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.

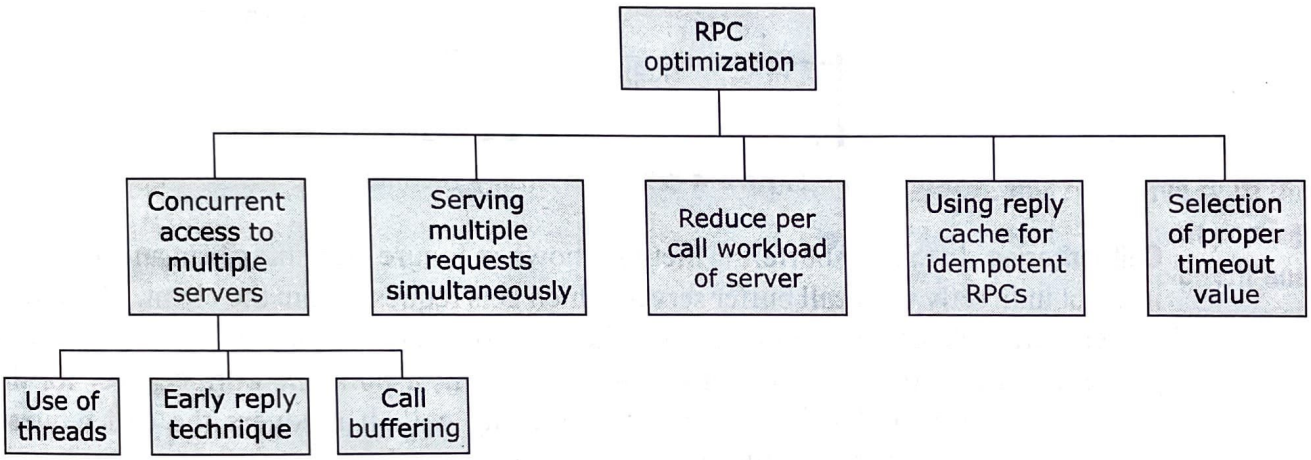

Figure 4-18 Techniques for RPC optimization

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.

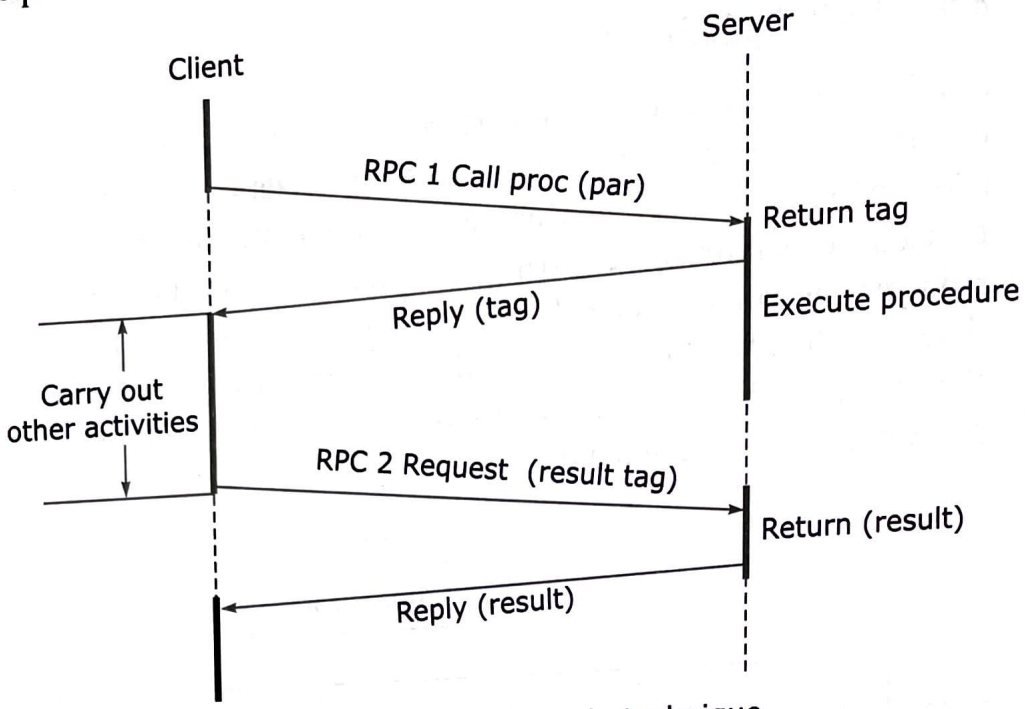

Figure 4-19 Early reply technique

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.

