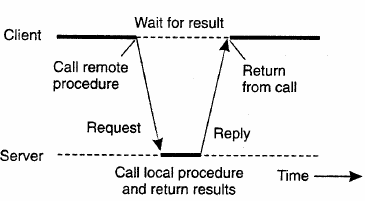
**Remote Procedure call**

A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

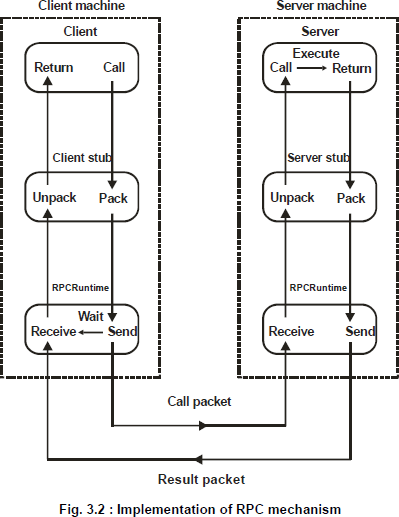
It further aims at hiding most of the intricacies of message RPC allows programs to call procedures located on other machines. But the procedures ‘send’ and ‘receive’ do not conceal the communication which leads to achieving access transparency in distributed systems.

Example: when process A calls a procedure on B, the calling process on A is suspended and the execution of the called procedure takes place. (PS: function, method, procedure difference, stub, 5 state process model definition)Information can be transported in the form of parameters and can come back in procedure result. No message passing is visible to the programmer. As calling and called procedures exist on different machines, they execute in different address spaces, the parameters and result should be identical and if machines crash during communication, it causes problems.



**Client Stub:** Used when read is a remote procedure. Client stub is put into a library and is called using a calling sequence. It calls for the local operating system. It does not ask for the local operating system to give data, it asks the server and then blocks itself till the reply comes.

**Server Stub:** when a message arrives, it directly goes to the server stub. Server stub has the same functions as the client stub. The stub here unpacks the parameters from the message and then calls the server procedure in the usual way.

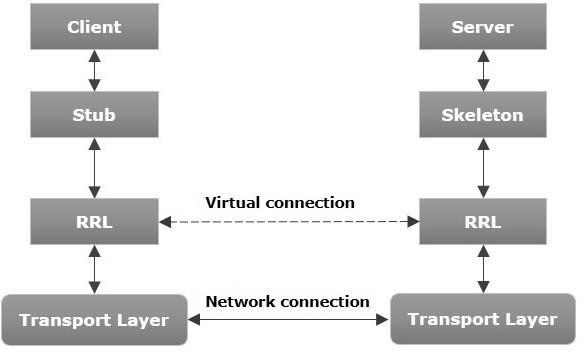


**Remote Method Invocation (RMI)**

RMI stands for Remote Method Invocation. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java.rmi.

The following diagram shows the architecture of an RMI application.



**Marshalling and Unmarshalling**

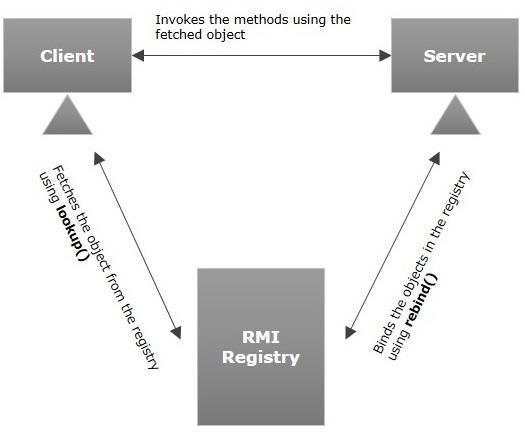
Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as marshalling.

At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as unmarshalling.

**RMI Registry**

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using bind() or reBind() methods). These are registered using a unique name known as bind name.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using lookup() method).



**Code:**

**RMI\_Interface:**

**package pkg\_RMI;**

**import java.rmi.Remote;**

**import java.rmi.RemoteException;**

**public interface RMI\_Interface extends Remote {**

**void displayMessage() throws RemoteException;**

**int factorial(int n) throws RemoteException;**

**}**

**RMI\_Client:**

**package** pkg\_RMI;

**import** java.net.MalformedURLException;

**import** java.rmi.RemoteException;

**import** java.util.Scanner;

**import** java.rmi.NotBoundException;

**import** java.rmi.Naming;

**public** **class** RMI\_Client {

**public** **static** **void** main(String[] args) **throws** MalformedURLException, RemoteException, NotBoundException{

**try** {

RMI\_Interface helloAPI = (RMI\_Interface) Naming.*lookup*("rmi://localhost:1880/hello");

// helloAPI.displayMessage();

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter a number to find the factorial: ");

**int** n = sc.nextInt();

**int** ans = helloAPI.factorial(5);

System.***out***.println("Factorial of "+n+" is "+ans);

}

**catch**(Exception e)

{

System.***out***.println("The RMI APP is not running...");

e.printStackTrace();

}

}

}

**RMI\_Server:**

**package** pkg\_RMI;

**import** java.rmi.AlreadyBoundException;

**import** java.rmi.RemoteException;

**import** java.rmi.registry.LocateRegistry;

**import** java.rmi.registry.Registry;

**import** java.rmi.server.UnicastRemoteObject;

**public** **class** RMI\_Server **extends** UnicastRemoteObject **implements** RMI\_Interface{

**public** RMI\_Server() **throws** RemoteException {

**super**();

}

**public** **static** **void** main(String[] args)**throws** RemoteException, AlreadyBoundException {

**try** {

Registry registry = LocateRegistry.*createRegistry*(1880);

registry.bind("hello", **new** RMI\_Server());

System.***out***.println("The RMI\_Server is running and ready...");

}

**catch** (Exception e) {

e.printStackTrace();

System.***out***.println("The RMI\_Server is not running...");

}

}

@Override

**public** **void** displayMessage()**throws** RemoteException{

System.***out***.println("----------------------------------------------");

System.***out***.println("Hello Akash!");

System.***out***.println("----------------------------------------------");

}

@Override

**public** **int** factorial(**int** n)**throws** RemoteException{

**int** fact = 1;

**for**(**int** i=1;i<=n;i++) {

fact \*= i;

}

**return** fact;

}

}

