International Journal of Engineering & Technology, 7 (2.26) (2018) 44-47

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Classes involved in implementing remote method invocation (RMI) feature in java

V Sudarsan [1] *, R Sugumar [2]

[1] *Bharathiar University, Coimbatore, Tamil Nadu*
[2] *Velammal Institute of Technology, Chennai, Tamil Nadu*
**Corresponding author E-mail: vsudarsaan@yahoo.com*

Abstract

This work is aimed at discussing about the Java classes that are involved in implementing the Remote Method Invocation feature in Java. Remote Method Invocation feature is used in applications involving distributed processing. Applications based on distributed processing enable load sharing.

Keywords: *Java Classes Involved in Implementing RMI; Remote Method Invocation; RMI in Java*

1. Introduction

Remote Method Invocation, in future referred to as RMI in short, allows invocation of methods of an object belonging to another system on the network. RMI is client server technique which enables a client to invoke a remote method on the server. The object belonging to the server is referred as Skeleton and the objects belonging to the client is referred as Stub.

In this paper we attempt to discuss in detail about the Java classes that are involved in implementing the Remote Method Invocation feature to provide a basic knowledge about the distributed processing capabilities of RMI. The classes in Java can be classified under two categories.

1) User defined Classes
2) Built-in Classes

User defined classes are created by the programmers writing programs in a language. User defined classes can call methods belonging to other built-in classes or user defined classes through their associated objects.

Built-in classes are those classes that are already available in Java code. To access the methods belonging to built-in classes through their associated objects, the classes in which the methods are defined must be imported through import statements in which further information about them are found.

2. Related work

Some past works based on Remote Method Invocation feature are analyzed.

1) Guillermo L. Taboada, Sabela Ramos, Roberto R. Exposito, Juan Tourino and Ramon Doallo analyzed the use of Java language in high performance computing applications involving both shared and distributed memory programming. The authors opined that the performance gap between Java and native languages such as C and Fortran has narrowed down due to the Just in Time (JIT) Compiler of the Java Virtual Machine (JVM). The authors analyzed the different options in Java for high performance computing such as
   1) Shared Memory Programming
   2) Java Sockets
   3) Remote Method Invocation
   4) Message passing interface.

   The authors concluded their work by stating that Java can achieve almost similar performance to natively compiled languages, both for sequential and parallel applications.
2) Ann Wollrath, Roger Riggs, and Jim Waldo proposed a distributed model for the Java system. This model was designed using the Remote Method Invocation feature of Java. The authors compared their approach with different techniques of distributed processing in Java such as
   - Socket Programming
   - Using Remote Procedural Call (RPC)
   - Remote Method Invocation

   The authors explained how the RMI technique augurs well for invoking objects on remote machines. The authors also compared it with other technique such as CORBA for handling Java objects and opined that CORBA falls short of seamless integration due to their inter operability requirement with other languages.
3) Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal and Aske Plaat built a new compiler based Java system that was designed from scratch to support efficient Remote Method Invocation on parallel systems. The authors claimed that the performance measurements of their approach in RMI implementation is substantially faster than that of Sun JDK and JIT. The authors claimed that the gain in efficiency is attributed to three factors: the use of compile time type information to generate specialized serializers; a more streamlined and efficient RMI protocol; and the usage of faster communication protocols.
4) George K. Thiruvathukal, Lovely S. Thomas and Andy T. KorczynskiMaassen proposed a Reflective Remote Method Invocation (RRMI) model in their work. The authors were of the opinion that despite providing features desirable for high performance distributed computing, the design and im-

plementation of RMI are deficient in key areas of importance. The authors attempted to address the deficiencies of RMI through their Reflective RMI model. RRMI is a class library, which supports core features of RMI but is not tied to the Java Virtual Machine in any way. It uses a descriptor-based scheme for defining and invoking remote methods. It supports synchronous and asynchronous forms of Remote Method Invocation borrowing ideas from the Message Passing Interface. The authors claimed that RRMI model has been tested under Windows, Solaris, and works with the Applet Viewer.

