www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 2 Issue 11 November, 2013 Page No. 3136-3140

Remote Method Invocation – Usage & Implementation

[1]*Nivedita Joshi*,[2]*Pooja Singh*

[1]Student, B.Tech (IT), Dronacharya College of Engineering, Maharishi Dayanand University
Gurgaon, Haryana, India
Nivedita.joshi92@yahoo.co.in

[2]Student, B.Tech (IT), Dronacharya College of Engineering, Maharishi Dayanand University
Gurgaon, Haryana, India
Poojasingh.8dec@gmail.com

Abstract:***Java Remote Method Invocation (RMI) allows programmer to execute remote methods using the same semantics as local functions calls. RMI is Java's version is Remote Procedure Call (RPC). RMI internal implementation is out of client scope and only deals with exposed interface of remote server object. The aim of RMI is to allow the programmers to invoke remote services from remote objects. The paper explains the RMI architectural layers and its mechanism. The paper deals with the working of all the layers of RMI and how they are implemented. This paper has taken into account an example to explain the proper working of RMI.***

Keywords: Object Serialization, Marshaling,Stub & Skeleton, Naming, RMI Registry.

1. Introduction

RMI is a way by which a programmer can create an object-oriented program where the objects on different computers, usually client and server, can interact over a distributed network.

The Java RMI is Java's native scheme for creating and using remote objects over network. It let us distribute java object instances across network on different machines, which can be invoked from local machine.

To interact with the methods of objects on remote machines using JVM (Java Virtual Machine), RMI is used. This process allows the exchange of data/statistics using multiple JVMs. It provides the location transparency by making the methods being accessed locally.

RMI is the Java version of Remote Procedure Call (RPC), but has the ability to pass more than one remote objects along with the request. This object being passed has the ability to change the service that is performed on the remote computer. This property of java is called "Moving Behavior" by Sun Microsystems.

For example, when a user at a remote computer fills out an expense account, the Java program interacting with the user could communicate, using RMI, with a Java program in another computer that always had the latest policy about expense reporting. In reply, that program would send back an object and associated

method information that would enable the remote computer program to screen the user's expense account data in a way that was consistent with the latest policy. The user and the company both would save time by catching mistakes early. Whenever the company policy changed, it would require a change to a program in only one computer.

Object parameter-passing mechanism is known as object serialization. An RMI request is a request to invoke the method of a remote object. The request is of the same syntax as a request to invoke an object method in the same computer. In general, RMI is designed to preserve the object model and its advantages across a network.

2. Remote Method Invocation

The Remote Method Invocation (RMI) model represents an Evaluation distributed object application. It allows an object inside a JVM, acting as a client, to invoke a methodon an object running on a remote JVM, actins as a server, and return the results to the client.Therefore, RMI implies a client and a server.

RMI uses a layered architecture; each of the layers could be enhanced or replaced without affecting the rest of the model.For example, a UDP/IP layer could replace the transport layer without affecting the upper layers.

RMI architecture explains the communication between two Java Virtual Machines, where the methods are invoked from local machine.

The RMI implementation consists of basically three abstraction layers. The first is the Stub and Skeleton layer, which lies beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

Remote Reference Layer comprehends how to infer and manage references made from clients to the remote service objects.

In JDK 1.1, this layer provides a unicast connection from clients to remote service objects that are running and export them onto a server. The transport layer is based on TCP/IP connections between machines in a network. Java RMI provides the following elements:

1. Remote object implementations.
2. Client interfaces, or stubs, to remote objects.
3. A Registry for remote object for finding objects over the network.
4. A network protocol for communication between remote objects and their client, which is Java Remote Method Protocol.

