International Journal of Emerging Engineering Research and Technology
Volume 3, Issue 5, May 2015, PP 68-76
ISSN 2349-4395 (Print) & ISSN 2349-4409 (Online)

Distributed Object Computing Using Java Remote Method Invocation

Hanumant Pawar, Sujeet Patil, Sourabh Karche, Udit Upadhayay, Mahesh Channaram

[1]*Department of Computer Engineering, MIT College of Engineering, Pune, India (Asst. Professor)*
[2]*Department of Computer Engineering, MIT College of Engineering, Pune, India (B.E. Student)*
[3]*Department of Computer Engineering, MIT College of Engineering, Pune, India (B.E. Student)*
[3]*Department of Computer Engineering, MIT College of Engineering, Pune, India (B.E. Student)*
[3]*Department of Computer Engineering, MIT College of Engineering, Pune, India (B.E. Student)*

ABSTRACT

We are using Remote Method Invocation (RMI) for Communication System that retains as much of semantics of Java RMI and only includes differences where they make sense. We have designed such a model and implemented a system that supports remote method invocation (RMI) for distributed objects in Java. A distributed system is defined as a system in which different computers communicate and coordinate their actions by passing messages. Distributed computing is a field of computer science that studies distributed systems. Initially to communicate between computers sockets were used. In socket communication, data can be sniffed if it is not encrypted. Also complexity is not handled properly. Later Remote Procedure Call (RPC) was evolved. Remote Procedure Call (RPC) is an inter-process communication (IPC) mechanism that enables data exchange and invocation of functionality residing in a different process. In RPC limitation of socket communication was overcome but it was too complex to implement. To provide simplicity distributed objects (CORBA) and RMI were developed. Java RMI provides same functionality as RPC and Distributed objects but it is simple for beginners to implement. To achieve the goal of distributed computing we have used Java RMI.

Keywords: Distributed computing, remote procedure call, Remote Method Invocation, Java

INTRODUCTION

Distributed objects are an extension to the Remote Procedure Call (RPC). The term distributed objects are designed to work together, but can reside either in multiple computers connected to each other or in different processes inside the same computer. One object sends a message to another object in a remote machine or process to perform some task.

Different distributed object models-

1) CORBA [1] (Common Object Request Broker Architecture) defined by the Object Management Group (OMG) is developed for communication of systems that are deployed on diverse platforms. CORBA is used for collaboration between systems on different operating systems, programming languages, and computing hardware. CORBA has many of the same design goals as object-oriented programming such as encapsulation and reuse. CORBA uses an object-oriented model although the systems that utilize CORBA do not have to be object-oriented.

2) DCOM (Distributed Common Object Model) is a set of Microsoft concepts and program interfaces in which client program objects can request services from server program objects on other computers in a network. DCOM is based on the Component Object Model (COM), which provides a set of interfaces allowing clients and servers to communicate within the same computer (that is running Windows 95 or a later version).

3) JAVA RMI (Remote Method Invocation) is a mechanism that allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine

Address for correspondence:
upadhayay.udit@gmail.com

or a different one. The RMI mechanism is basically an object-oriented RPC mechanism. Using the Java programming language and development environment, objects on different computers can interact in a distributed network. RMI is the Java version of Remote Procedure Call (RPC), but can pass one or more objects along with the request. The object includes information that will change the service that is performed in the remote computer.

In this paper we will briefly describe the Java RMI. We will also describe the RMI architecture and relevant RMI interfaces. Finally, we discuss related work and conclusions.

ARCHITECTURE

The RMI system consists of three layers [2]:

- The stub/skeleton layer - client-side stubs and server-side skeletons
- The remote reference layer - remote reference behavior (such as invocation to a single object or to a replicated object)
- The transport layer - connection set up and management and remote object tracking

The application layer sits on top of the RMI system. The relationship between the layers is shown in the following figure.

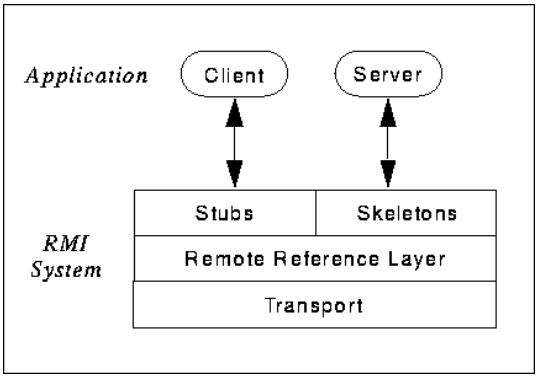

Figure1. *Architecture of RMI*

A remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server.

