EAST WEST UNIVERSITY

RMI Based Distributed Query Processing

Submitted By

Avirupa Roy Talukder

ID: 2011-2-60-001

Alok Kumar Roy

ID: 2011-2-60-045

Supervised by

Dr. Shamim Akhter

Assistant Professor

Department of Computer Science & Engineering

East West University

The project has been submitted to the Department of the Computer Science & Engineering at East West University in the partial fulfillment of the requirement for the degree of Bachelor of Science in Computer Science and Engineering**.**

DECLARATION

The project has been submitted to the Department of the Computer Science & Engineering. East West University in the partial fulfillment of the requirement for the degree of Bachelor of Science in Computer Science & Engineering performed by me under supervision of Assistant Professor Dr. Shamim Akhter, Department of Computer Science & Engineering at East West University. This is also needed to certify that, the project work under the course **'Project (CSE497)'.** We, hereby, declare that this project has not been submitted elsewhere for the requirement of any degree or diploma or any other purposes.

Signature of the candidates

(Avirupa Roy Talukder)

(Alok Kumar Roy)

LETTER OF ACCEPTANCE

The Project entitled **RMI Based Distributed Query Processing** submitted by Avirupa Roy Talukder, Id: 2011-2-60-001 and Alok Kumar Roy, Id: 2011-2-60-045 to the Department of Computer Science and Engineering, East West University, Dhaka, Bangladesh is accepted as satisfactory for the partial fulfillment of the requirement for the degree of Bachelor of Science in Computer Science and Engineering on August 10, 2016.

Board of Examiners

1. **----------**

Dr. Shamim Akhter
Assistant Professor (Supervisor)
Department of Computer Science & Engineering
East West University, Dhaka, Bangladesh

2. **----------**

Dr. Md. Mozammel Huq Azad Khan
Professor & Chairperson (Chairperson)
Department of Computer Science & Engineering
East West University, Dhaka, Bangladesh

Abstract

In this project, we have developed distributed query processing system with Java RMI based system. The system is developed for the computer that doesn't have any Oracle/SQL Server or Database and does not have enough memory to afford Oracle/SQL Server. However, we can achieve benefit of the Database/SQL server. Here, we introduce the necessary functions to implement the project.

RMI (Remote Method Invocation) is used to call the distributed function to make server side Database connection. RMI based Distributed Query Processing is a smart system to connect two computers under a network. At first, we create client and server. There is a database in server side to connect the server with Database through JDBC (mysql-connector.jar).The client sends its query to the server and the sever checks the database for processing the query and after the query processing results send back to clients.

ACKNOWLEDGEMENT

First of all we express our gratefulness to the Almighty, without His divine blessing it would not be possible to complete this project successfully. It has been a great pleasure to us to develop **RMI Based Distributed Query Processing**. We have gathered sufficient knowledge and experience during this project.

We would like to thank our honorable teacher and supervisor of this project, Dr. Shamim Akhter, Assistant Professor, Department of Computer Science and Engineering, East West University who guided us to proper analysis of the system and helped to develop an elegant and efficient system. He does not only give us the great idea, but also encouraged us to seek out the clearest and deepest description of theoretical ideas and experimental findings. We are very grateful to him for his continuous support, advice and guidance. It was a great pleasure to study and work with gifted people like him who influenced us in many ways.

Finally, we would like to convey our special thanks to our parents whom have always given us tremendous support. Without their love and encouragement, we would not had achieve this far.

TABLE OF CONTENTS

Title		Page
Declaration		i
Letter of Acceptance		ii
Abstract		iii
Acknowledgement		iv
Table of Contents		v-viii
Chapter 01	**Introduction**	**01-03**
1.1 Overview		02
1.2 Motivation		02
1.3 Objective		02
1.4 Contribution		02
1.4 Organization of the project		03
Chapter 02	**Background Study**	**04-07**
2.1 Distributed System		05
2.2 What is RMI		05
2.3 Participating process		06
2.4 How RMI works		06
2.5 Advantages		07
Chapter 03	**RMI Implementation**	**08-23**
3.1 Java RMI: Hello World (in 1 PC)		09