2.1 Layers of RMI

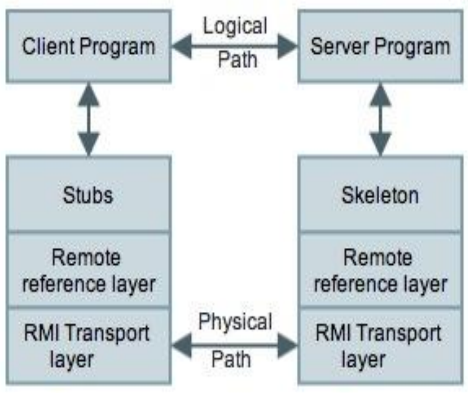


Figure 1: RMI Architecture

- The Stub Layer (& Skeletons):

A stub is a program on the client side of the client/serverrelationship.The stub layer resides between application layer and the rest of the RMI system and acts as an interface.

Stub is the transparent proxy object that makes the interface on client side to communicate with the server side. It is generated by a JDK took, rmic, from the server object compiled code, and is distributed to the client.

The network-related code dwells in the stub and skeleton, so that the client and server will not have to deal with the networkand sockets in their code.A skeleton is a remote object at the server-side. This stub consists of methods that invoke dispatch calls to the remote implementation of objects.

- The Remote Reference Layer:

The second layer of RMI Architecture deals with the interpretation of references made from client remote objects to server remote objects. This layer deals the lower level transport interfaces.

With the help of Remote Object Activation it activates the latent remote object for unicast communication.

- The Transport Layer:

The third layer of RMI architecture provides the connection between two JVMs. The transport layer sets up the connections to remote address spaces, manages them, monitors the connection liveliness, and listens the incoming calls.

For incoming calls, the transport layer establishes a connection. It locates the target, dispatches the remote calls and passes the connection to the dispatcher.

2.2 RMI Mechanism

i. The Client Program uses the stub for making a request for a remote object. The server program receives this request from the skeleton

ii. RMI invocation is initiated by calling a method on stub object, which maintains an internal reference to the remote object it represents.

iii. The stub forwards the method invocation request through Remote Reference Layer with the help of marshaling process. This layer forwards the request to appropriate remote object.

iv. Marshaling: this process transforms the local objects to a suitable portable form, so they can be easily broadcasted to a remote process. Each array, string or user-defined object is checked while being marshaled to conclude whether it implements java.rmi.Remote interface. If it is a remote object, then that reference is used for marshaling.

v. If it is a Serializable object, then first it is serialized into bytes that are sent to remote object and then they are reassembled to form a copy of local object. If the object is neither then it throws a java.rmi.MarshalException to client.

vi. The remote reference layer then receives the marshaled arguments from the stub, which then converts the client request into single network-level requests.

vii. The remote reference layer on server side receives transport-level request and transforms it into a request for skeleton to match referenced object.

viii. The skeleton converts the remote request into suitable method call and carries out the process of un-marshaling the method arguments appropriate for server. The arguments sent as remote objects are converted into local stubs and those sent as serialized objects are converted into local copies of originals.

ix. If a return value is generated then the object is marshaled by skeleton and sent back to the client through server remote reference layer.

x. The end result is transmitted back to client through a suitable transport protocol.

2.3 Steps to create RMI-based clients and server

- Creation of RMI-based Server:

Create the remote interface and the servant component class. To host these servant classes create the RMI Server. Compile the class files and generate the Skeletons and IDL File. Start up the Server

- Creation of RMI-based Client:

Create the Client class. Enable the Stub Generation and compile the Client. Make sure the Server is running and then startup the Client process.

3. RMI Registry

The server application creates an object and makes it accessible remotely. For making the object remote, the server has to register the RMI-enabled objects that are available to the clients.

The clients can find these Remote Services on a *naming service*, which is obtainable on publicly defined port. RMI defines its own naming service, the RMI Registry, having a standard port of 1099. A standard JDK tool, rmiregistry, handles the registry.

If an object implements the java.rmi.Remote interface, then it is bounded to registry context. The interface that is being referenced is implemented by each registry context.

