TRIBHUWAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

PULCHOWK, LALITPUR

A Case Study On

Implementation of RMI Using JAVA

SUBMITTED BY:

Ajaya Mandal (071-BCT-501)

Prakriti Dumaru (071-BCT-523)

Sagar Bhandari (071-BCT-532)

Shreeti Shrestha (071-BCT-541)

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

March, 2018

Abstract

In the realm of distributed systems, a key-role is played by the communication among cooperating processes. The Java remote method invocation (RMI) system represent an evolution of the classical remote procedure call (RPC) mechanism – reformulated in the terms of the object-oriented paradigm – which enables programmers to create distributed Java technology-based applications. The advantages of using this approach reside in its simplicity and the possibility to run applications on different platforms. The aim of this work is to implement a simple Java RMI application in order to show its internal behavior and its well-known effectiveness.

1. Introduction

Distributed systems require entities which reside in different address spaces, potentially on different machines, to communicate. Commonly, programming languages provide a basic communication mechanism: ***sockets***. While flexible and sufficient for general communication, the use of sockets requires the design of protocols which is cumbersome and can be error-prone.

An alternative to sockets is ***Remote Procedure Call (RPC).*** RPC systems abstract the communication interface to the level of a procedure call. Thus, instead of having to deal directly with sockets, programmers have the illusion of calling a local procedure when, in practice, the arguments of the call are packaged up and shipped off to the remote target of the call.

However, RPC was not designed for the object-oriented programming and, consequently, does not translate well into distributed object systems where communication between program-level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require ***Remote Method Invocation or RMI***. In such systems, programmers have the illusion of invoking a method of an object, when in fact the invocation may act on a remote object (one not resident in the caller's address space).

Various RMI systems exist but they fall short of seamless integration due to their interoperability requirement with many languages. For instance, ***CORBA*** presumes a heterogenous, multi-language environment and thus must have a language neutral object

model. In contrast, the Java RMI system assumes the homogenous environment of the ***Java Virtual Machine (JVM),*** and the system can therefore follow the Java object model whenever possible.

In order to show its internal behavior and its well-known effectiveness, the aim of this work is to implement a simple Java RMI application in which a client sends to a server an array-list of real values and the server returns its mean, variance and standard deviation.

1.1 Java Object Model

Java is a strongly-typed object-oriented language with a C-like syntax. The main characteristic of Java is ***portability***, i.e. the property of being platform-independent, which means that programs written in the Java language must run similarly on any hardware or operating system platform. This is achieved by compiling code to an intermediate representation called ***bytecode***, instead of directly to specific machine code. Bytecode instructions are analogous to machine code, but they are intended to be interpreted by a virtual machine written specifically for the host hardware.

Clearly, portability is a highly desirable quality in a distributed system scenario because it allows to free programmers from worrying about the underlying architecture of the different machines cooperating in a distributed computation.

Another interesting feature of Java is its separation of the notion of ***interface*** and ***class***. An interface in Java describes a set of methods for an object, but provides no implementation. A class, on the other hand, can describe as well as implement methods. A class may also include fields to hold data, but interfaces cannot. Furthermore, a class may implement any number of interfaces. It is important to understand that there is no other “legal” way in which a process can have access to the data of an object which instantiates a class except by invoking the public methods of its interfaces.

> *This separation is crucial in distributed systems because it allows programmers to place interfaces on one machine, i.e. on the client, and the real object on another, i.e. on the server.*

1.2 Java RMI Architecture

RMI applications often comprise two separate programs, ***a server*** and ***a client***. A typical server program creates some remote objects, makes references to these objects accessible and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

Distributed object applications need to do the following:

a. **Locate remote objects**: Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of the remote invocation.
b. **Communicate with remote objects**: details of the communications are handled by RMI. To programmers, remote communication looks similar to regular Java method invocations.
c. **Load class definitions for objects that are passed around:** As RMI enables objects to be passed back and forth, it provides mechanisms, called ***Dynamic code loading***, for loading an object's class definition as well as for transmitting an object's data.

Like any other Java application, a distributed application built by using Java RMI is made up of ***interfaces and classes.*** The interface declares methods. The classes implement the methods declared in the interfaces. Objects with methods that can be invoked across ***Java Virtual Machines*** are called Remote objects.