3.2 Java RMI: Adding 2 Numbers (Between 2 PCs) 13

3.3 Java RMI: String Passing (Between 2 PCs) 18

Chapter 04 RMI with Database 24-36

4.1 Setting Local host connection with Database 25

4.2 Define the Remote Interfaces 25

4.3 Define the Remote Classes 26

4.4 Creating the Server 28

4.5 Create the Java Client Program 30

4.6 Output of the Project 35

Chapter 05 Conclusion and Future Work 37-38

5.1 Conclusion 38

5.2 Future Work 38

Appendix 39

References 45

List of Figures:

Figure 2.1	RMI Architecture	5
Figure 3.1	rmic of Hello World program	11
Figure 3.2	Starting Registry	12
Figure 3.3	Start server of Hello world	12
Figure 3.4	Running Client of Hello worlds	13
Figure 3.5	Creating class and stub file of two number	16
Figure 3.6	Start the rmiregistry	17
Figure 3.7	Starting server of two number addition	17
Figure 3.8	Output of two number addition	18
Figure 3.9	Creating Stub file of String pass program	21
Figure 3.10	Start the registry 3001	22
Figure 3.11	Starting the server	22
Figure 3.12	Create class file and run the client	23
Figure 4.1	Create class file of Addition and AdditionServer	26
Figure 4.2	rmic Addition	28
Figure 4.3	start rmiregistry	29
Figure 4.4	Execution of the server	29
Figure 4.5	Database of Students	30
Figure 4.6	Excecution of client	31
Figure 4.7	Option choice window	31
Figure 4.8	Add new record into database	32
Figure 4.9	Inserted database	32
Figure 4.10	Command of delete a record from database	33
Figure 4.11	Deleted Database	33

Figure 4.12	Choosing retrieve option	34
Figure 4.13	Command of retrieve a record from database	34
Figure 4.14	Retrieved result	35
Figure 4.15	Choosing Update option	35
Figure 4.16	Command of update a record into database	36
Figure 4.17	Updated databases	36

Chapter 1

Introduction

1.1 Overview

In this project, we propose to work with RMI and sever to solve the problem. The main reason to work with this system is flexibility and maintainability. In this section, we present a blend of all features and technologies to introduce RMI architecture.

The server main task is managing network resources. We use an IP address and port number which plays a vital role to connect client and server. To generate the remote method client should connect to the server and request a method to execute.

The client sends parameters to server and after getting Instruction server compute the Instruction and sends result to client. The client is also equipped with any device that capable to handle the request and response. In this project, we have developed a system that doesn't require any Oracle/SQL Server.

1.2 Motivation

For our project we consider the query processing and show result to the client system are our target object. Remote Method Invocation primary task is to access remote system. We don't know where the server is. We only know that we have a server and clients can access this server without the help of installing SQL Server.

We tried to develop a system for those device whose have limitations to access Database and get information without install Oracle/SQL Server.

1.3 Objective

Specific objective of this project includes:

- Study and analysis of Remote Method Invocation architecture and its existing features.
- Implement a database and connect database with server.

1.4 Contribution

- To run jdk, on our respective system first we set jdk class path as an environment variable.
- To run mysql connector as our respective file set file path and mysql connector as an class path variable.
- Design and implementation of client which will allow user to use database without installing Database at their side.

1.5 Organization of the project

As we try to develop a RMI Based Distributed Query Processing. First of all we define an interface to declares remote methods .Then Implement the remote interface and the server. Before that we will explain briefly about RMI registry. Then we will try to explain how client and server communicate with each other. We will describe how to connect JDBC with RMI. Next comes the Query processing discussion which will pass from client system.

Chapter 2
BackGround Study

2.1 Distributed System

A distributed system is a collection of independent computers that appears to its users as single coherent systems. The difference between the various computers and the way in which they communicate are hidden from users. An important characteristic of distributed system is that user and application can interact with a distributed system in a consistent and uniform way. We define a distributed system as one in which hardware and software components located at networked computers communicate and coordinate their action only by passing massage [1].

2.2 What is RMI?

RMI is a mechanism for communicating between two machines running Java Virtual Machines. It permits Java methods to refer to a remote object and invoke methods of the remote object. When Java code on machine A needs a service or a method, respectively, of that means object B on machine *B* it starts a remote method invocation.[2] The remote object may reside on another Java virtual machine, the same host or on completely different hosts across the network. It allows any data type .RMI allows both Client and Server to load new object types as required.

