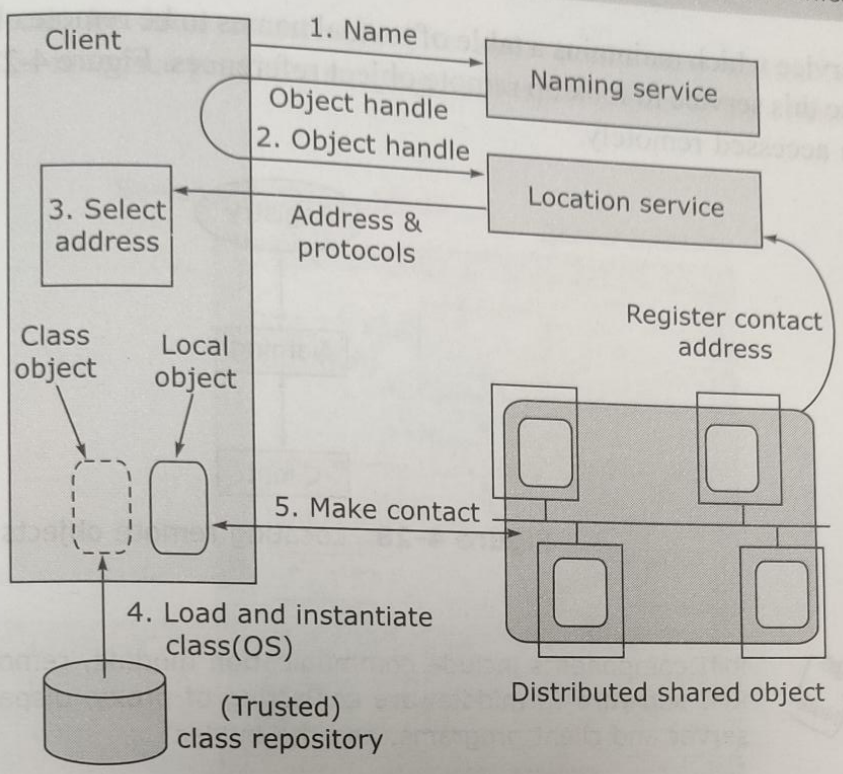

Figure 4-27 RMI implementation

RMI software This is the middleware layer and consists of the following:

- *Proxy:* It makes RMI transparent to the client and forwards the message to the remote object. It marshals the arguments and unmarshals the result and also sends and receives messages from the client. There is one proxy for each remote object for which it holds a remote reference.
- *Dispatcher:* A server consists of one dispatcher and a skeleton for each class which represents a remote object. This unit receives the request message from the communication module, selects the appropriate method in the skeleton, and passes the request message.
- *Skeleton:* It implements the method in the remote interface. It unmarshals the arguments in the request message and invokes the corresponding method in the remote object. After the invocation is complete, it unmarshals the result along with excerptions in the reply message to send the proxy's method. The interface compiles automatically generating classes for proxy, dispatcher, and skeleton.

Server and client programs The server program contains classes for dispatcher, skeleton, and the remote objects it supports. A section creates and initializes at least one object hosted by the server. The other remote objects are created later when requests come from the client. The client program contains classes of processes for the entire remote object, which it will invoke. The client looks up the remote object reference using a binder.

The binder The concept of binders can be explained with an example. An object *A* requests remote object reference for object *B*. The binder in a distributed system is a

service which maintains a table of textual names to be remote object-referenced. Servers use this service to look up remote object references. Figure 4-28 shows how objects can be accessed remotely.

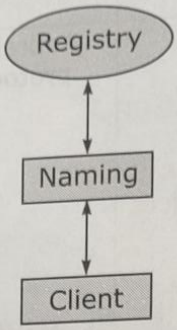

Figure 4-28 Locating remote objects

RMI components include communication module, remote reference module, RMI software (a middleware consisting of proxy, dispatcher, and skeleton), server and client programs, and the binder.