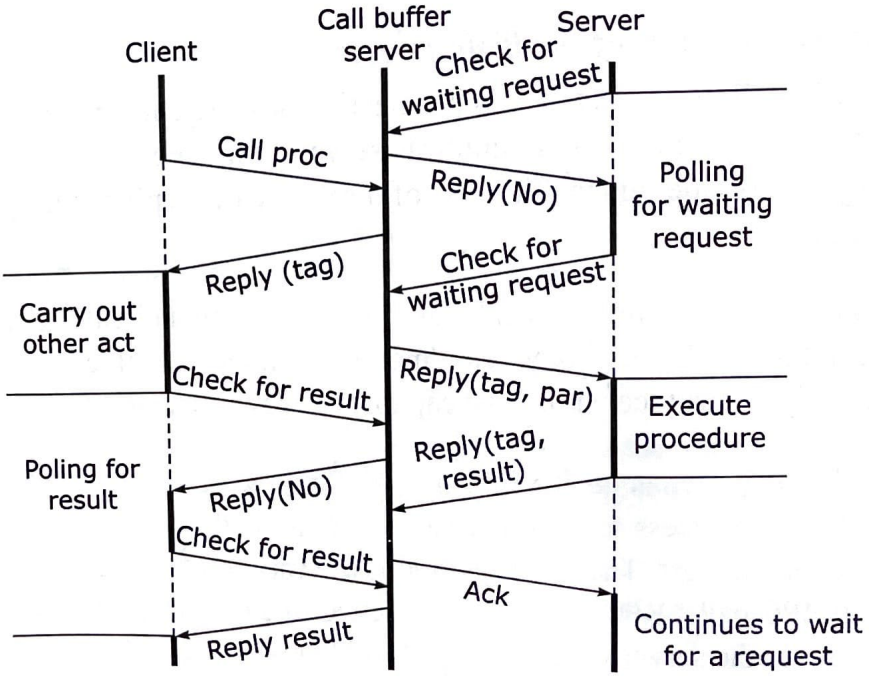

Figure 4-20 Call-buffer approach for concurrent access to multiple servers

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.

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.

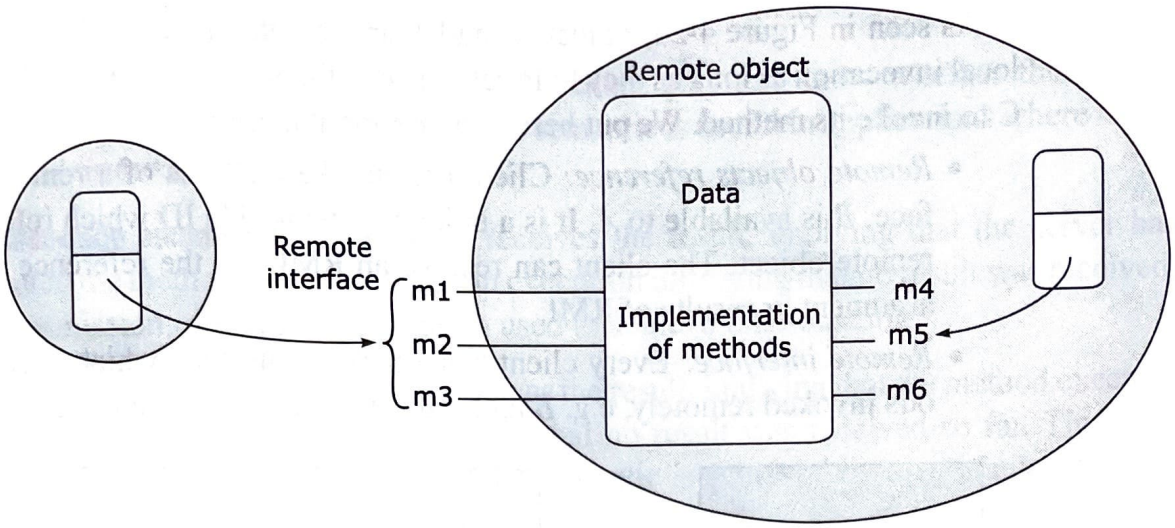

Figure 4-22 Remote object and remote interface

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*.

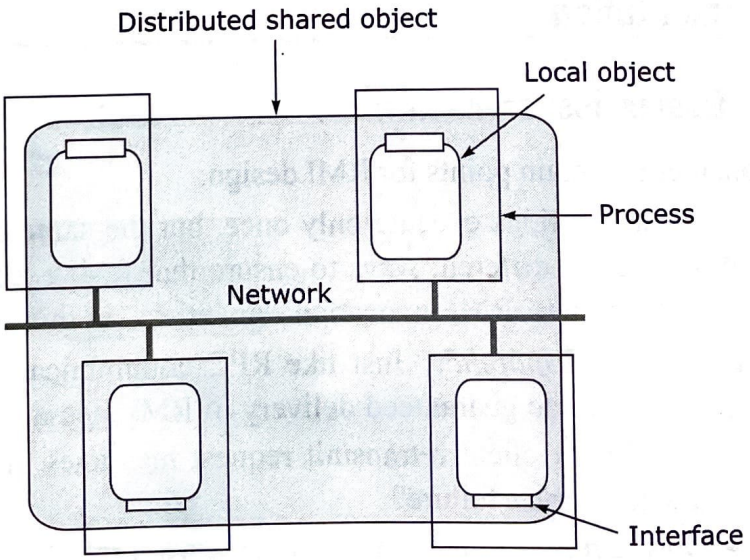

Figure 4-23 Distributed shared object

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.

