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.

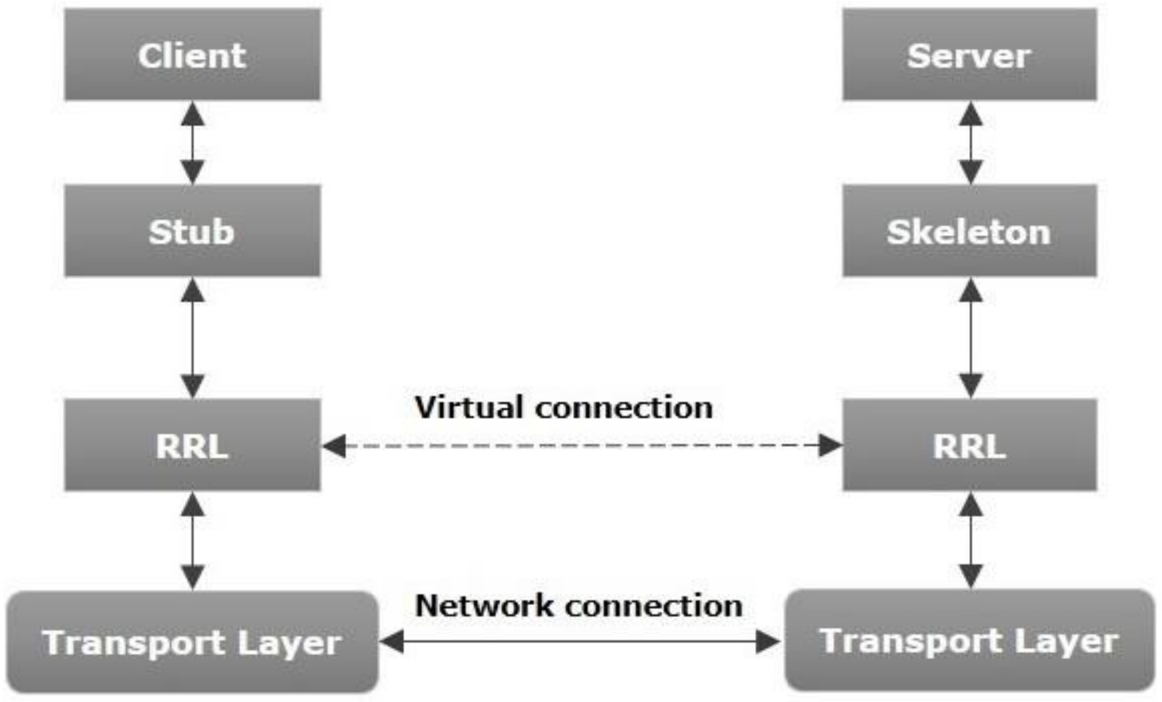
RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Architecture of an RMI Application

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture.

- **Transport Layer** − This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** − This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer)** − It is the layer which manages the references made by the client to the remote object.

Working of an RMI Application

The following points summarize how an RMI application works −

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

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**.

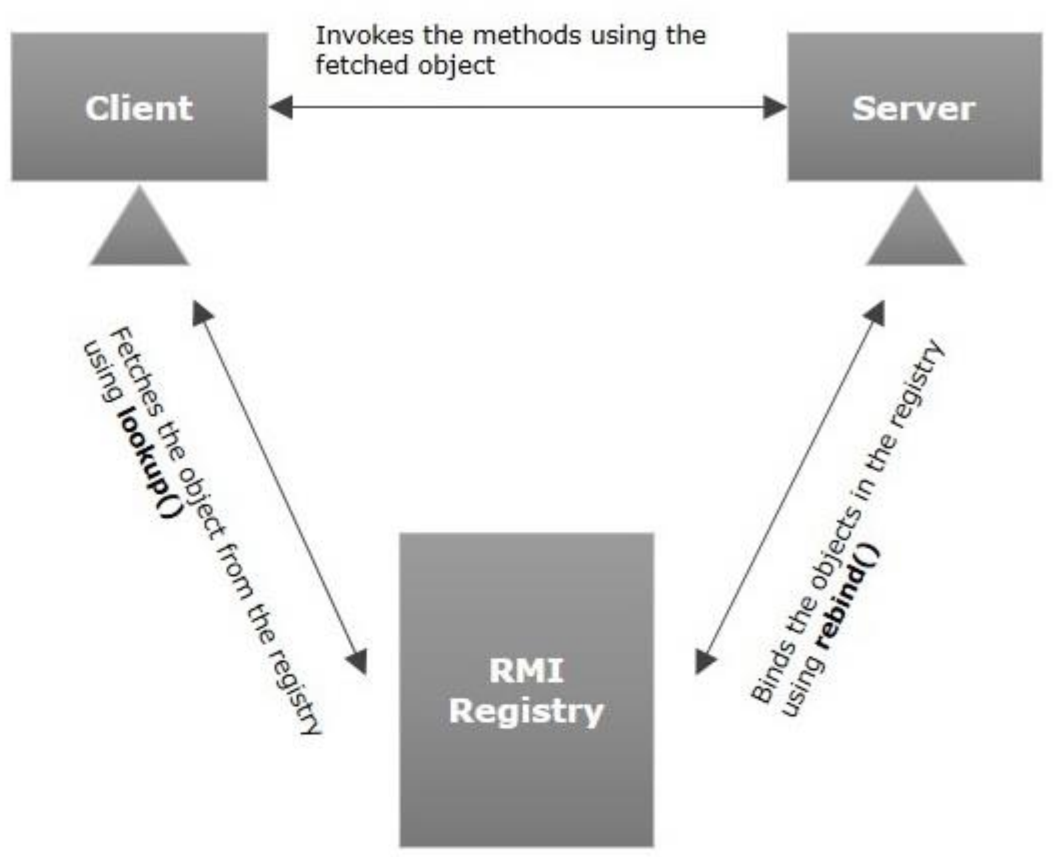
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