4.8.3 Types of Objects

Objects are classified in two main classes based on when they are bound and how long they exist. Based on the time of binding, objects are classified as *runtime* and *compile-time*, while objects can be transient or persistent based on how long they exist. Both these types are shown in Figure 4-29.

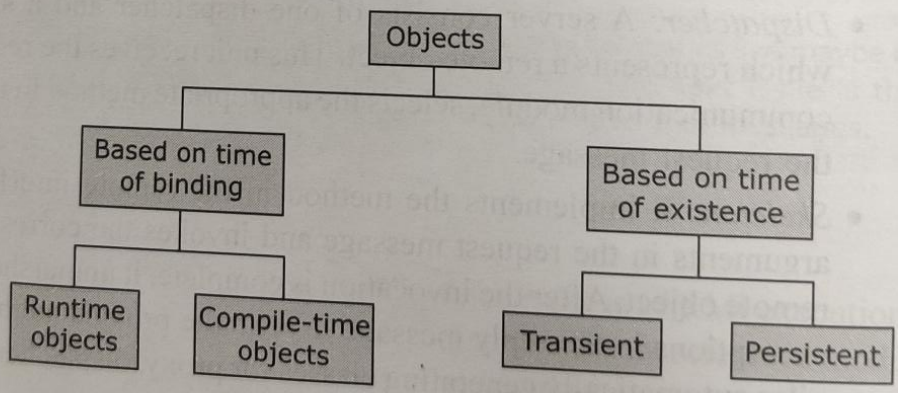

Figure 4-29 Types of objects

Runtime objects Distributed applications are built easily using compile-time objects. These types of objects are dependent on the programming language. To avoid this dependency, they are bound at runtime. So applications can be reconstructed from objects written in multiple languages.

Compile-time objects The object adapter acts as a *wrapper* around the implementation details to give it the appearance of an object. An interface is defined to implement the

object which can be later registered. The objects are now available for remote invocations. The interface provides an image of remote object to the client, as shown in Figure 4-30.

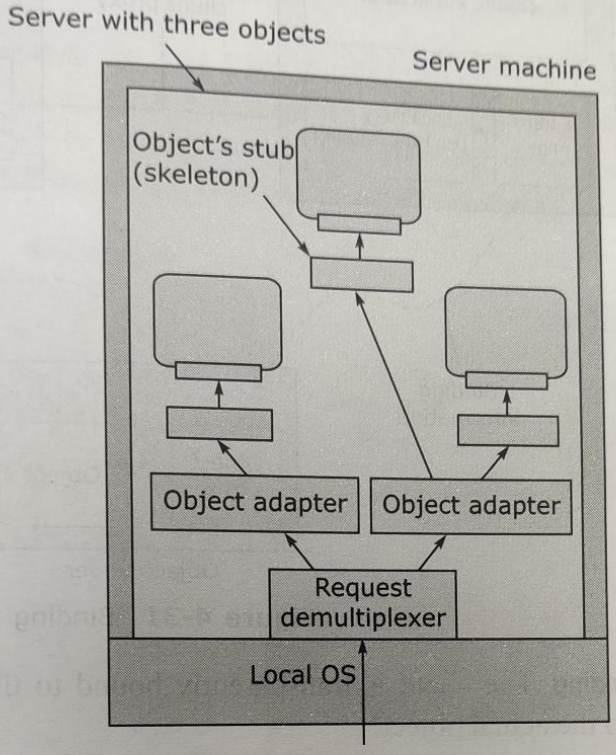

Figure 4-30 Object adapter

Persistent objects These objects exist even if they are not contained in the server process address space. It implies that the client manages the persistent object and can store its state on any secondary storage and exit. Now when a new server needs the object's state, it copies from the secondary storage device into its own address space and handles invocation requests.

Transient objects They exist only for the time when the server manages the object. When the server ceases to exist, the object gets vanished.

Based on the time of binding, objects are classified as runtime and compile-time objects, and as transient or persistent based on when they are bound and how long they exist.

