Chapter-8
Distributed Application using RMI and CORBA

Introduction to Distributed Application

- A distributed system is a collection of independent computers that appear to the users of the system as a single computer.
- In such system, computers connected via network communicate and coordinate their action via message passing.
- Example Internet, ATM etc.
- A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal.

Distributed Application

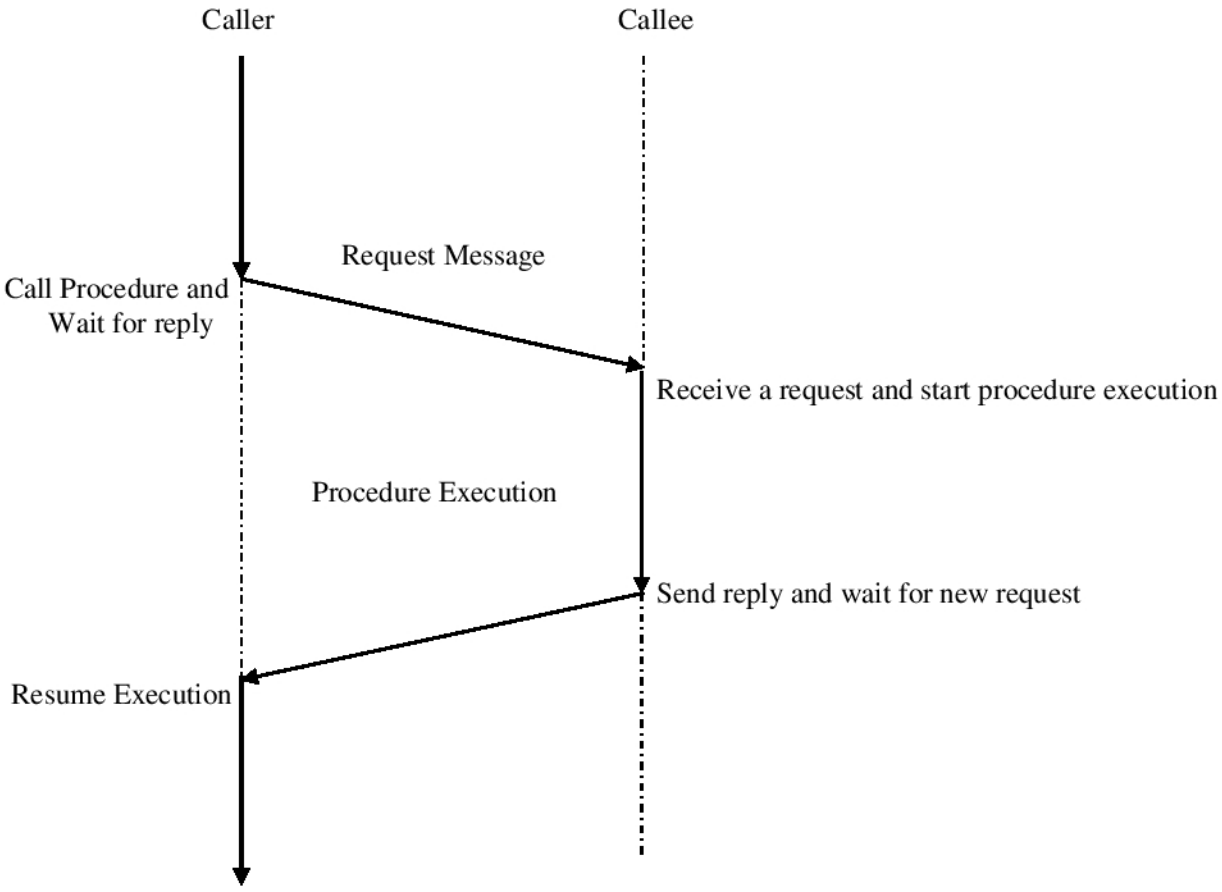
- Software that executes on two or more computers in a network. In a client-server environment, distributed applications have two parts:
  1. The 'front end' that requires minimal computer resources and runs on the client computer(s)
  2. The 'back end' that requires large amounts of data crunching power and/or specialized hardware, and runs on a suitably equipped server computer.
- So a distributed application is an application A, the functionality of which is divided into a set of co-operating sub-components A1, A2…An with n>1such that each Ai component are autonomous processing units which can run on different computers and exchange information over the network controlled by operation system called Distributed operating system.

Distributed Computing Technologies

Some of the standard technologies that are used for developing and deploying the distributed applications are explained below.

