

## Unit -5 RMI [5 Hrs.]

### 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**.

Following are the goals of RMI –

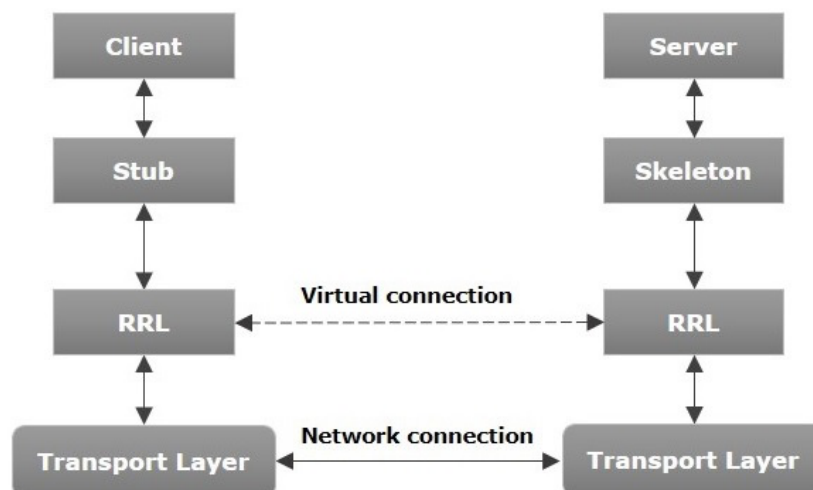
- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

### 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.



- **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.

## Understanding stub and skeleton

**RMI uses stub and skeleton object for communication with the remote object.**

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

### stub

**The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it.**

It resides at the client side and represents the remote object. **When the caller invokes method on the stub object, it does the following tasks:**

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

### skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

## 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**.

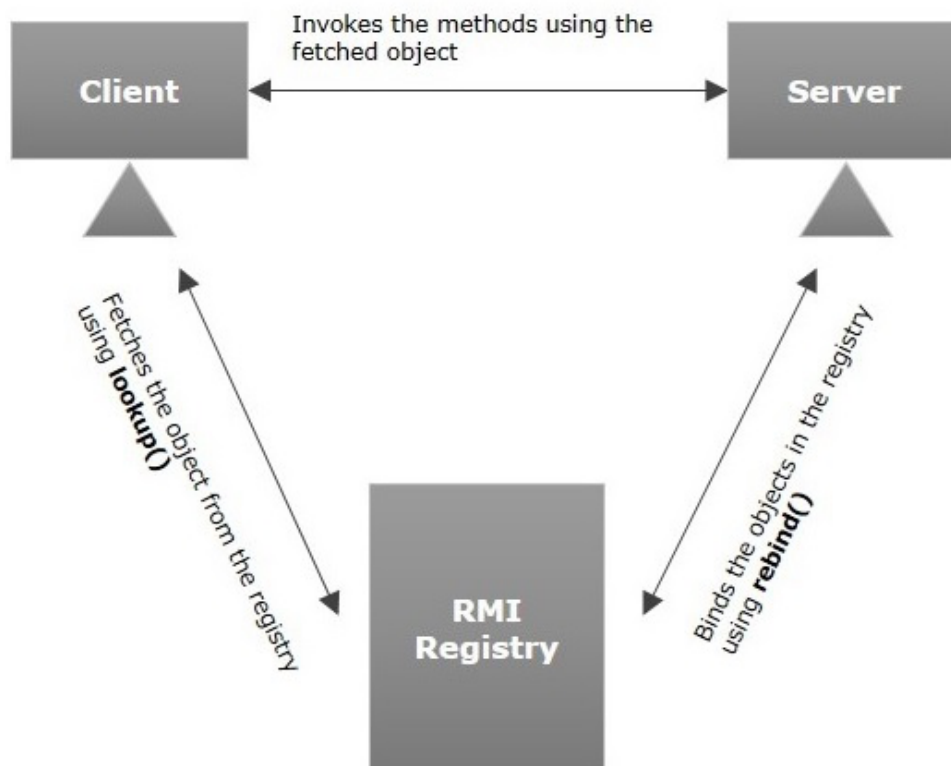
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