5) Ninghui Li, John C. Mitchell and Derrick Tong presented a security architecture for securing distributed applications built using proxy based architectures such as RMI. The authors felt that the use of proxies introduces security vulnerabilities that should be addressed before relying on systems using technologies such as Java RMI. The authors claimed that the security architecture proposed in their work enables to counter the threats of the Java RMI by providing mutual authentication between client and service and thereby enabling efficient client access control.

3. User defined classes

The Remote Method Invocation feature must provide the following user defined classes.

3.1. Provide an interface for the server

The interface for the server declares a set of server methods that are invoked from a remote client.

3.2. Provide an interface for the client

The interface for the client declares a set of client methods that are invoked by the server.
Server and Client interfaces are remote interfaces which are invoked from a remote Java Virtual Machine. The Server and Client interfaces should extend the built-in class java.rmi.Remote. In addition, the java.rmi.RemoteException must be included in the methods declared in Server and Client interfaces in their throws clause.

3.3. Provide an application for the server

The server application class should implement the server interface. This class should also extend the built-in class UnicastRemoteObject. This class can also contain the code to create and load the RMI registry with the details of the Port Number. In such case, the Remote Exception must be thrown.

3.4. Provide an application for the client

The client application class should implement the client interface. This class should also extend the built-in class UnicastRemoteObject.

4. Built-in classes

The Built-in classes are used within the user defined classes. The necessary built-in classes that must be used are discussed.

4.1. Remote interface

The user defined interfaces given above (server interface and client interface) must extend the Java.rmi.Remote interface. It helps to identify all the remote objects. We can gain access to a method remotely only when they are specified in a remote interface. The server and client interfaces in the given example are classic samples of remote interfaces. It can be noticed from the given example that they extend the Remote interface.

4.2. Remote exception class

The java.rmi.RemoteException class contains the exceptions thrown during runtime. This exception is thrown when the invocation of a remote method is unsuccessful.

4.3. Registry interface

Java.rmi.registry.Registry is an interface, which performs functions such as lookup, binding, rebinding and listing the contents of the registry.

4.4. Locate registry class

java.rmi.registry.LocateRegistry class is used to create a remote object registry or to fetch a reference to a remote object registry. Methods supported by this class are createRegistry() and getRegistry ().

4.5. Naming class

Java. rmi.Naming class consists of methods such as rebind() and lookup(). Bind () and rebind () methods are used for binding a name with the remote object and once bound, a remote object host can use the method lookup () to find the remote objects using their name.

4.6. Unicast remote object class

All remote objects to facilitate availability of objects to remote machines must extend java.rmi.server.UnicastRemoteObject class.

5. Sample implementation

5.1. Server interface

```
Import java.rmi.Remote;
Public interface ServerInterface extends Remote public void register (ClientInterface CO, String str
Throws java.rmi.RemoteException;
```

5.2. Client interface

```
Public interface ClientInterface extends
java.rmi.Remote
Public void display () throws
 java.rmi.RemoteException;
```

5.3. Server application

```
Import java.net.InetAddress;
Import java.rmi.registry.LocateRegistry;
Import java.rmi.registry.Registry;
Import java.rmi.RemoteException;
Import java.rmi.Naming;
Import java.rmi.server.UnicastRemoteObject;
Public class ServerAppl extends
UnicastRemoteObject
Implements ServerInterface {
String portNum, registryURL;
Public ServerAppl () throws RemoteException
{Super () ;}
Public void startRegistry () throws
RemoteException {Try
{Registry r=
LocateRegistry.getRegistry (3232);
Relist ( );
```

```
Catch (RemoteException e)
{Registry r =
LocateRegistry.createRegistry (3232);
}

Public synchronized void register (
ClientInterface CO, String str)
Throws java.rmi.RemoteException
{Try { System.out.println ("Client "+ str
 + "calling server method\n");
CO. display ();
}
Catch (Exception e) {}}
Public static void main (java.lang.String args []) {
Try {String rURL;
ServerAppl s=new ServerAppl();
s.startRegistry();
InetAddress ownIP=
InetAddress.getLocalHost();
String strLine=ownIP.getHostAddress();
rURL = "rmi://"+strLine+":" + 3232 +
"/callback";
Naming.rebind(rURL, s);
System.out.println ("Server Ready");
}
Catch (Exception E) {}
}}
```