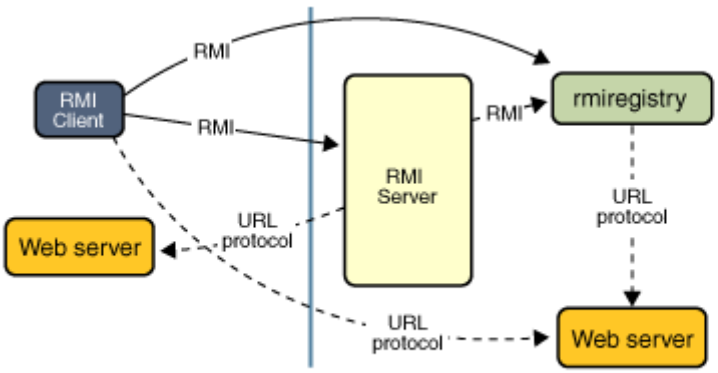

Figure 2.1: RMI Architecture

2.3 Participating process

Client: A client is the receiving end of a service or the requestor of a service in a client/server model type of system.[3] The client is located on another computer or system which has to access via network. In java RMI, client is a process to call a method of remote object.

Server: The system which main task is to managing network resources is called server. To implement the remote methods we write this program – clients connect to the server and request that a method be executed. The remote methods to the client are local methods to the server.

Registry Service: A Registry Service is an application that provides the facility of registration & lookup of Remote stub.[3] The object registry runs on a known port (1099 by default) A server registers its objects with a textual name with the object registry when it starting. A client, before performing invoking a remote method, must first contact the object registry to obtain access to the remote object.

Stub: A stub is proxy that stands for RMI Server on client side and handles remote method invocation on behalf of RMI Client.

Skeleton: It is a proxy that stands for RMI client on server side and handles remote method invocation on RMI Server on behalf of client [3]. It unmarshalls the arguments in the request massages and invokes the corresponding method in the remote object. But we need not use skeleton because we use updated version of JDK.

Bind (): This method is used to register a remote object (stub of object) in the rmiRegistry.

Rebind (): This method is used to rebind the remote object in the rmiRegistry.

Lookup (): This method is used to lookup remote stub in the rmiRegsitry.

2.4 How RMI works

- In distributed system, RMI must have to keep record of the distributed objects .That's why RMI uses a network-based registry. By binding it to a name in the registry the server object makes a method available for remote invocation.[4] The client object can check for availability of an object by looking up its name in the registry. The registry acts as a limited central management point for RMI. The registry is simply a name repository. It does not address the problem of actually invoking the remote method.

- The two objects may be resided on separate machines. A mechanism is used to transmit the client's request to call a method on the server object to the server object and provide a response. The code for the server object must be processed by an RMI compiler called rmic, which is part of the JDK.[4]

- The rmic compiler generates two files: a stub and a skeleton. The stub resides on the client machine and the skeleton resides on the server machine. The stub and skeleton are comprised of Java code that provides the necessary link between the two objects.

    - The stub object has to build an information block that's consists of
        - an identifier of the remote object to be used,
        - an operation number describing the method to be called and
        - the marshalled parameters (method parameters have to be encoded into a format suitable for transporting them across the net)
        - send this information to the server

    - The tasks of the skeleton object are:
        - to unmarshal the parameters
        - to call the desired method on the real object lying on the server,
        - to capture the return value or exception of the call on the server, to marshal this value, to send a package consisting of the value in the marshalled form back to the stub on the client, machine A.

- When a client invokes a server method, the JVM looks at the stub to do type . The request is then routed to the skeleton on the server [4], which in turn calls the appropriate method on the server object. In other words, the stub acts as a proxy to the skeleton and the skeleton is a proxy to the actual remote method.

2.5 Advantages

Java RMI has numerous advantages, as follows:

- Portable to any JVM.
- Easy to write/Easy to maintain: Facilitates write remote Java servers and Java clients that access those server.
- Safe and secure: Uses built-in Java security mechanisms to facilitate system safety during user download implementations.