## 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 –



## Writing RMI Applications/RMI Programming Model

There are six steps to write the RMI applications:

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

### **1) Create the Remote Interface**

For creating the remote interface, extend the Remote interface and declare the **RemoteException** with all the methods of the remote interface. Here we are creating a remote interface **Message** that extends the **Remote interface**. There is only one method named **showMessage()** and it declares **RemoteException**.

```
import java.rmi.*;
public interface Message extends Remote{
    public void showMessage() throws RemoteException;
}
```

### **2) Provide the implementation of Remote Interface**

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to

- either extend the UnicastRemoteObject class,
- or use the exportObject() method of the UnicastRemoteObject class

Here, we are extending the UnicastRemoteObject class, so we must define a constructor that declares RemoteException.

```
import java.rmi.*;
import java.rmi.server.*;
public class MessageImpl extends UnicastRemoteObject
    implements Message{

    MessageImpl() throws RemoteException{
        super();
    }

    public void showMessage(){
        System.out.println("Hello World! from Server..");
    }
}
```

### **3) Create the Stub and Skeleton objects using the rmic Tool**

The rmic tool invokes the RMI compiler and crates stub and skeleton objects. Run following command in your terminal.

```
rmic Message
```

### **4) Start the Registry Service by the rmiregistry Tool**

Now start the registry service by using the rmiregistry tool. If you don't specify a port number, it uses a default port number. In this example, we are using the port number 5000.

To start registry service run following command in your terminal.

```
rmiregistry 5000
```

## **5) Create and Run the Server Application**

Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. In this example, we are binding the remote object by the name rmiserver.

```
import java.rmi.*;
import java.rmi.registry.*;

public class Server{
    public static void main(String[] args){
        try{
            Message obj=new MessageImpl();
            Naming.rebind("rmi://localhost:5000/rmiserver",obj);
            System.out.println("Server Started!");
        }catch(Exception ex){
            System.out.println(ex.toString());
        }
    }
}
```

## **6) Create and Run the Client Application**

At the client we are getting the object by the lookup() method of the Naming class and invoking the method on this object.

In this example, we are running the server and client applications in the same machine so we are using localhost. If we want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

```
import java.rmi.*;

public class Client{
    public static void main(String[] args){
        try{
            Message obj=(Message)Naming.lookup
                ("rmi://localhost:5000/rmiserver");
            obj.showMessage();
        }catch(Exception ex){
            System.out.println(ex.toString());
        }
    }
}
```

## Output:

```
My RMI — rmiregistry 5000 — 80x24
Last login: Mon Mar 28 20:20:35 on console
/Users/mymack/.zshrc:source:2: no such file or directory: /Users/mymack/.npm/npm
.sh
mymack@localhost My RMI % javac *.java
mymack@localhost My RMI % rmic MessageImpl
Warning: rmic has been deprecated and is subject to removal in a future
release. Generation and use of skeletons and static stubs for JRMP
is deprecated. Skeletons are unnecessary, and static stubs have
been superseded by dynamically generated stubs. Users are encouraged
to migrate away from using this tool to generate skeletons and static
stubs. See the documentation for java.rmi.server.UnicastRemoteObject.
mymack@localhost My RMI % rmiregistry 5000
```

```
My RMI — -zsh — 80x24
Last login: Mon Mar 28 20:35:46 on ttys001
/Users/mymack/.zshrc:source:2: no such file or directory: /Users/mymack/.npm/npm
.sh
mymack@localhost My RMI % java Client
mymack@localhost My RMI %

My RMI — java Server — 80x24
Last login: Mon Mar 28 20:34:01 on ttys000
/Users/mymack/.zshrc:source:2: no such file or directory: /Users/mymack/.npm/npm
.sh
mymack@localhost My RMI % java Server
Server Started!
Hello World! from Server..
```

