Chapter 8: Remote Method Invocation (RMI)

Overview of 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. Similar to the RPC mechanism found on other systems. In RMI, methods of remote objects can be invoked from other JVMs. It is provided in the package java.rmi.

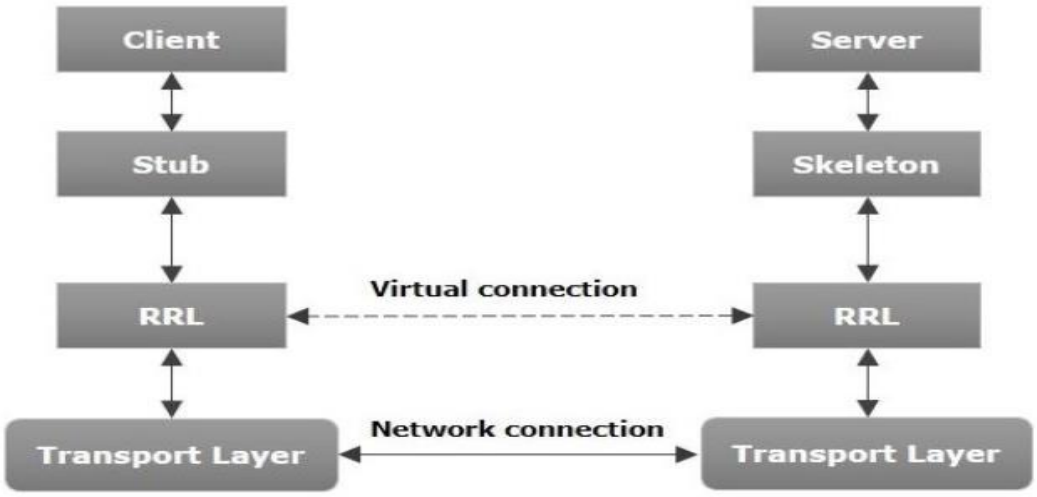
Architecture of an RMI Application

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.



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**.

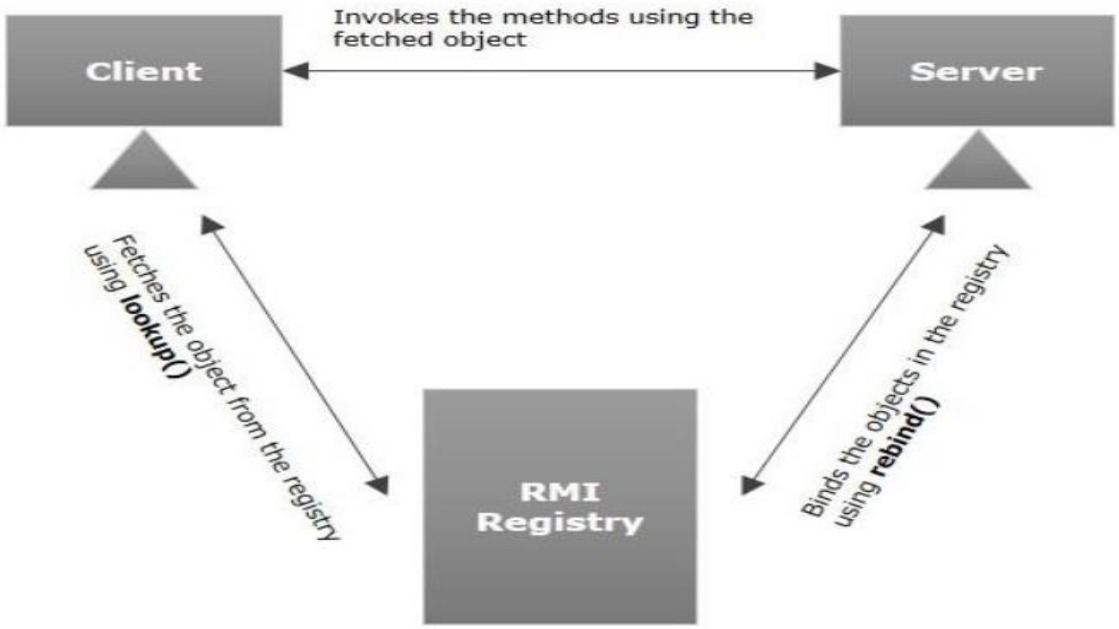
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

RMI Registry

RMIregistry 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:



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

JAVA RMI ─ RMI APPLICATION

To write an RMI Java application, you would have to follow the steps given below:

- Define the remote interface
- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program
- Compile the application
- Execute the application

Definingthe Remote Interface

A remote interface provides the description of all the methods of a particular remote object. The client communicates with this remote interface.