1. **Remote Procedure Call(RPC)**
   - The RPC become widely accepted IPC (Inter process Communication) mechanism in distributed system.
   - RPC can be defined in the following manner: When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and the execution of the called procedure take place on B. Information can be transported from the caller to callee in the form of parameter and can come back in the as a procedure result. No message passing is visible to the programmer. This method is known as remote procedure call.
   - The RPC model is similar to the well-known and well defined procedure call model used for the transfer of control and data within a program in the following manner.
     a. For making a procedure call, the caller places an argument(Parameter) to the procedure in some specified location (Shared Memory).
     b. Controlled is then is transferred to the sequence of instruction that constitute the body of the procedure.
     c. The procedure body is executed in a newly created execution environment
     d. After the procedure execution is over, control return to the calling point possibly returning a result.

As shown in figure, when remote procedure call is made, the caller and the callee process interact in the following manner.



i) The caller commonly known as client process sends a call (request) message to the caller commonly known as server process and wait (block) for the reply message. The request message contains the remote procedure parameter.
ii) The server process executes the procedure execution in the reply message to the client process.
iii) Once the reply message is received, the result of the procedure execution is extracted and the caller's execution is resumed.

2. Remote Method Invocation(RMI)

- The main goal of RMI is to have a program running on one machine invoke a method belonging to an object whose execution is performed on another machine.
- It allows communication between two JVM running on different computers hence it is useful for communication between java objects.
- **RMI is object oriented RPC mechanism.**

Reasons for Using RMI

1. Some operations can be executed significantly faster on the remote system than the local client computer
2. Specialized data or operations are available on the remote system that cannot be replicated easily on the local system, for example, database access.
3. Simpler than programming over our own TCP sockets and protocols.

Architecture of RMI

- RMI is suited for distributed system as it allows the code that define the behavior and the code that implement the behavior to remain separate and to run on separate machine.
- So using RMI we can develop clients which are concerned about the definition of services and also develop server which are concerned with providing the services.
- So the key concept in RMI is that **interfaces** define services and **classes** defines implementations.
- Basically the RMI architecture consists of four layer as shown in figure below.

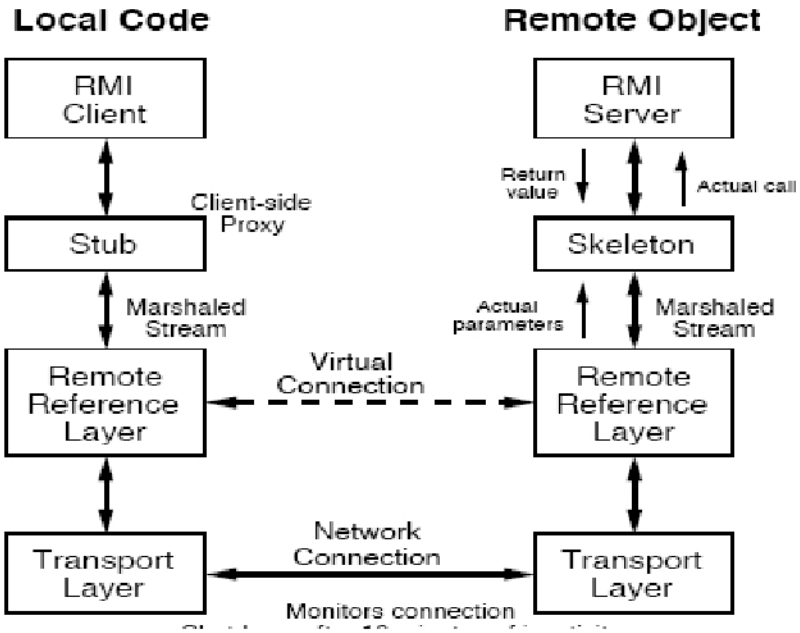


Fig: RMI layers

1. **The Application Layer**
   - In this layer, the client and server application are actually implemented.

2. **The proxy layer**
   - This layer consists of client stub and server skeleton objects.
   - RMI uses **stub and skeleton** objects to provide the connection between the client and the remote object.
   - A stub is a proxy for a remote object which is responsible for forwarding method invocation from the client to the server where actual remote object implementations reside. The client reference to a remote object is therefore a reference to a local stub.
   - A skeleton is a server side object which contains a method that dispatches calls to the actual remote object implementations. A remote object has associated local skeleton object to dispatch remote calls to it.

3. **The Remote Reference Layer (RRL)**
   - The remote reference layer provides packaging of a method call and its parameter and return values for the transport over the network.
   - The process of taking a collection of data items and assembling them into a suitable form for transmitting it as a message is called **marshaling.**
   - The process of encoding the parameters for converting the parameters into the form suitable to transport from on machine to another is called **parameter marshaling.**
   - Finally, the process of disassembling the message on arrival to produce equivalent collection of data items at the destination is called **Unmarshalling.**

4. **The Transport Layer**
   - The transport layer provides actual machine to machine communication using TCP/IP communication.
   - So the transport layer is responsible for setting up connection, managing existing connections, handling remote objects residing in its address and closing the connection.
   - The transport layer is accessed by RRL to send and receive and message to and from other machine.