## RMI program to find product of two numbers.

### Product.java

```
import java.rmi.*;
public interface Product extends Remote{
    public int multiply(int a,int b) throws RemoteException;
}
```

### ProductImpl.java

```
import java.rmi.*;
import java.rmi.server.*;

public class ProductImpl extends UnicastRemoteObject
    implements Product{

    ProductImpl() throws RemoteException{
        super();
    }

    public int multiply(int a,int b){
        int res=a*b;
        return res;
    }
}
```

### Server.java

```
import java.rmi.*;
import java.rmi.registry.*;

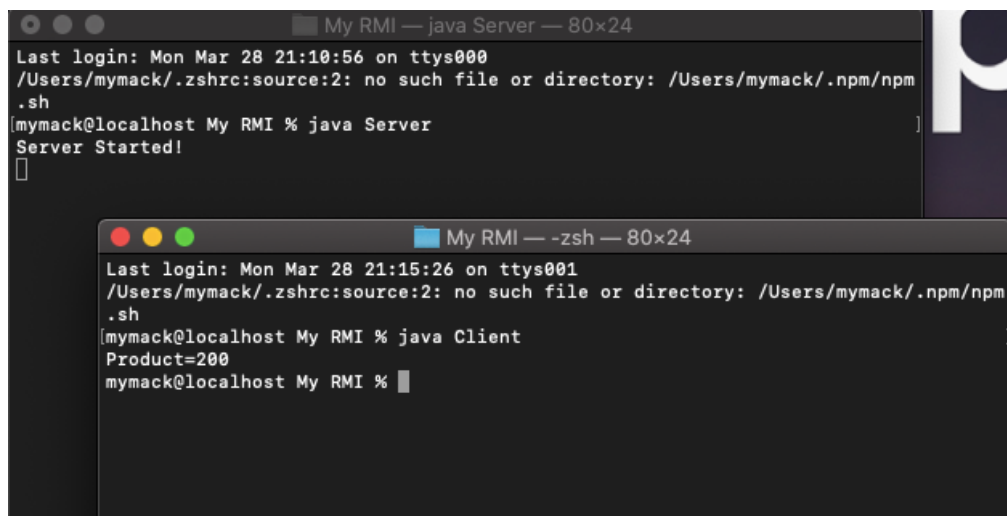
public class Server{
    public static void main(String[] args){
        try{
            Product obj=new ProductImpl();
            Naming.rebind("rmi://localhost:5000/rmiserver",obj);
            System.out.println("Server Started!");
        }catch(Exception ex){
            System.out.println(ex.toString());
        }
    }
}
```

### Client.java

```
import java.rmi.*;

public class Client{
    public static void main(String[] args){
        try{
            Product obj=(Product)Naming.lookup
                ("rmi://localhost:5000/rmiserver");
            int res=obj.multiply(10,20);
            System.out.println("Product="+res);
        }catch(Exception ex){
            System.out.println(ex.toString());
        }
    }
}
```

### Output:



The screenshot shows two terminal windows. The top window, titled 'My RMI — java Server — 80x24', shows the output of running 'java Server', which is 'Server Started!'. The bottom window, titled 'My RMI — -zsh — 80x24', shows the output of running 'java Client', which is 'Product=200'.

```
My RMI — java Server — 80x24
Last login: Mon Mar 28 21:10:56 on ttys000
/Users/mymack/.zshrc:source:2: no such file or directory: /Users/mymack/.npm/npm
.sh
[mymack@localhost My RMI % java Server
Server Started!
]

My RMI — -zsh — 80x24
Last login: Mon Mar 28 21:15:26 on ttys001
/Users/mymack/.zshrc:source:2: no such file or directory: /Users/mymack/.npm/npm
.sh
[mymack@localhost My RMI % java Client
Product=200
mymack@localhost My RMI % ]
```

## CORBA (Common Object Request Broker Architecture)

CORBA is the standard developed by the Object Management Group to provide interoperability among distributed objects. It is the world's leading middleware solution.

