IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Innovative Technique of Message Passing In Loosely Coupled System

Monika Arya*
Department of Computer Science, Azad Institute of Engineering & Technology, Lucknow
monika_a20@rediffmail.com

Abstract

The basic idea is that whenever we think about distributed system then we say that it is collection of loosely coupled processors where inter processor communication takes place by message passing process. Now what happen is that whenever we try to pass message in any user defined distributed system then the user is forced to use system's specific objects for message passing. What we mean to say is user cannot use his or her own object to be passes as message. Now here we try to solve this problem using "Innovative Techniques of message passing in loosely coupled system" technique. That is we are now capable of passing user defined objects in user designed distributed system.

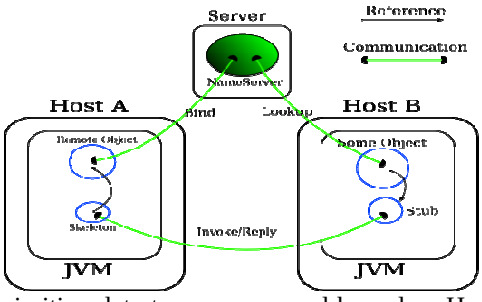
Keyword:

I. Introduction

A system of processes in which the interactions are solely through messages is often called loosely-coupled. Such systems are attractive from a programming viewpoint. They are designed by decomposing a specification into its separable concerns, each of which could then be implemented by a process; the operation of the system can be understood by asserting properties of the message sequences transmitted among the component processes. A key attribute of loosely-coupled systems is a guarantee that a message that has been sent cannot be unsent. As a consequence, a process can commence its computation upon receiving a message, with the guarantee that no future message it receives will require it to undo its previous computations [1].

Basically in this we try to solve the problems of Remote Method Invocation (RMI), i.e. in general RMI user cannot pass user defined parameters or objects in remote methods [1]. Here we try to pass the user defines objects and other entities as parameters of RMI methods.

II. Existing System



When invoking remote methods, primitive data types are passed by value. Hence, any changes to the data on the remote host are not reflected at the original host. Second, to pass an object to a remote method by value, the object must implement the java lang. Serializable interface. As before, changes to the object's copy will not propagate to the local object. Finally, if we want to pass an object over the network by remote reference, the object must be an exported remote object (extend the Unicast Remote Object class), and must implement a remote interface (which extends javarmi.Remote). A stub for the remote object will be serialized and passed to the remote host. The remote host can then use that stub to invoke methods of our remote object. There is only one copy of the data at any time, which means that all hosts are updating the same data [3].

III. Remote Method Invocation (Rmi)

Remote Method Invocation (RMI) allows a Java object executes on one machine to invoke a method of a java object that executes on another machine. This is an important feature, because it allows building distributed application [2].

IV.A Simple Client/Server Application Using Rmi

This section provides step-by-step directions for building a simple client/server application by using RMI. The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum [2].

Step One: Enter and Compile the Source Code

This application uses four source files. The first file, AddSERVERIntf.java, defines the remote interface that is provided by the server. It contains one method that accepts two **double** arguments and returns their sum. All remote interfaces must extend the Remote interface, which is part of java.rmi. Remote defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a RemoteException [2].

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

The second file, AddServerImpl.java, implements the remote interface. The implementation of the add( ) method is straightforward. All remote objects must extend UnicastremoteObject, which provides functionality that is needed to make objects available from remote machines.

```
import java.rmi.*;
import java.rmi.server.*;
public class AssServerImpl extends UnicastRemoteObject
   implements AddServerIntf  {
   public AddServerImpl( ) throws RemoteException  {
    }
  Public double add(double d1, double d2) throws RemoteException {
  }
}
```

The third source file, AddServer.java, contains the main program for the server machine. Its primary function is to update the RMI registry on that machine. This is done by using the rebind( ) method of the Naming class(found in java.rmi). That method associates a name with an object reference. The first argument to the rebind( ) method is a string that names the server as “AddServer”. Its second argument is a reference to an instance of AddServerImpl.

```
import java.net.*;
import jave .rmi.*;
public class AddServer  {
  public static void main(String args[ ])   {
try  {
 AddServerImpl addServerImpl = new AddServerImpl( );
 Naming. rebind( “AddServer”,addServerImpl);
}
catch(Exception e)  {
  System.out.println(“Exception: “ +e);
} } }
```