RMI Mechanism

- The basic idea behind communication using RMI mechanism is the communication between stub and skeleton as other mechanism is hidden from the programmer.
- The action performed in client side (stub) and server side (skeleton) are explained below.
- The stub method on client side consists of
  i. An identifier of the remote object to be used.
  ii. A description of the method to be called.
  iii. The marshaled parameter

- The stub then sends this information to the server. The skeleton i.e. server side object performs the following action for each remote method call.
  i. It un-marshals the parameter.
  ii. It locates the object to be called.
  iii. It calls the desired method.
  iv. It acquires the return result and marshals the result or exception traces if exist.
  v. It sends a package consisting of the marshaled result back to the stub on the client.

RMI Registry Service

- The registration of the remote object must be done by the server in order for the client to **look it up i.e. for lookup service**, is called the RMI Registry. So, 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.

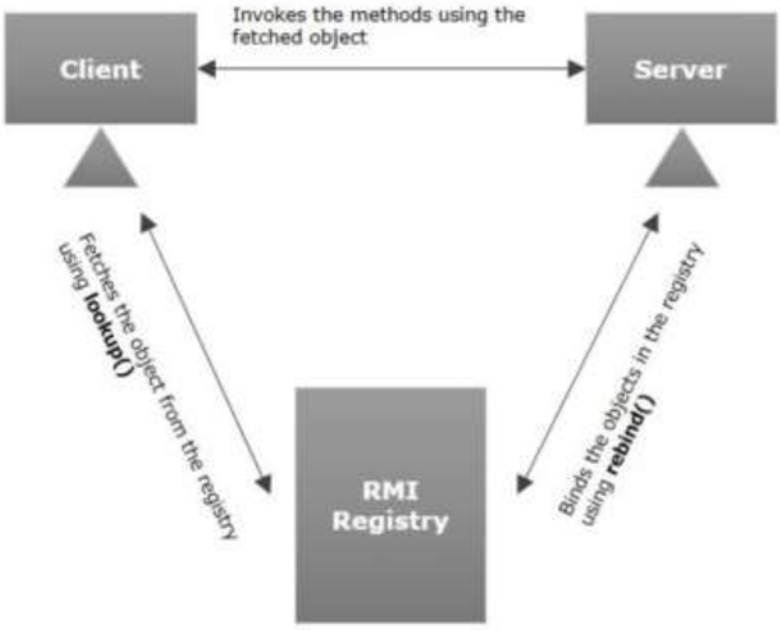


Fig: RMI registry Process

- 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 java.rmi.Naming** class has provided following methods for RMI registry management
  i. **public static void bind (String Name, Remote obj)** : This method is used to binds/register the specified name to a remote object(Stub of Object) in the rmiRegistry.
  ii. **public static void rebind (String Name, Remote obj)**: This method is used to rebind the specified name to a new remote object in the rmiRegistry.
  iii. **public static void unbind (String Name)**: This method is used to Destroys the binding for the specified name that is associated with a remote object in the rmiRegistry.
  iv. **public static void lookup (String Name)**: This method is used to Returns a reference i.e. a stub, for the remote object associated with the specified name.

Steps in creating RMI applications

For creating RMI application, the first step is to import java.rmi package so as to use the classes and interfaces provided by java.rmi package. Once imported, we can develop an application that uses RMI using the following steps.

1. Create and compile the remote interface that specifies the methods that will have remote access.

- Must be public
- Must extends the interface java.rmi.Remote
- Each remote method must have a throws java.rmi.RemoteException clause
- Interface must reside on both the server and the client side
- Any method parameters or return value of a reference type must implement Serializable interface.

(Serializable interface is that interface that allow translating object state into a format that can be stored or transmitted across network connection). The process of serializing an object is called marshalling of an object. The opposite operation, extraction of object state from a series of bytes is called deserialization or unmarshalling.)