1) Stub/Skeleton layer - The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. This layer does not deal with specifics of any transport, but transmits data to the remote reference layer via the abstraction of *marshal streams*. Marshal streams employ a mechanism called *object serialization* which enables Java objects to be transmitted between address spaces. Objects transmitted using the object serialization system are passed by copy to the remote address space, unless they are remote objects, in which case they are passed by reference.

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:

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

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:

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

2) The Remote Reference Layer- The remote reference layer deals with the lower-level transport interface. This layer is also responsible for carrying out a specific remote reference protocol which is independent of the client stubs and server skeletons.

Each remote object implementation chooses its own remote reference subclass that operates on its behalf. Various invocation protocols can be carried out at this layer. The remote reference layer has two cooperating components: the client-side and the server-side components. The client-side component contains information specific to the remote server (or servers, if the remote reference is to a replicated object) and communicates via the transport to the server-side component. During each method invocation, the client and server-side components perform the specific remote reference semantics. For example, if a remote object is part of a replicated object, the client-side component can forward the invocation to each replica rather than just a single remote object.

In a corresponding manner, the server-side component implements the specific remote reference semantics prior to delivering a remote method invocation to the skeleton. This component, for example, would handle ensuring atomic multicast delivery by communicating with other servers in a replica group. The remote reference layer transmits data to the transport layer via the abstraction of a stream-oriented connection. The transport takes care of the implementation details of connections. Although connections present a streams-based interface, a connectionless transport can be implemented beneath the abstraction.

3) The Transport Layer

In general, the transport layer of the RMI system is responsible for:

- Setting up connections to remote address spaces.
- Managing connections.
- Monitoring connection.
- Listening for incoming calls.
- Maintaining a table of remote objects that reside in the address space.
- Setting up a connection for an incoming call.
- Locating the dispatcher for the target of the remote call and passing the connection to this dispatcher.

The concrete representation of a remote object reference consists of an endpoint and an object identifier. This representation is called a *live reference*. Given a live reference for a remote object, a transport can use the endpoint to set up a connection to the address space in which the remote object resides. On the server side, the transport uses the object identifier to look up the target of the remote call.

The transport for the RMI system consists of four basic abstractions:

- An endpoint is the abstraction used to denote an address space or Java virtual machine.
- A channel is the abstraction for a conduit between two address spaces.

- A connection is the abstraction for transferring data (performing input/output).
- The transport abstraction manages channels.

The design and implementation also supports multiple transports per address space, so both TCP and UDP can be supported in the same virtual machine.

DEVELOPING RMI CLIENT AND SERVER

For developing RMI client and server [3] following points need to be known

Serializable Class

A class is Serializable if it implements the java.io. Serializable interface. Subclasses of a Serializable class are also serializable. Many of the standard classes are Serializable, so a subclass of one of these is automatically also Serializable. Normally, any data within a Serializable class should also be Serializable. Although there are ways to include non-serializable objects within a serializable object, it is awkward to do so.

Using a serializable object in a remote method invocation is straightforward. One simply passes the object using a parameter or as the return value. The type of the parameter or return value is the Serializable class. Both the Client and Server programs must have access to the definition of any Serializable class that is being used. If the Client and Server programs are on different machines, then class definitions of Serializable classes may have to be downloaded from one machine to the other. Such a download could violate system security.

Remote Exception

Mostly while programming java RMI RemoteException class is used.A RemoteException is the common superclass for a number of communication-related exceptions that is used to handle the execution of a remote method call. Each method of a remote interface, an interface that extends java.rmi.Remote, must list RemoteException in its throws clause.

Remote Classes and Exception

A Remote class has two parts: the interface and the class itself. The Remote interface must have the following properties:

- The interface must be public.
- The interface must extend the interface java.rmi.Remote.

Every method in the interface must declare that it throws java.rmi.RemoteException. Other exceptions may also be thrown.