4.8.4 Binding a Client to an Object

As compared to RPC, RMI provides system-wide object reference so that objects can be freely associated between processes on different machines. A proxy placed in the process address space implements the interface containing the method, which the process can invoke. Figure 4-31 explains the process of binding a client to an object. Bindings can be classified as *implicit* and *explicit*.

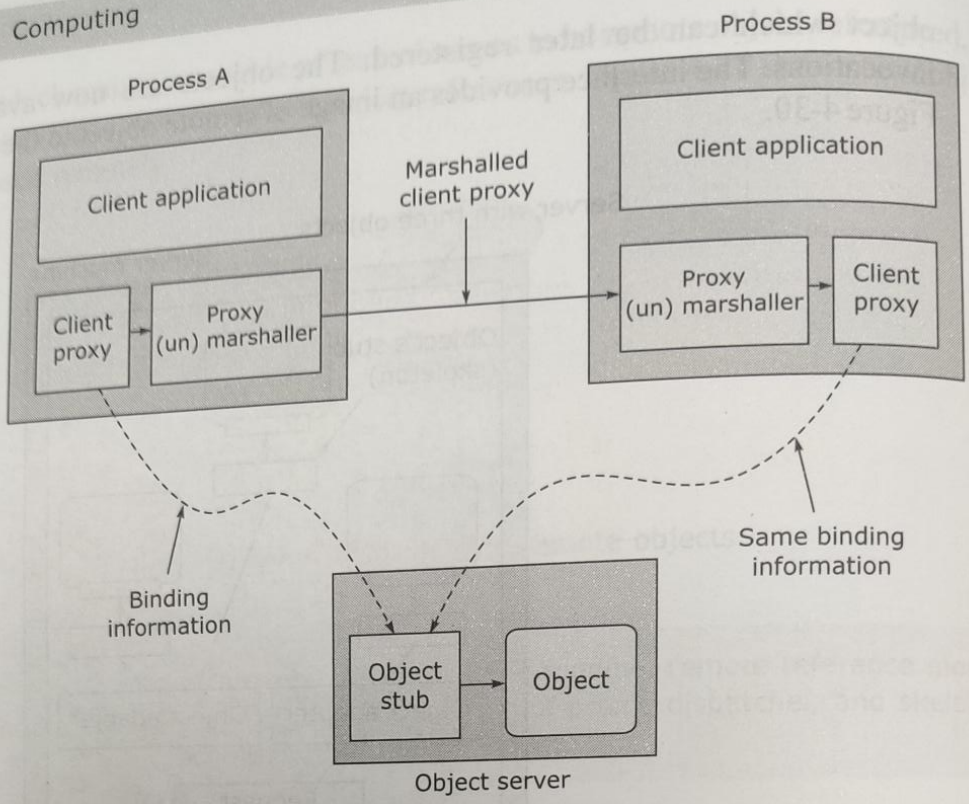

Figure 4-31 Binding

Implicit binding The client is transparently bound to the object when the reference is resolved to the actual object.

Explicit binding The client first calls a specific function to bind the object before the method invocation.

4.8.5 RMI Parameter Passing

RMI system supports system-wide object references and hence uses parameter passing techniques. Assume a situation where all objects can be accessed from remote machines. Hence, object references can be used as parameters in RMI. References are passed by value, i.e. copied from machine to machine. When the object reference is given to the process, it binds to the referred object when needed. However, this method is inefficient for very small distributed objects like Boolean or integers. A request is generated to a different address space or a different machine, if the invocation is required to be made to a remote machine. Hence, a reference to a local or a remote machine is treated differently. When the method is invoked with the object reference as a parameter, this reference is copied and passed as a value if it refers to the remote object. In fact, the object is passed as reference. However, if the object is local, i.e. in same address space as the client, it is copied as a whole and passed with the invocation. Such local objects are passed by value.

RMI binding can be classified as explicit and implicit, while parameter passing can be pass-by-value or pass-by-reference.

4.9 Case Study: Java RMI

Java hides the difference between local and remote method invocation from the user. After marshalling the type of object, it is passed as a parameter to the RMI (called *serializable*) in Java.