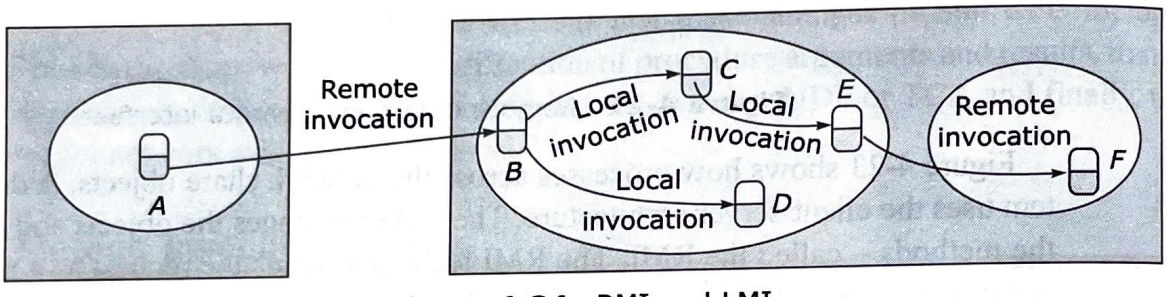

Figure 4-24 RMI and LMI

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.

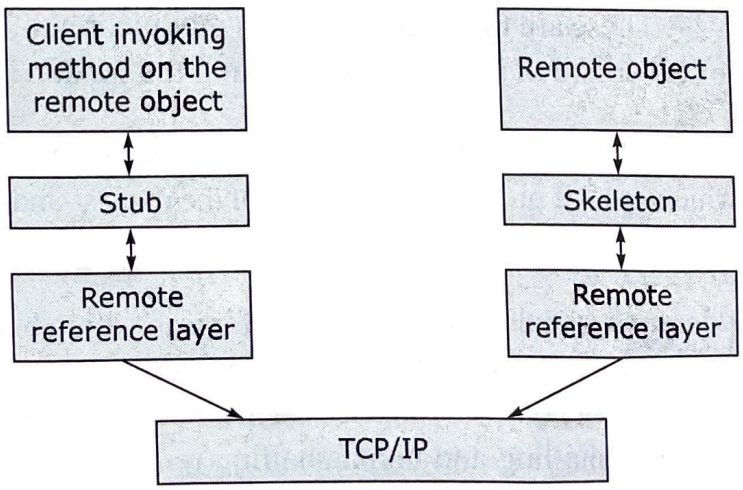

Figure 4-25 RMI flow diagram

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.

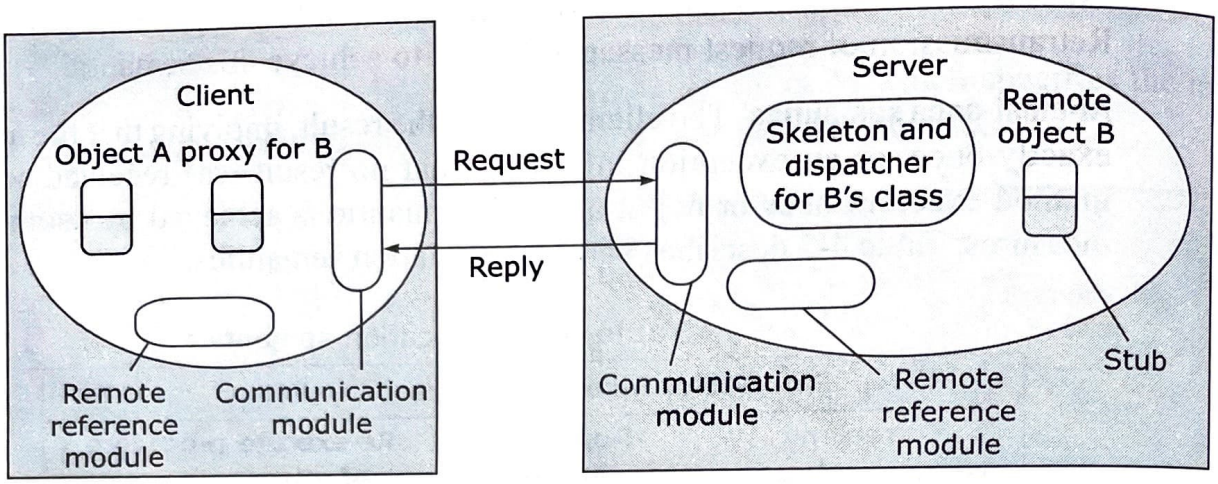

Figure 4-26 RMI components

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.