The Remote class itself has the following properties:

- It must implement a Remote interface.
- It should extend the java.rmi.server.UnicastRemoteObject class. Objects of such a class exist in the address space of the server and can be invoked remotely. While there are other ways to define a Remote class, this is the simplest way to ensure that objects of a class can be used as remote objects.
- It can have methods that are not in its Remote interface. These can only be invoked locally

Unlike the case of a Serializable class, it is not necessary for both the Client and the Server to have access to the definition of the Remote class. The Server requires the definition of both the Remote class and the Remote interface, but the Client only uses the Remote interface. Roughly speaking, the

Remote interface represents the type of an object handle, while the Remote class represents the type of an object. If a remote object is being used remotely, its type must be declared to be the type of the Remote interface, not the type of the Remote class.

RMI Registry

The Object Registry is a name server that relates objects with names. Objects are registered with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

An object of a remote class can be referenced in two different ways:

- Within the address space where the object was constructed, the object is an ordinary object which can be used like any other object.
- Within other address spaces, the object can be referenced using an object handle. While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object.

The java.rmi.registry.Registry remote interface provides methods for lookup, binding, rebinding, unbinding, and listing the contents of a registry. The java.rmi.Naming class uses the registry remote interface to provide URL-based naming.

The REGISTRY_PORT is the default port (1099) of the registry.

The lookup method returns the remote object bound to the specified name. The remote object implements a set of remote interfaces. Clients can cast the remote object to the expected remote interface. (This cast can fail in the usual ways that casts can fail in the Java language.)

The bind method associates the name with the remote object. If the name is already bound to an object the AlreadyBoundExcepton is thrown.

The rebind method associates the name with the remote object. Any previous binding of the name is discarded.

The unbind method removes the binding between the name and the remote object. If the name is not already bound to an object the NotBoundException is thrown.

The list method returns an array of Strings containing a snapshot of the names bound in the registry. The return value contains a snapshot of the contents of the registry.

Programming a Client

In this section programming of client has been discussed. The Client itself is just a Java program. It need not be part of a Remote or Serializable class, although it will use Remote and Serializable classes.

A remote method invocation can return a remote object as its return value, but one must have a remote object in order to perform a remote method invocation. So to obtain a remote object one must already have one. Accordingly, there must be a separate mechanism for obtaining the first remote object. The Object Registry fulfills this requirement. It allows one to obtain a remote object using only the name of the remote object.

The name of a remote object includes the following information:

The Internet name (or address) of the machine that is running the Object Registry with which the remote object is being registered. If the Object Registry is running on the same machine as the one

that is making the request, then the name of the machine can be omitted.

The port to which the Object Registry is listening. If the Object Registry is listening to the default port, 1099, then this does not have to be included in the name.

Here is the example of Client program:

```
/**
 * Client program for the "Hello, world!" example.
 * @param argv The command line arguments which are ignored.
 */
public static void main (String[] argv) {
  try {
    HelloInterface hello =
      (HelloInterface) Naming.lookup ("//ortles.ccs.neu.edu/Hello");
    System.out.println (hello.say());
  } catch (Exception e) {
    System.out.println ("HelloClient exception: " + e);
  }
}
```

The Naming.lookup method obtains an object handle from the Object Registry and listening to the default port. The result of Naming.lookup must be cast to the type of the Remote interface.

The remote method invocation in the example Client is hello.say(). It returns a String which is then printed. A remote method invocation can return a String object because String is a Serializable class. The code for the Client can be placed in any convenient class.

Programming a Server

In this section programming of server has been discussed. The Server itself is just a Java program. It need not be a Remote or Serializable class, although it will use them. The Server does have some responsibilities:

1. If class definitions for Serializable classes need to be downloaded from another machine, then the security policy of your program must be modified. Java provides a security manager class called RMISecurityManager for this purpose. The RMISecurityManager defines a security policy that allows the downloading of Serializable classes from another machine. The "Hello, World!" example does not need such downloads, since the only Serializable class it uses is String. As a result it isn't necessary to modify the security policy for the example program. If your program defines Serializable classes that need to be downloaded to another machine, then insert the statement System.setSecurityManager (new RMISecurityManager()); as the first statement in the main program below.
2. At least one remote object must be registered with the Object Registry. The statement for this is: Naming.rebind (objectName, object); where object is the remote object being registered, and objectName is the String that names the remote object.