The fourth source file, AddClient.java, implements the client side of this distributed application. AddClient.java requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the rmi protocol. The string includes the IP address or name of the server and the string "AddServer". The program then invokes the lookup( ) method of the Naming class. This method accepts one argument, the rmi URL, and returns a reference to an object of type AddServerIntf. All remote methor invocations can then be directed to this object.
The program continues by displaying its arguments and then invokes the remote add( ) method. The sum is returned from this method and is then printed.

```
import java.rmi.*;
public class AddClient {
  public static void main (String args [ ]) {
   try  {
    string addServerURL= "rmi://"+args[0] + "/AddServer";
    AddServerIntf addServerIntf= (AddServerIntf) Naming.lookup(addServerURL);
    System.out.println("The first number is :" +args[1]);
    double d1=Double.valueof(args[1]).doubleValue( );
    System.out.println("The Second number is : "+args[2]);
    double d2= Double.valueof(args[2]).doubleValue( );
    System.out.println("The sum is:" + addServerIntf.add(d1,d2));
}
catch (Exception e)  {
   System.out.println( "Exception: "+e);
 }}}
```

After you enter all the code, use javac to compile the four source that you created.

Step Two: Generate a Stub

Before, you can use the client and server; you must generate the necessary stub. In the context of RMI, a **stub** is a java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.
A remote method may accept arguments that are simple types or objects. In the latter case, the object may have references to other objects. All of this information must be sent to the remote machines. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine.
If a response must be returned to the client, the process works in reverse. Note that the serialization and deserialization facities are also used if objects are returned to a client.
To generate a stub, using a tool called the RMI complier, which is invoked from the command line, as shown here:

```
rmic  AddServerImpl
```

This command generates the file AddServerImpl_Stub.class. When using rmic, be sure that CLASSPATH is set to include the current directory [2].

Step Three: Install Files on the Client and Server Machines

Copy Addclient.class, AddServerImpl_Stub.class, and AddServerIntf.class to a directory on the client machine. Copy AddServerIntf.class, AddServerImpl.class, AddServerImpl_Stub.class, and AddServer.class to a directory on the server machine.

Step Four: Start the RMI Registry on the Server Machine

A program called rmiregistry, which executes on the server machine. It maps names to object references. First, check that the CLASSPAth environment variable includes the directory in which files are located. Then start the RMI registry from the command line, as shown here:Start rmiregistry [2]

Step Five: Start the Server

The server code is started from the command line, as shown here:
Java AddServer
Recall that the AddServer code instantiates AddServerImpl and Registers that object with the name "AddServer".

Step Six: Start the Client

The AddClient software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together.
java AddClient server 1 8 9
java AddClient 11.12.13.14 8 9
In the first line, the name of the server is provided. The second line uses its IP address (11.12.13.14).
This example can try without having a remote server. To do so, start rmiregistry, start AddServer, and then execute AddClient using command line.
Java AddClient 127.0.0.1 8 9
Here, the address 127.0.0.1 is the "loop back" address for the local machine. Using this address allows to exercise the entire RMI mechanism without actually having to install the server on a remote computer.
In either case, sample output from this program is shown here:
The first number is: 8
The Second number is: 9
The sum is : 17.0

V. Problem Description

In Remote methods one cannot pass user define objects as parameters. If user wants to pass the user define objects, the server gives the error.

In computer science, loose coupling (or loosely coupled) is a type of coupling that describes how multiple computer systems, even those using incompatible technologies, can be joined together for transactions, regardless of hardware, software and other functional components. Loosely coupled systems describe those that work on an exchange relationship where little input is needed from each of the additional systems. In a loosely coupled system hardware and software may interact but they are not dependant on each other to work. Computers in a network are considered loose-coupled systems as a client machine may request data from the server, but the two systems also work independently of each other.

In software terminology, loosely coupled refers to software where routines, modules, functions, and similar components are executed only as needed, and do not run at the launch of the software application and while it is being used. Web services are a type of software application that uses loose coupling.

VI. Proposed System

From above discussions we can easily see that one cannot pass user define objects as parameters in remote methods.
In our system we try to fabricate a system using which one could easily pass user defined objects as parameters through RMI.

RMI Sytem	Proposed System
• int add(int a, int b) ---protoype correct for java RMI • int add(parameter par) user defined object(UDO) This cannot be done in Java RMI • System A System B UDO – Transfer – Not Possible	• Helps in doing the second prototyping in which used defined object par is passed in remote method as parameter. • User defined object (UDO) transfer is not possible in RMI but possible in our proposed system. • System A System B UDO — Transfer Possible

VII. References

[1]. Loosely-Coupled Processes Jayadev MisraDepartment of Computer Sciences The University of Texas at Austin Austin, Texas 78712 (512) 471-9547 misra@cs.utexas.edu

[2]. "The complete Reference", Herbert Schildt.

http://www.cs.bgu.ac.il/~spl121/RMI