An object becomes remote by implementing a remote interface which has the following characteristics:

a. It extends the interface ***java.rmi.Remote***
b. Each method of the interface declares ***java.rmi.RemoteException*** in its throws clause, in addition to any application-specific exceptions.

The RMI system consists of three basic layers:

a. Stub/Skeleton

Rather than making a copy of the implementation object in the receiving JVM, RMI passes a remote stub for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. Similarly, the skeleton is the server-side counterpart of the stub. The ***stub/skeleton layer*** does not deal with specifics of any transport, but transmits data to the remote reference layer via the abstraction of marshal streams. Marshall streams employ a mechanism which enables Java objects to be transmitted between address spaces.

b. Remote Reference Layer

The ***Remote Reference Layer*** is responsible for carrying out the semantics of the type of invocation. For example, this layer is responsible for handling unicast or multicast invocation to a server. Each remote object implementation chooses its own invocation semantics - whether communication to the server is unicast, or the server is part of a multicast group (to accomplish server replication).

c. Transport

Finally, the ***transport layer*** is responsible for connection set-up with remote locations and connection management. A transport defines what the concrete representation of an end-point is, so multiple transports may exist. The design and implementation also allow multiple transports per address space (so both TCP and UDP can be supported in the same address space). Thus, client and server transports can negotiate to find a common transport between them.

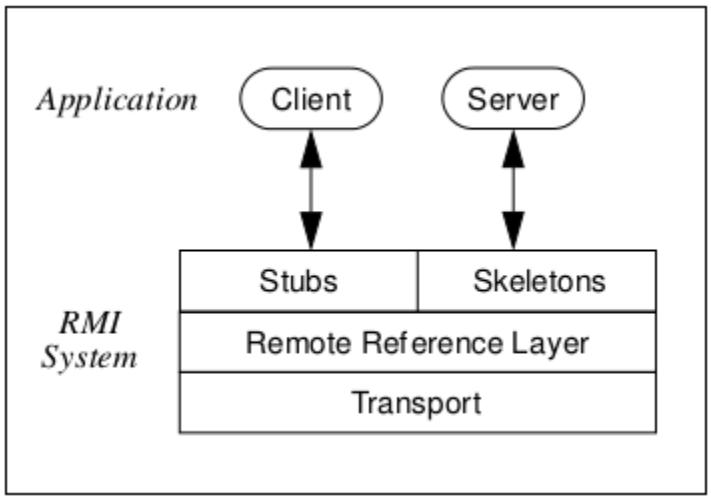


Figure 1: Java RMI architecture

2. Creating Distributed Applications Using RMI

Using RMI to develop a distributed application involves these general steps:

a. Designing and implementing the components of the application

A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods.

b. Compiling sources

As with any Java program, the javac compiler can be used to compile the source files. The source files contain the declarations of the remote interfaces, their implementations, any other server classes and the client classes. Note that with versions prior to Java 5.0, an additional step was required to build stub classes, by using the rmic compiler. However, this step is no longer necessary.

c. Making classes network accessible

Making classes network accessible simply means that client and server must be able to communicate through a network connection and thus they must at least belong to a local ad hoc network.

d. **Starting the application**

Finally, starting the application includes running the server and the client.

3. Implementation Details

The application is implemented in Java and it consists of one remote interface, its implementation, a server class called HelloServant and a client class called Client. The remote interface, called HelloService, simply defines the methods that can be invoked remotely.

Java Implementation of Interface:

```
package rmi;

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

public interface HelloService extends Remote {
        public int add(int a,int b) throws RemoteException;
        public int sub(int a,int b) throws RemoteException;
        public int mult(int a,int b) throws RemoteException;
        public float div(int a,int b) throws RemoteException;
}
```

Implementation of Remote Interface:

```
package rmi;
import java.rmi.RemoteException;
```

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

public class HelloServant extends UnicastRemoteObject implements HelloService{
        private static final long serialVersionUID = 1L;
        public HelloServant() throws RemoteException {
                super();
        }

        public int add(int a, int b) throws RemoteException {
          int result=a+b;
          return result;
   }
        public int sub(int a, int b) throws RemoteException {
          int result=a-b;
          return result;
        }
        public int mult(int a, int b) throws RemoteException {
          int result=a*b;
          return result;
        }
        public float div(int a, int b) throws RemoteException {
          float result=a/b;
          return result;
        }