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).

The following illustration explains the entire process −


Invokes the methods using the fetched object
Client
Server
Fetches the object from the registry using lookup()
Binds the objects in the registry using rebind()
RMI Registry

Goals of RMI

Following are the goals of RMI −

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

<u>**Operations On RMI**</u>

The is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

Code Mobility

In distributed computing, **code mobility** is the ability for running programs, code or objects to be migrated (or moved) from one machine or application to another.[1] This is the process of moving **mobile code** across the nodes of a network as opposed to distributed computation where the *data* is moved.

It is common practice in distributed systems to require the movement of code or processes between parts of the system, instead of data.

Examples of code mobility include scripts downloaded over a network (for example JavaScript, VBScript), Java applets, ActiveX controls, Flash animations, Shockwave movies (and Xtras), and macros embedded within Microsoft Office documents.

The purpose of code mobility is to support sophisticated operations. For example, an application can send an object to another machine, and the object can resume executing inside the application on the remote machine with the same state as it had in the originating application.

According to a classification proposed by Fuggetta, Picco and Vigna. code mobility can be either strong or weak: *strong code mobility* involves moving both the code, data and the execution state from one host to another, notably via a process image (this is important in cases where the running application needs to maintain its state as it migrates from host to host), while *weak code mobility* involves moving the code and the data only. Therefore, it may be necessary to restart the execution of the program at the destination host.

Several paradigms, or *architectural styles*, exist within code mobility:[1]

- Remote evaluation — A client sends code to a remote machine for execution.
- Code on demand — A client downloads code from a remote machine to execute locally.
- Mobile agents — Objects or code with the ability to migrate between machines autonomously.

Writing RMI Services - Writing RMI Client- Developing of RMI

Remote Method Invocation Programe (Addition of two No)

Calculator.java

```
import java.rmi.Remote;

import java.rmi.RemoteException;

public interface Calculator extends Remote

{   public int add(int a,int b) throws RemoteException;

  }
```

CalculatorImpl.java

```
import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class CalculatorImpl extends UnicastRemoteObject implements Calculator

{     protected CalculatorImpl() throws RemoteException

  {      super();     }

       public int add(int a,int b) throws RemoteException

       {

           return a+b;

    }  }
```

CalculatorClient.java

```
import java.rmi.Naming;
public class CalculatorClient
{       public static void main(String a[])
  {        try
          {          Calculator c=(Calculator) Naming.lookup("//localhost:5000/sonoo");
                     System.out.println("Addition : " ,c.add(5,5));        }
   catch(Exception e)
 {
    System.out.println(e);
  }
  }    }
```

CalculatorServer.java

```
import java.rmi.Naming;

public class CalculatorServer

 {    public CalculatorServer()

   {      try

          {            Calculator c=new CalculatorImpl();

                   Naming.rebind("rmi://localhost:5000/sonoo",c);         }

          catch(Exception e)

          {       System.out.println(e);              }      }

          public static void main(String a[])

          {       new CalculatorServer();        }

}
```

Execution Steps:

1) First Compile All 4 Programs
2) create stub and skeleton object by rmic tool

   rmic CalculatorImpl

3) start rmi registry in separate one command prompt

   rmiregistry 5000

4) start the server in another command prompt

   java MyServer

5) start the client application in another command prompt

   java MyClient