2. Create server side applications i.e. write and compile a class that implements the remote interface in 1.

- Must extend java.rmi.server.UnicastRemoteObject for providing functionality that is needed to make object available from remote machine.
- Must implement the remote interface in 1
- Its constructors must be defined explicitly since they each may throw java.rmi.RemoteException
- Resides on the server side only

3. Write and compile a server class that (RMI Registry Step)

- Installs a security manager using System.setSecurityManager(new java.rmi.RMISecurityManager());
  This step may be optional.
- Instantiates an object of the implementation class in 2; this is the remote object
- Binds the remote object implObj to a unique identifier (a String) for the rmi registry
  java.rmi.Naming.rebind("uniqueID", implObj);

4. Write and compile a client class that

- Installs a security manager as in 4 (this step may be optional also).
- Requests an object from the remote server using its hostname and the unique identifier of the object, and then casts that object to the interface type from 1.
  InterfaceType it = (InterfaceType)java.rmi.Naming.lookup("rmi://r-lnx233.cs.uiowa.edu:1099/uniqueID");
  1099 is the default port and if used, its specification here can be omitted.
- If the client is running on the same machine as the remote object (the server), use
  InterfaceType it =(InterfaceType)java.rmi.Naming.lookup("rmi://localhost/uniqueID");
  or
  InterfaceType it =(InterfaceType)java.rmi.Naming.lookup("rmi:///uniqueID");
- Now methods on the remote object can be called using the interface class object it when the rmi system is up and running.

. Lookup() : This method is used to lookup remote stub in the rmiRegsitry.

5. **Create the Stub and Skeletons class using the RMI compiler, rmic tool.**

6. **Install files on the client and server machine.**
   The client machine should contain all the classes including client program, stub, and interface.
   The server machine should contain all the classes including server program, stub and skeleton .

7. **Start the bootstrap rmi registry in the background on the server side.**
   % start rmiregistry // assumes port 1099 // default port number for RMI registry is 1099
   % start rmiregistry 1492 & // specifies a port

7. Execute the server class on the machine that the client named, which was r-lnx233.
   % java ServerClass

Note: The rmi registry and the server must run on the same machine and in the same directory.

8. Execute the client class on another machine
   % java ClientClass

Creating a Client/Server Application using RMI

1. Write an RMI application to display 'Hello World'.
   i. First define remote interface. This consists of method that can be invoked by a client.

```
import java.rmi.*;
public interface HlowInterface extends Remote
{
```

```
        public String sayHellow() throws RemoteException;
}
```

Compile it using javac HlowInterface.java, it will create HlowInterface.class in the working directory.

ii. *Implement the interface in a server side application. A class that implement this remote interface can be used as a remote object. Client can then remotely invoke the sayHello( ) method.*

```
import java.rmi.*;
import java.rmi.server.*;
public class HlowImplementation extends UnicastRemoteObject implements HlowInterface
{
        public HlowImplementation() throws RemoteException
        {
        }
        public String sayHellow()
        {
                return "Hello World";
        }
}
```

Compile it using javac HelloImplementation.java, it will create HelloImplementation.class in the working directory.

iii. *Write and compile the server side class that contains the main program for server.*

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloServer
{
        public static void main(String args[])
        {
                //System.setSecurityManager(new RMISecurityManager());//install security manager
                try
                {
                        HlowImplementation hi=new HlowImplementation(); //remote object
                        Naming.rebind("Hlow101",hi); //Registering the remote object as "Hlow101"
                }
                catch(Exception e)
                {
                        System.out.println("Error occured in server "+e.getMessage());
                }
        }
}
```

Compile it using javac HelloServer.java, it will create HelloServer.class in the working directory.

iv. *Write and compile the client side class that contain the main program for client.*

```
import java.rmi.*;
public class HelloClient
{
        public static void main(String [] args)
        {
                //System.setSecurityManager(new RMISecurityManager());
                try
                {
                        String strURL="rmi://localhost/Hlow101"; // note localhost can be replaced
                                                                 with client IP address also
                        HlowInterface hi=(HlowInterface)Naming.lookup(strURL); //Getting reference
                                                                 to the remote object
```

```
                    System.out.println("Ther server says"+hi.sayHellow());//invoke the remote
                                                                                    object
                }
                catch(Exception e)
                {
                    System.out.println("Error in client machine"+e.getMessage());
                }
            }
        }
```

Compile it using javac HelloClient.java, it will create HelloClient.java in the working directory.

v. *Generate stubs and skeletons.*
The stub and skeleton is generated by by using the RMI compiler rmic as below

C:\> rmic HlowImplementation

This command generates two new file named HlowImplementation_Stub.class and HlowImplementation_Skel.class.
Note, HlowImplementation is a class that implements the methods declared in interface

vi. *Start the RMI Registry on server machine*
To start the rmiregistry, type the following in command prompt

C:\> start rmiregistry

Upon pressing the enter key, the RMI registery will be started.

vii. *Start the server program*

C:\> java HelloServer

viii. Start the client program.

C:\> java HelloClient

ix. *Finally, you will see the following output in client machine*