}
```

Implementation of Server:

```
package rmi;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class ApplicationServer {

        public static void main( String[] args ) throws RemoteException {
                Registry registry = java.rmi.registry.LocateRegistry.createRegistry(9999);
                registry.rebind("hello", new HelloServant());
        }
}
```

Implementation of Client:

```
package rmi;

import java.rmi.Naming;
```

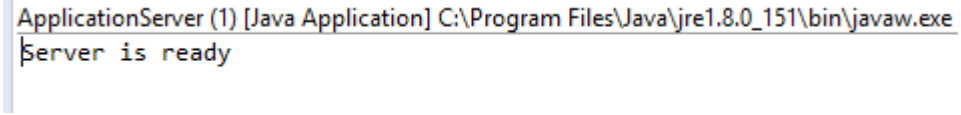
```
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.Scanner;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Client {
	private static Scanner scanner = new Scanner( System.in );
	public static void main( String[] args) throws NotBoundException,
MalformedURLException,RemoteException{
		HelloService service = (HelloService)
Naming.lookup("rmi://localhost:9999/hello");
		boolean exit= false;
		while (!exit ) {
			System.out.println("Select the desired function by entering the
number:");
			System.out.println("1. Addition");
			System.out.println("2. Subtraction");
			System.out.println("3. Multiplication");
			System.out.println("4. Division");
			System.out.println("0. Exit");

			InputStreamReader reader= new InputStreamReader (System.in);
			BufferedReader input = new BufferedReader(reader);
			String commandInput;

			int command = 0	;

			try{
				commandInput = input.readLine( );
				command = Integer.parseInt(commandInput);
			} catch(IOException e){
				System.out.println("Error Occured:"+ e.getMessage());
			}

			switch(command){
				case 1:
				{
					System.out.println("\n Enter the numbers:");
					int x= scanner.nextInt();
					int y= scanner.nextInt();
					int result=service.add(x,y);
					System.out.println("Result is :"+result);
```

```
                    break;
                }
                case 2:
                {
                    System.out.println("\n Enter the numbers:");
                    int x= scanner.nextInt();
                    int y= scanner.nextInt();
                    int result=service.sub(x, y);
                    System.out.println("Result is :"+result);
                    break;
                }
                case 3:
                {
                    System.out.println("\n Enter the numbers:");
                    int x= scanner.nextInt();
                    int y= scanner.nextInt();
                    int result=service.mult(x,y);
                    System.out.println("Result is :"+result);
                    break;
                }
                case 4:
                {
                    System.out.println("\n Enter the numbers:");
                    int x= scanner.nextInt();
                    int y= scanner.nextInt();
                    float result=service.div(x,y);
                    System.out.println("Result is :"+result);
                    break;
                }
                case 0:
                {
                    exit = true;
                    break;
                }
            }
        }
    }
}
```

4. Results

The following are the snapshots of the outputs observed from the Client side by running the Client implementation:

a. Server Side:



b. Client Side:

```
Select the desired function by entering the number:
1. Addition
2. Subtraction
3. Multiplication
4. Division
0. Exit
3

 Enter the numbers:
4
5
Result is :20
Select the desired function by entering the number:
1. Addition
2. Subtraction
3. Multiplication
4. Division
0. Exit
2

 Enter the numbers:
6
4
Result is :2
Select the desired function by entering the number:
1. Addition
2. Subtraction
3. Multiplication
4. Division
0. Exit
4

 Enter the numbers:
2
2
Result is :1.0
Select the desired function by entering the number:
1. Addition
2. Subtraction
3. Multiplication
4. Division
0. Exit
1
```

Conclusion

Java RMI leverages the basic assumptions of platform independence, which is very important in the area of distributed systems where machines are typically different. We can assume independence due to the architecture neutrality that the Java Virtual Machine provides. In this case study we have seen a small application that demonstrates how the mechanism is actually simple and effective and how it allows to develop much more complex applications with equal ease.