Figure 4-32 shows how the Java RMI is implemented. The reference to remote object consists of the network address and endpoint of server. It also contains the local ID for the actual object in the server address space. Each object in Java is an instance of a class, which contains the implementation of one or more interfaces.

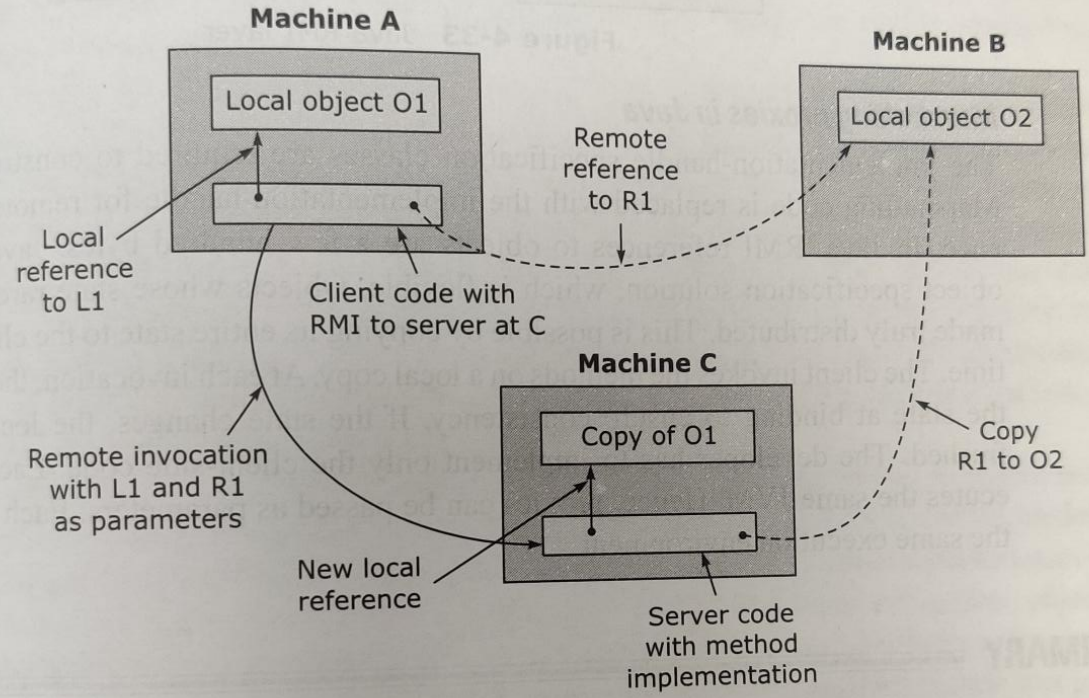

Figure 4-32 Java remote object

A Java remote object is built from two classes: (i) server class and (ii) client class.

Server class This class contains the implementation of server-side code, i.e. objects which run on the server. It also consists of a description of the object state and the implementation of methods which operate on that state. The skeleton on the server-side stub is generated from the interface specification of the object.

Client class This class contains the implementation of client-side code and proxy. It is generated from the object interface specification. The proxy basically converts each method call into a message that is sent to the server-side implementation of the remote object.

We now describe how proxies can be used as referenced to remote objects. Communication is set up to the server and is cut down when a call is complete. The proxy state contains the server's network address, end to the server and local ID of the object. Hence, a proxy consists of sufficient information to invoke the methods of a remote

object. Proxies are serializable in Java, marshalled and sent as a series of bytes to another process where it is unmarshalled, and can invoke methods in remote objects. Java RMI layer is depicted in Figure 4-33.