It enables the exchange of information, independent hardware platforms, programming languages and operating systems. It is often defined as “software bus” because it is a software based communication interface through which the objects are located and accessed.

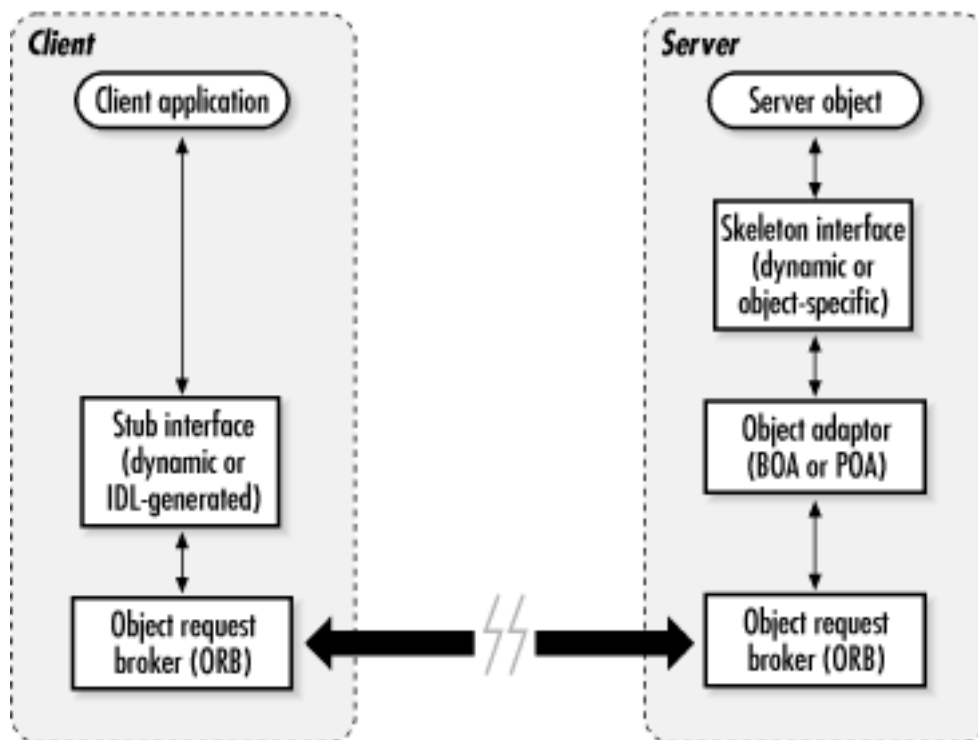


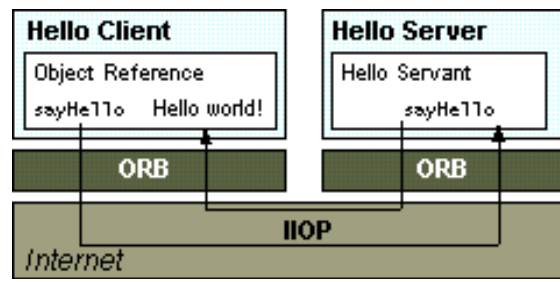
Fig: CORBA Architecture

The major components that make up the CORBA architecture include the:

- **Interface Definition Language (IDL)**, which is how CORBA interfaces are defined,
- **Object Request Broker (ORB)**, which is responsible for all interactions between remote objects and the applications that use them,
- The **Portable Object Adapter (POA)**, which is responsible for object activation/deactivation, mapping object ids to actual object implementations.
- **Naming Service**, a standard service in CORBA that lets remote clients find remote objects on the networks, and
- **Inter-ORB Protocol (IIOP)**.