To create a remote interface –

- Create an interface that extends the predefined interface **Remote** which belongs to the package.

- Declare all the business methods that can be invoked by the client in this interface.
- Since there is a chance of network issues during remote calls, an exception named **RemoteException** may occur; throw it.

Following is an example of a remote interface. Here we have defined an interface with the name **Hello** and it has a method called **printMsg()**.

```
import java.rmi.Remote;
import java.rmi.RemoteException;
// Creating Remote interface for our application
public interface Hello extends Remote {
void printMsg() throws RemoteException;
}
```

Developingthe Implementation Class (RemoteObject)

We need to implement the remote interface created in the earlier step. (We can write an implementation class separately or we can directly make the server program implement this interface.)

To develop an implementation class –

- Implement the interface created in the previous step.
- Provide implementation to all the abstract methods of the remote interface.

Following is an implementation class. Here, we have created a class named **ImplExample** and implemented the interface **Hello** created in the previous step and provided **body** for this method which prints a message.

```
// Implementing the remote interface
public class ImplExample implements Hello
{
// Implementing the interface method
public void printMsg() {
System.out.println("This is an example RMI program");
}
}
```

Developingthe Server Program

An RMI server program should implement the remote interface or extend the implementation class. Here, we should create a remote object and bind it to the **RMIregistry**.

To develop a server program –

- **Create a class** that extends the implementation class implemented in the previous step. (or implement the

remote interface)

- **Create a remote object** by instantiating the implementation class as shown below.
- Export the remote object using the method **exportObject()** of the class named **UnicastRemoteObject** which belongs to the package **java.rmi.server**.
- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry**.
- Bind the remote object created to the registry using the **bind()** method of the class named **Registry**. To this method, pass a string representing the bind name and the object exported, as parameters.

Following is an example of an RMI server program.

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class Server extends ImplExample{
public Server() {}
public static void main(String args[]) {
try {
// Instantiating the implementation class
ImplExample obj = new ImplExample();
// Exporting the object of implementation class
// (here we are exporting the remote object to the stub)
Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
// Binding the remote object (stub) in the registry
Registry registry = LocateRegistry.getRegistry();
registry.bind("Hello", stub);
System.err.println("Server ready");
} catch (Exception e) {
System.err.println("Server exception: " + e.toString());
e.printStackTrace();
}
}}
```

DevelopingtheClientProgram

Write a client program in it, fetch the remote object and invoke the required method using this object.

To develop a client program –

- Create a client class from where you want invoke the remote object.
- Get the RMI registry using the **getRegistry()** method of the **LocateRegistry** class which belongs to the package **java.rmi.registry.**
- Fetch the object from the registry using the method **lookup()** of the class Registry which belongs to the package **java.rmi.registry.** To this method you need to pass a string value representing the bind name as a parameter. This will return you the remote object down cast it.
- The lookup() returns an object of type **remote**, down cast it to the type **Hello**.
- Finally invoke the required method using the obtained remote object.

Following is an example of an RMI client program.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
private Client() {}
public static void main(String[] args) {
try {
// Getting the registry
Registry registry = LocateRegistry.getRegistry(null);
// Looking up the registry for the remote object
Hello stub = (Hello) registry.lookup("Hello");

// Calling the remote method using the obtained object
stub.printMsg();
// System.out.println("Remote method invoked");
} catch (Exception e) {
System.err.println("Client exception: " + e.toString());
e.printStackTrace();
}
}
}
```

Compiling the Application

To compile the application –

- Compile the Remote interface.
- Compile the implementation class.

- Compile the server program.
- Compile the client program.