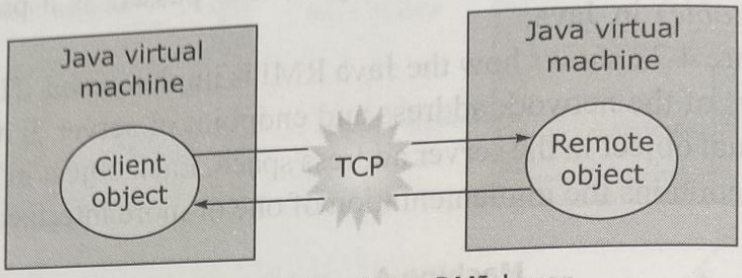

Figure 4-33 Java RMI layer

Marshalling proxies in Java

The implementation-handle specification classes are required to construct the proxy. Marshalling code is replaced with the implementation-handle for remote object reference. In Java, RMI references to objects are a few hundred bytes. Java RMI allows object specification solution, which is flexible. Objects whose state rarely change are made truly distributed. This is possible by copying its entire state to the client at binding time. The client invokes the methods on a local copy. At each invocation, the client checks the state at binding to ensure consistency. If the state changes, the local copy is refreshed. The developer has to implement only the client-side code. Each process executes the same JVM. Hence, proxies can be passed as parameters. Each machine uses the same execution environment.

SUMMARY

Distributed systems widely use the RPC model of IPC. It is an extension of the local procedure call. Transparency in RPC implies that the client cannot distinguish between remote and local procedure calls. The components of RPC implementation are: client, client stub, RPC runtime, server stub, and the server. The main function of the client and the server stub is to pack and unpack the RPC parameter into a message and vice versa and transmit over the network. The RPC runtime system provides transparency.

RPC communication takes place with Call or Request and Reply messages. The call messages are used to request the server to execute a remote procedure. The reply messages pass the result of the executed remote procedure to the client. Messages are encoded and decoded through a process called marshalling. Servers are implemented as stateful or stateless servers. Based on the time for which the servers exist, they are classified as instance-per-call, instance-per-session and persistent servers. The server is chosen based on the application. RPC mechanism can pass parameters using call-by-value, call-by-reference, or call-by-copy. Call semantics of RPC depend on remote procedure execution under fault conditions. The various call semantics are: maybe, last-one, last-of-many, at-least-once, and exactly-once.

Depending on the IPC needs, any of the following communication protocols is used: the Request (R) protocol, the Request-Reply (RR) protocol, the Request Reply/Acknowledgement (RRA) protocol. Special protocols used to handle complicated RPCs have large duration calls, long gaps between calls, and large arguments or results. Clients can be bound to a server at compile time, link time, or call time. Some RPCs like the synchronous RPC send a one-way message from the client to the server, while the callback RPC provides a peer-to-peer relation between the processes. Similarly, batch mode RPC allows client requests to be batched.

Performance of distributed systems can be improved by various optimization techniques in case of concurrent access to multiple servers, serving multiple requests simultaneously, reducing per-call workload of servers, reply caching of idempotent remote procedures, and proper selection of timeout values.

The distributed object model is an extension of the local object model. Each object in a distributed system has a global unique ID and a remote interface, specifying which operations can be remotely invoked. The RMI technique sends requests as a message to the server, which executes the methods of the object and returns the result message to the client. The two main design issues are: invocation semantics and level of transparency. Local invocations execute only once, but the same does not hold true for RMI. The invocation semantics used are: maybe invocation, at-least-once semantic, and at-most-once semantic. Just like RPC communication protocols, transparency involves hiding the marshalling, message passing, and the task of locating and contacting the remote object for the client.

The various components of RMI are: communication module, remote reference module, RMI software (proxy, dispatcher, skeleton), server and client programs, and the binder. The binding process can be implicit or explicit. Objects are classified into two main classes based on when they are bound and how long they exist. Based on the time of binding, objects are classified as runtime objects and compile-time objects, while objects can be transient or persistent based on how long they exist. In RMI, the parameters can be passed-by-value or passed-by-reference. Java RMI hides the difference between local and remote method invocation from the user. A Java remote object is built from two classes: client class and server class. Proxies are serializable in Java, marshalled and sent as a series of bytes to another process where it is unmarshalled and can invoke methods in the remote object.

EXERCISE

Objective Questions

Select the correct option(s).

1. Distributed system middleware is designed using programming languages like
   (a) Java
   (b) CORBA
   (c) IDL
   (d) IML