This figure shows how a one-method distributed object is shared between a CORBA client and server to implement the classic "Hello World" application.



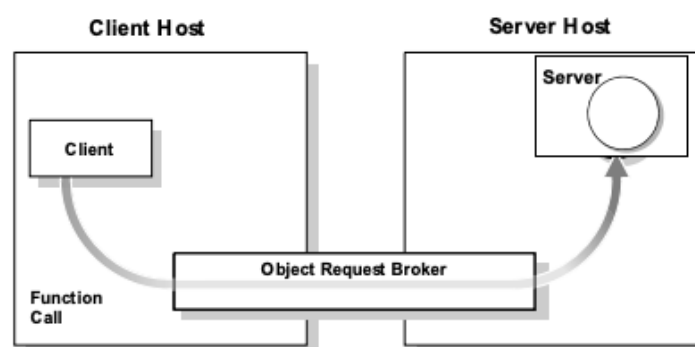
**A one-method distributed object shared between a CORBA client and server.**

### Interface Definition Language (IDL)

- The services that an object provides are given by its interface. Interfaces are defined in OMG's Interface Definition Language (IDL). IDL is independent of any programming language.
- Mappings from IDL to specific programming languages are defined as part of the CORBA specification. Mappings for C, C++, Smalltalk, Ada, COBOL, and Java have been approved by OMG.
- The syntax of both Java and IDL were modeled to some extent on C++, so there are a lot of similarities between the two in terms of syntax. However, there are differences between IDL and Java.

### Object Request Broker (ORB)

- The core of the CORBA architecture is the ORB. Each machine involved in a CORBA application must have an ORB running in order for processes on that machine to interact with CORBA objects running in remote processes.
- Object clients and servers make requests through their ORBs and the remote ORB locates the appropriate object and passes back the object reference to the requestor.
- The ORB provides the communication infrastructure needed to identify and locate objects, handles connection management, etc. The ORBs communicate with each other via the IIOP.



## Naming Service

Defines how CORBA objects can be looked up by a name. **It is a *Common Object Service (COS)* and allows an object to be published using a symbolic name and allows clients to obtain references to the object using a standard API.**

The CORBA naming service provides a naming structure for remote objects.

## IIOP

The CORBA standard includes specifications for inter-ORB communication protocols that transmit object requests between various ORBs running on the network.

The Inter-ORB Protocol (IIOP) is an inter-ORB protocol based on TCP/IP and so is extensively used on the Internet.

## POA

**The POA connects the server object implementation to the ORB.** It extends the functionality of the ORB ~~and some its services include~~: activation and deactivation of the object implementations, generation and management of object references, mapping of object references to their implementations, and dispatching of client requests to server objects through a skeleton.

## Comparing RMI with CORBA :

| SN | RMI   | CORBA   |
|----|---|---|
| 1  | RMI is a Java-specific technology.  | CORBA has implementation for many languages.  |
| 2  | It uses Java interface for implementation.                                    | It uses <b>Interface Definition Language (IDL)</b> to separate interface from implementation.   |
| 3  | RMI objects are garbage collected automatically.                              | CORBA objects are not garbage collected because it is language independent and some languages like C++ does not support garbage collection. |
| 4  | RMI programs can download new classes from remote JVM's.                      | CORBA does not support this code sharing mechanism.   |
| 5  | RMI passes objects by remote reference or by value.                           | CORBA passes objects by reference.  |
| 6  | Java RMI is a server-centric model.   | CORBA is a peer-to-peer system.   |
| 7  | RMI uses the Java Remote Method Protocol as its underlying remoting protocol. | CORBA use Internet Inter- ORB Protocol as its underlying remoting protocol.   |
| 8  | The responsibility of locating an object implementation falls on JVM.         | The responsibility of locating an object implementation falls on Object Adapter either Basic Object Adapter or Portable Object Adapter.     |