3.1 Methods for registering Remote Object

The methods of remote objects are invoked by implementing the java.rmi.Remote interface. Following are the methods for registering the remote objects:

i. bind(): It binds the specified name to the remote object. The parameter of this method should be in an URL format.

ii. unbind(): Destroys the binding for a specific name of a remote method in the registry.

iii. rebind(): It again binds the specified name to the remote object. The current binding will be replaced by rebinding.

iv. list(): It returns the names that were bound to the registry in an array form. These names are in the form of URL-formatted string.

v. lookup(): A stub, a reference will be returned for the remote object which is related with a specified name.

4. Implementing an RMI System

The steps involved in building a distributed application with RMI include:

- Interface definitions for the remote methods
- Implementations of the remote services
- Stub files .
- A server to host the remote services
- An RMI Naming service
- A client program that needs the remote services

4.1 Define an interface for declaring remote methods

It involves implementing a remote interface for between the client and the server. It defines the remote objects that are requested by client.

We are creating a simple application to add two numbers. So we declare the add() method in interface Addition.java.

```
import  java.rmi.Remote;
import  java.rmi.RemoteException;

public interface Addition extends Remote{

  public long add(long a, long b) throws
RemoteException;

}
```

The interface is extended so that it can be called remotely in between the client and server. The RemoteException occurs when there is some failure in RMI process.

4.2 Define the class and implement remote methods

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class AdditionImpl extends UnicastRemoteObject
implements Addition
{protected AdditionImpl() throws RemoteException
   {super();}

public long add(long a,long b)throws RemoteException
   {return a+b; }}
```

We define a class AdditionImpl.java and implement the interface and define the body of the remote method.

The UnicastRemoteObject is a base class for user-defined remote objects having the general form as, Public class UnicastRemoteObject extends RemoteServer

4.3 Defining the Server Program

The first Parameter is a URL to a registry that contains the name of the application, which here is “AdditionService”. The second parameter is an object name that is accessed remotely between client and server. The rebind is a method of Naming class which is implemented in java.rmi.* package.

1099 is the default RMI port and 127.0.0.1 is a localhost-ip address

```
import java.rmi.Naming;
public class AdditionServer
{AdditionServer()
   {
     try{

        Addition c= new AdditionImpl();

Naming.rebind("rmi://127.0.0.1:1099/AdditionService",c);

       }

       catch(Exception e){

          e.printStackTrace();
       }
   }
   public static void main(String[] args)
   {
       new AdditionServer();
   }


}
```

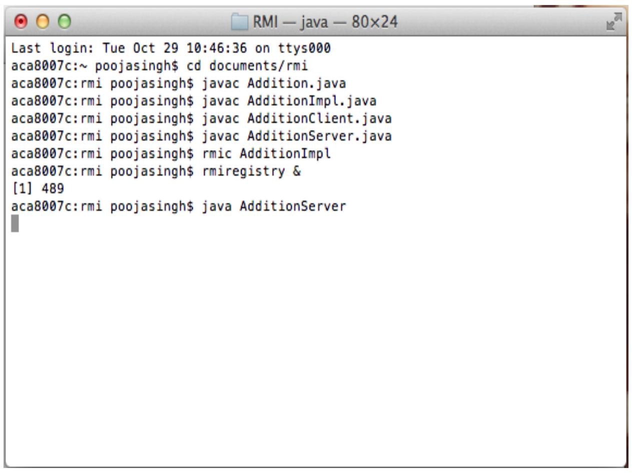
4.4 Defining the client program

To access the remote object on client side, which is already binding at a server side by, reference URL, we use the lookup method, which has the same reference URL.

This lookup is a method of Naming class which is available in java.rmi.* package. The name of the URL must be same as specified on server side class.

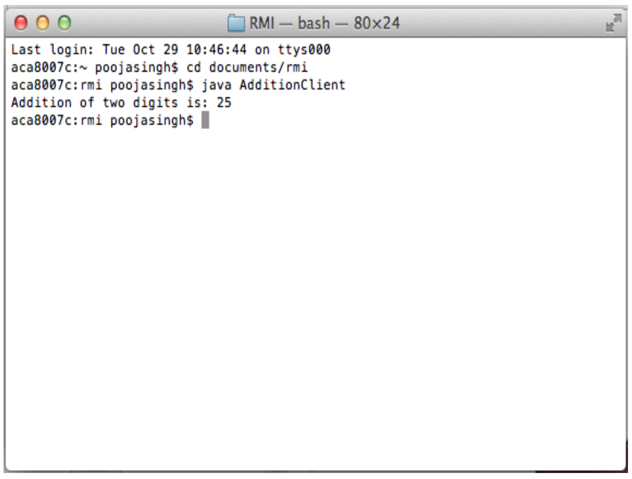
```
import java.rmi.Naming;