5.4. Client application

```
Import java.net.InetAddress;
Import java.rmi.*;
Import java.rmi.server.*;
Public class ClientAppl extends
UnicastRemoteObject implements ClientInterface
{Public ClientAppl () throws RemoteException
{Super (); }
Public void display ()
Throws java.rmi.RemoteException
{Try
{InetAddress ownIP=
InetAddress.getLocalHost();
String strLine=ownIP.getHostAddress();
System.out.println ("Server Calling
Client "+strLine);
}
Catch (Exception e) {}
}
Public static void main (String args[])
{String str="";
Try
{int RMIPort=3232;
String serverip="192.168.3.1";
String registryURL =
"rmi://"+serverip+":" + 3232 + "/callback";
ServerInterface h =
(ServerInterface)Naming.lookup(registryURL);
ClientInterface callbackObj = new ClientAppl ();
h.register ();
System.exit (0);
}
Catch (Exception e)
{System.out.println (e);
System. Exit (0);
}
}
}
```

6. Results and discussion

A sample implementation for the four user defined classes are given above. The built-in classes are referred inside the user defined classes to gain access to the necessary code. To test the code, the source files given above are to be created using their class names with .java extension. RMI uses a client server approach. Hence, to execute the application, we need to execute the server and client programs in different machines on the network. Follow the steps given below for executing the application.

- Create the given four-java class files with their given names.
- Create a folder (eg. RMIS) in server system and copy all the source files (*.java) into that folder.
- Move to that folder (eg. RMIS) in command prompt and compile the source files using the command javac *.java. On compilation, the executable code for all the source files are created in the name of source file but with a .class extension.
- The Java software must be installed in the system to compile & execute the code. Now give the command java ServerAppl to invoke the server application.

After the Server Ready message is displayed, go to client system and create a folder (eg. RMIC).
Copy all the executable files (.class) from the server system folder (eg. RMIS) to the client system folder (eg. RMIC). Then execute the client application using the command java Cliental. Make a note of the following considerations to successfully execute the application.

- The client application can also be executed on a different folder in the same system if two systems are not available.
- The client application can be executed on multiple systems on the network
- To execute client application on others systems on the network, the JRE (Java Runtime Environment) software must be installed on all the client systems on the network.
- To execute the server application, the Java Software must be installed in server system to facilitate compilation of software.

Some screenshots are provided for a better clarity in understanding. The application is executed on three systems with ip addresses 192.168.3.1 to 192.168.3.3. The ip address of the server is 192.168.3.1. The following screenshot belongs to the client 192.168.3.3. A similar output will be shown on other clients.

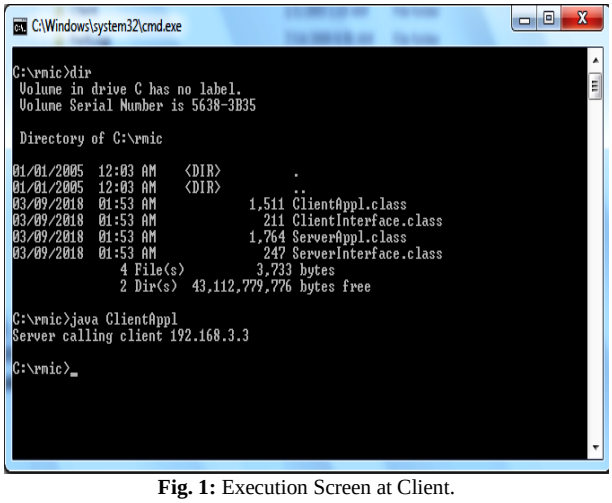

Fig. 1: Execution Screen at Client.