```
C:\Users\Hp-User\Desktop\Java programs\RMI\hello>java HelloClient
Ther server saysHello World
```

2. Write an RMI application to find the multiplication and additions of two numbers that are passed from the client machine.

Remote Interface:

```
import java.rmi.*;
public interface AddMulInterface extends Remote
{
    double add (double d1,double d2) throws RemoteException;
    double mul(double d1,double d2)throws RemoteException;
}
```

Remote Implementation in server side:

```
import java.rmi.*;
import java.rmi.server.*;
public class AddMulImpl extends UnicastRemoteObject implements AddMulInterface
{
    public AddMulImpl() throws RemoteException
    {
    }
    public double add(double a,double b) throws RemoteException
    {
```

```
		return (a+b);
	}
	public double mul(double a,double b) throws RemoteException
	{
		return (a*b);
	}
}
```

Server side application:

```
import java.rmi.*;
import java.rmi.server.*;
public class AddMulServer
{
	public static void main(String [] args)
	{
		try
		{
			AddMulImpl ami=new AddMulImpl();//remote object
			Naming.rebind("addmul",ami);
		}
		catch (Exception e)
		{
			System.out.println("Error occured in server"+e);
		}
	}
}
```

Client Side application:

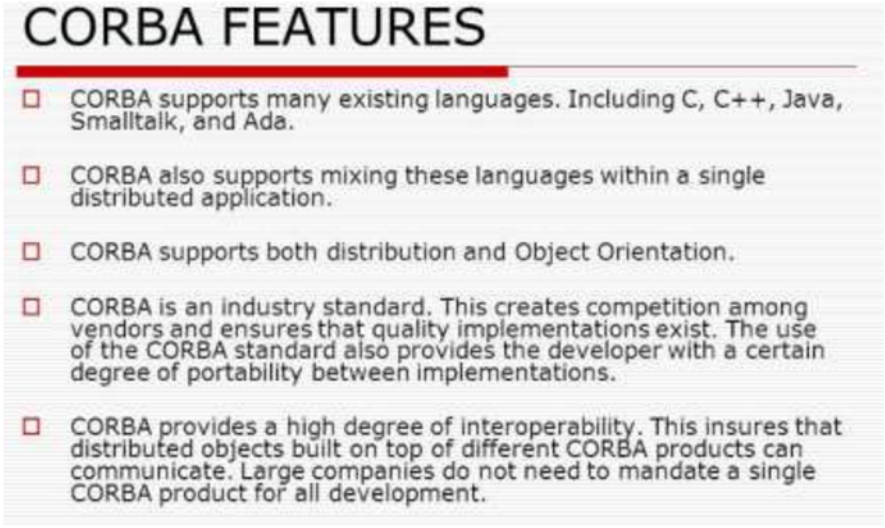
```
import java.rmi.*;
import java.util.Scanner;
public class AddMulClient
{
	public static void main(String [] args)
	{
		Scanner sc=new Scanner(System.in);
		System.out.println("Enter first number");
		double fn=sc.nextDouble();
		System.out.println("Enter the second number");
		double sn=sc.nextDouble();
		try
		{
			String strURL="rmi://localhost/addmul";
			AddMulInterface ami=(AddMulInterface)Naming.lookup(strURL);
			double addres=ami.add(fn,sn);
			double mulres=ami.mul(fn,sn);
			System.out.println("The sum of two number is"+addres);
			System.out.println("The product of two number is"+mulres);
		}
		catch(Exception e)
		{
			System.out.println("Error occured in client"+e);
		}
	}
}
```

3. CORBA

- CORBA allows communications between objects that are written in different programming language such as java, C++ etc.
- Object Request Broker (ORB) is considered as kinds of universal translator for inter object communication.
- So object do not talk directly to each other instead they use object broker located across the network to bargain between them.
- ORB follow the specification set up by the Object Management Group (OMG) for inter-ORB communication, this specification is called Internet Inter-ORB protocol or simply IIOP.

CORBA vs RMI

- RMI is a java-centric distributed object system but CORBA is designed to be language independent. So in CORBA object interfaces are specified in a language that is independent of the actual implementation language i.e. RMI is language dependent bur CORBA is language independent.
- RMI can be easier to master but CORBA is difficult to master.
- RMI follows server centric model while CORBA follows peer to peer ORB communication model.
- RMI is free but CORBA cost money according to the vendor.
- RMI objects are automatically garbage collected but CORBA object are not.
- RMI interfaces are defined in java but CORBA interfaces are defined in IDL (Interface definition language).



Assignment

Differentiate between RMI and CORBA.