public class AdditionClient
{
public static void main(String[] args)
{
   try
   {
Addition c= (Addition)Naming.lookup
("//127.0.0.1:1099/AdditionService");
System.out.println("Addition of two digits is:
"+c.add(10,15));
   }
 catch(Exception e){
       System.out.println(e);
}}}
```

- Compile all the source java files.
  - javac Addition.java
  - javac AdditionImpl.java
  - javac AdditionClient.java
  - javac AdditionServer.java
- The command rmic enables the stub generation.
  Syntax: rmic AdditionImpl
  This command produces AdditionImpl_Stub.class file.
- Start the RMI remote Registry: The references of the remote objects are registered to RMI Registry.
  Syntax: rmiregistry & (which opens rmiregistry.exe)
- Run the server program and then run the client program on another terminal window.



```
Last login: Tue Oct 29 10:46:36 on ttys000
aca8007c:~ poojasingh$ cd documents/rmi
aca8007c:rmi poojasingh$ javac Addition.java
aca8007c:rmi poojasingh$ javac AdditionImpl.java
aca8007c:rmi poojasingh$ javac AdditionClient.java
aca8007c:rmi poojasingh$ javac AdditionServer.java
aca8007c:rmi poojasingh$ rmic AdditionImpl
aca8007c:rmi poojasingh$ rmiregistry &
[1] 489
aca8007c:rmi poojasingh$ java AdditionServer
```

Figure 2: Running Server Program



```
Last login: Tue Oct 29 10:46:44 on ttys000
aca8007c:~ poojasingh$ cd documents/rmi
aca8007c:rmi poojasingh$ java AdditionClient
Addition of two digits is: 25
aca8007c:rmi poojasingh$ 
```

Figure 3: Running Client Program

5. Conclusion

The paper describes how Remote Method Invocation (RMI) represents distributed object application. The paper explains how RMI implies a client and server and implements remote connections between them.

The servers' job is to accept request from a client, perform services, and then send the results back to the client.

The use of Registry and Naming classes is to bootstrap our distributed applications.

RMI implementation involves four software programs namely:

- Client program: does the request
- Server program: implements the request
- Stub Interface: implemented by client to know the remote functions
- Skeleton Interface: implemented by server

Advantages of RMI:

- It's easy and clean to implement and produces more robust and flexible applications.
- Distributed systems are created while decoupling the client and server objects.
- No client installation is required except java environment.
- While changing database, only server objects are recompiled but not the interface and client program

References

[1] Java Remote Method Invocation:
http://en.wikipedia.org/wiki/Java_remote_method_invocation

[2] Java RMI Tutorial:
http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html#serial

[3] The JavaTM Tutorials-RMI:
http://docs.oracle.com/javase/tutorial/rmi/

[4] Ninghui Li, John C. Mitchell and Derrick Tong, "Securing Java RMI-based Distribute Applications"

[5] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal and Aske Plaat, "An Efficient Implementation of Java's Remote Method Invocation" (1999)

[6] Remote Method Invocation:
http://www.javacamp.org/moreclasses/rmi/rmi4.html

[7] Remote Method Invocation:
http://www.javatpoint.com/RMI

[8] Naming Methods:
http://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/api/java.rmi.Naming.html