Here is the example Server:

```
/**
 * Server program for the "Hello, world!" example.
```

```
 * @param argv The command line arguments which are ignored.
 */
public static void main (String[] argv) {
  try {
    Naming.rebind ("Hello", new Hello ("Hello, world!"));
    System.out.println ("Hello Server is ready.");
  } catch (Exception e) {
    System.out.println ("Hello Server failed: " + e);
  }
}
```

The rmiregistry Object Registry only accepts requests to bind and unbind objects running on the same machine, so it is never necessary to specify the name of the machine when one is registering an object. The code for the Server can be placed in any convenient class.

EXECUTION STEPS

- First compile all java files.
- Generate the stubs and skeleton using rmic tool.
- Start the rmiregistry using 'start rmiregistry' for windows OS and 'rmiregistry' in Linux OS.
- Start the Remote Server Objects.
- Now Run the Client program.

RMI PROGRAM EXAMPLE

A typical RMI based distributed application have four java files[4] which are a remote interface, A server implementation class which implements the remote interface, a server class which creates the stub for the server implementation class and bind objects to a port on server machine and a client program which invokes the remote method. We will demonstrate this with an example in which the remote interface provide a function for addition of two numbers.

Create the RMI Interface

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

5.2. Create the Client Program

```
import java.rmi.*;
public class CalcClient
{
        public static void main(String s[])throws Exception
```

```
        {
                String url="rmi://" + s[0]+"/Calc-Server";
                calc c=(calc)Naming.lookup(url);
                int no1,no2,ch;
                int res=0;
                no1=Integer.parseInt(s[1]);
                no2=Integer.parseInt(s[2]);
                ch=Integer.parseInt(s[3]);
                switch(ch)
                {
                case 1:res=c.add(no1,no2); break;
                }
                System.out.println("Result >> " + res);
        }
}
```

Create Server Implementation Class

```
import java.rmi.*;
import java.rmi.server.*;
public class CalcImp extends UnicastRemoteObject implements calc
{
        public CalcImp()throws RemoteException
    {}
        public int add(int a,int b)throws RemoteException
        {return a+b;}
}
```

Server Class for Creating and Binding Implementation Class Stub

```
import java.rmi.*;
public class CalcServer
{
        public static void main(String s[])
        {
                try
                {
                CalcImp c=new CalcImp();
```

```
            Naming.rebind("Calc-Server",c);
            }
            catch(Exception e){e.printStackTrace();}
    }}
```

ADVANTAGES AND DISADVANTAGES

Following are the advantages of Java RMI [5]

- RMI is easy to develop and use.
- In RMI, dynamic class loading is very powerful.
- RMI uses the Java built-in safety mechanism to ensure the safety of users download the program execution system. RMI uses specially designed to protect the system from malicious applet against safety management procedures.
- RMI is object-oriented.

Disadvantages of Java RMI are as follows [5]

- Java RMI only supports java.
- IT could be insecure, when using Dynamic class loading.
- Overhead of marshaling and unmarshaling.
- Overhead of object serialization.

CONCLUSION

Distributed computing finds various application in computer networks, peer to peer network, wireless sensor network, web based application etc. A major challenge in distributed computing is interoperability. Various platforms like CORBA, DCOM and java RMI aim to solve this challenge by providing a middleware layer. In this paper we have explained the concept of distributed computing using java remote method invocation. Various aspects of java remote method invocation from implementation point of view are discussed in details. The Main challenge in developing any distributed application is deployment of the application on different platforms. Although the java RMI Simplifies the process of development and deployment as compared to other platforms but the main disadvantage of java RMI is that it only supports java programming language. Java RMI is a part of oracle 'Whole world java' Campaign but the world is not ready for it yet and java RMI faces interoperability issues. Even so, Java RMI is an excellent tool for the beginners, who are learning the concept of distributed computing.

REFERENCES

[1] Java, RMI and CORBA A white paper prepared **by** David Curtis, Object Management Group

[2] http://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/guide/rmi/spec/rmiTOC.doc.html

[3] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html

[4] http://docs.oracle.com/javase/tutorial/rmi/index.html

[5] http://www.stackoverflow.com/questions/2339853/benefits-and-disadvantages-of-using-java-rmi