The following screenshot is captured from the server (192.168.3.1) after executing the client application on both the client's 192.168.3.2 and 192.168.3.3.

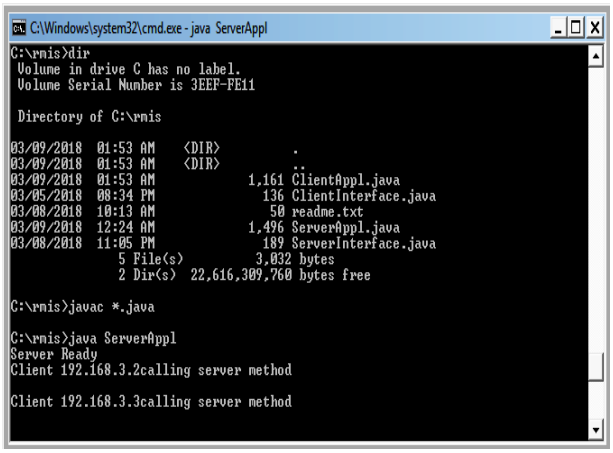

Fig. 2: Execution Screen at Server.

Note the messages displayed in the screenshots of the server and the client command prompts. The server command prompt displays a message "Client 192.168.3.2 calling server method" when the client calls the server method register. Similar message is displayed whenever a client calls the server method. Then another message is displayed as "Server Calling client 192.168.3.3. This message is displayed when the server calls the client method display.

7. Conclusion

In this paper, we attempted to discuss in detail about the classes involved in implementing the Remote Method Invocation feature in Java. We categorized the classes into user defined and built-in classes and explained the usage of these classes with a sample implementation. We have also explained the execution process for testing purposes.

We made an attempt to explain the purpose of each of these classes in implementing the Remote Method Invocation feature. RMI also supports some other functions not discussed here. Our core objective is to provide information about the minimum required classes involved in implementation of RMI to provide a basic knowledge to understand the distributed processing capabilities of RMI.

References

[1] Guillermo L. Taboada, Sabela Ramos, Roberto R. Exposito, Juan Tourino and Ramon Doallo, "Weka-Parallel: Java in High Performance Computing Arena: Research, Practice and Experience", *Science of Computer Programming*, May 14 2011.

[2] Ann Wollrath, Roger Riggs, and Jim Waldo, Simple, "A Distributed Object Model for the Java System", *Proceedings of the USENIX 1996 conference on Object Oriented Technologies* (1996).

[3] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. BAL and Aske Plaat, "An Efficient Implementation of Java's Remote Method Invocation", *Vrije Universiteit*, Amsterdam.

[4] George K. Thiruvathukal, Lovely S. Thomas and Andy T. KorczynskiMaassen, "Reflective Remote Method Invocation", *Faculty Publications*, 1998.

[5] Ninghui Li, John C. Mitchell and Derrick Tong, "Securing Java RMI based Distributed Applications", *Semantics Consistency in Information Exchange*.

[6] Nivedita Joshi and Pooja Singh, "Remote Method Invocation – Usage and Implementation", *International Journal of Engineering and Computer Science*, Volume 2 Issue 11, November 2013.

[7] Hemant Kumar Srivastava, Rounak Sinha and Sumita Gupta, "Implmentation of Socket Programming and RMI Using Simulating Environment", *International Journal of Scientific and Engineering Research*, Volume 4 Issue 5, May 2013.

[8] Harsh Mittal, Manoj Jain and Latha Banda, "Monitoring Local Area Network Using Remote Method Invocation", *International Journal of Computer Science and Mobile Computing*, Volume 2, Issue 5, May 2013.

[9] D. Madhavi, "Transparency in RMI for Distributed Systems: Middleware Layer", *Internation Journal of Emerging Technologies in Engineering Research*, Volume 4, Issue 1, January 2016.