Chapter 3
RMI Implementation

3.1 Java RMI: Hello World (in 1 PC)

At first we try to run a “Hello World!” program with java RMI. It is a distributed version using Java RMI. The distributed Hello World example make a remote method call to the server, so that it could retrieve the message "HelloWorld!". When the client runs, It receives the "Hello, world!" message from the server. For execution of this, we need to fulfill these steps:

Define the functions of the remote class as a Java interface

A remote object is an instance of a class that implements a remote interface. A remote interface extends the interface java.rmi.Remote. It declares a set of remote methods. Each remote method must declare java.rmi.RemoteException.

```
 Import java.rmi.Remote;
Import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Implement the server

Server class has a main method that creates an instance of the remote object implementation, exports the remote object, and then binds that instance to a name in a Java rmi registry.

```
importjava.rmi.Naming;
importjava.rmi.RemoteException;
importjava.rmi.server.UnicastRemoteObject;
importjava.rmi.registry.LocateRegistry;
importjava.rmi.registry.Registry;

public class HelloImpl extends UnicastRemoteObject implements Hello
{
publicHelloImpl() throws RemoteException {}

public String sayHello()
{
return "Hello world!";
}
public static void main(String args[])
{
try
{
HelloImplobj = new HelloImpl();
Registry registry = LocateRegistry.getRegistry(2003);
registry.bind("HelloServer", obj);
}
catch (Exception e)
```

```
{
System.out.println("HelloImpl err: " + e.getMessage());
e.printStackTrace();
}
}
}
```

Create and export a remote object

The main method of the server needs to create the remote object that provides the service. When we extend java.rmi.server.UnicastRemoteObject, class is automatically exported. This can be done as follows:

```
                    Server obj = new Server();
    Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
```

Instantiate a remote object

The main method of the server creates an instance of the remote object implementation:

```
                    HelloImplobj = new HelloImpl();
```

Register the remote object with a Java RMI registry

```
          Registry registry = LocateRegistry.getRegistry();
                    registry.bind("Hello", stub);
```

Implement the client

The client part of the distributed Hello World example remotely invokes the sayHello method in order to get the string "Hello world!", which is output when the client runs. Here is the code for the client:

```
importjava.rmi.RMISecurityManager;
importjava.rmi.Naming;
importjava.rmi.RemoteException;
importjava.rmi.registry.LocateRegistry;
importjava.rmi.registry.Registry;

public class HelloClient
{
public static void main(String arg[])
{
String message = "blank";

System.setProperty("java.security.policy","c:\\hello.policy");
System.setSecurityManager(new RMISecurityManager());
```

```
try
{

Registry registry = LocateRegistry.getRegistry( 2003);
Hello obj = (Hello) registry.lookup("HelloServer");
System.out.println(obj.sayHello());
}
catch (Exception e)
{
System.out.println("HelloClient exception: " + e.getMessage());
e.printStackTrace();
}
}
}
```

Compile the Java source files

To compile the Java source files, run the javac command as follows:

```
Javac Hello.java HelloImpl.java HelloClient.java
```

Use rmic to generate stubs

In this" Hello World!" example ,to create the stub for the HelloImpl remote object implementation, the command of run rmic is given below:

rmicHelloImpl

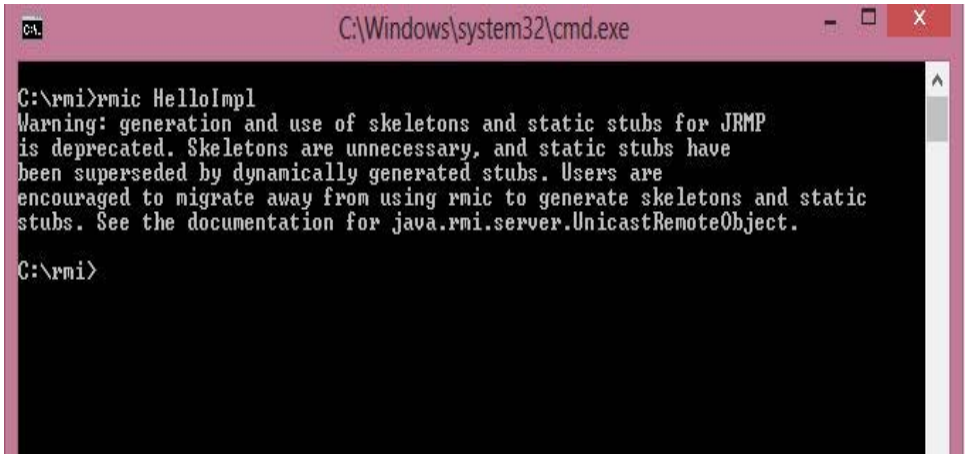

Figure 3.1: rmic of Hello World program

Start the RMI registry

startrmiregistry 2003

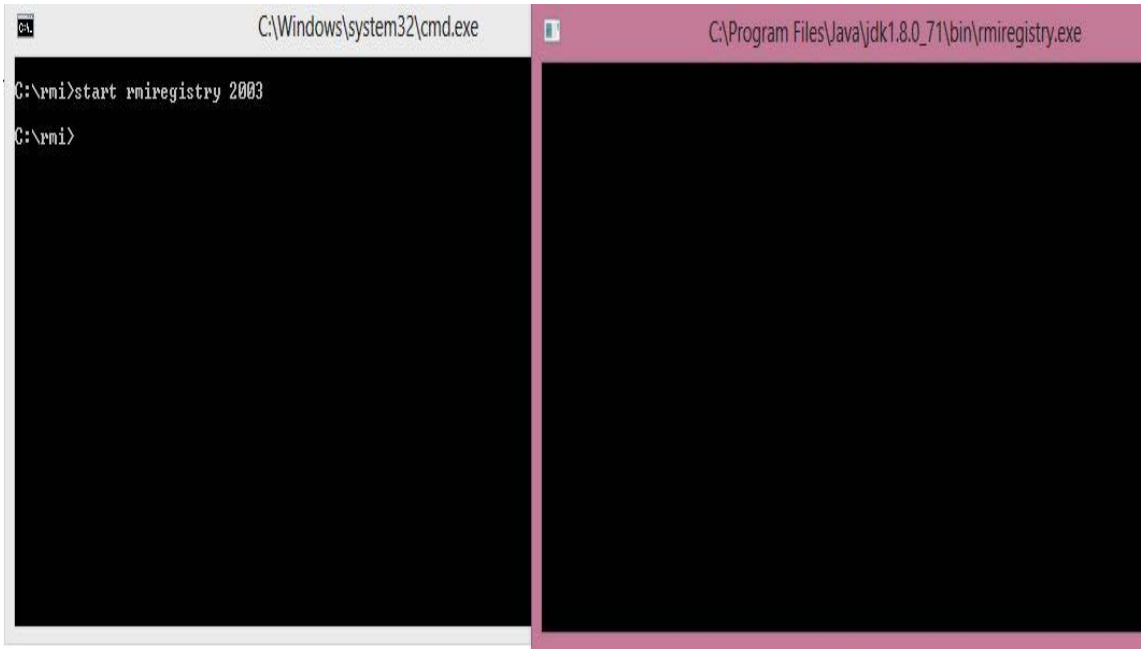

Figure 3.2: Starting Registry

Start the server

java HelloImpl

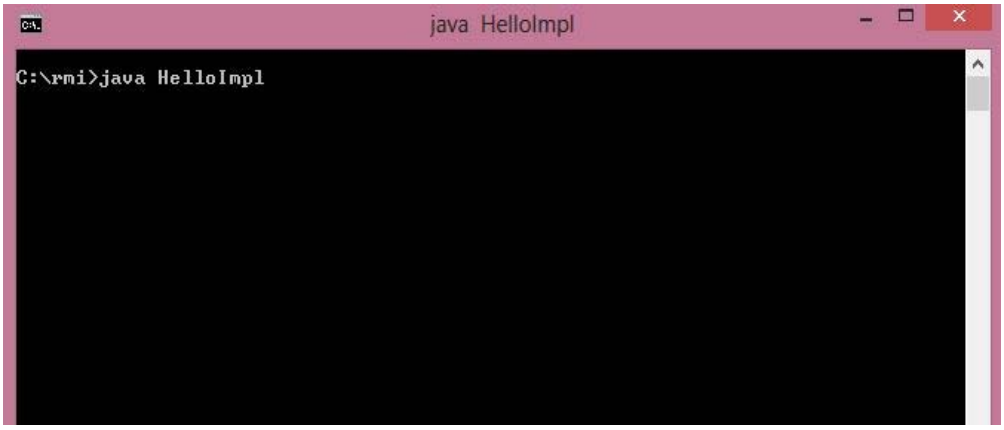

Figure 3.3: Start server of Hello world

Run the client

java HelloClient

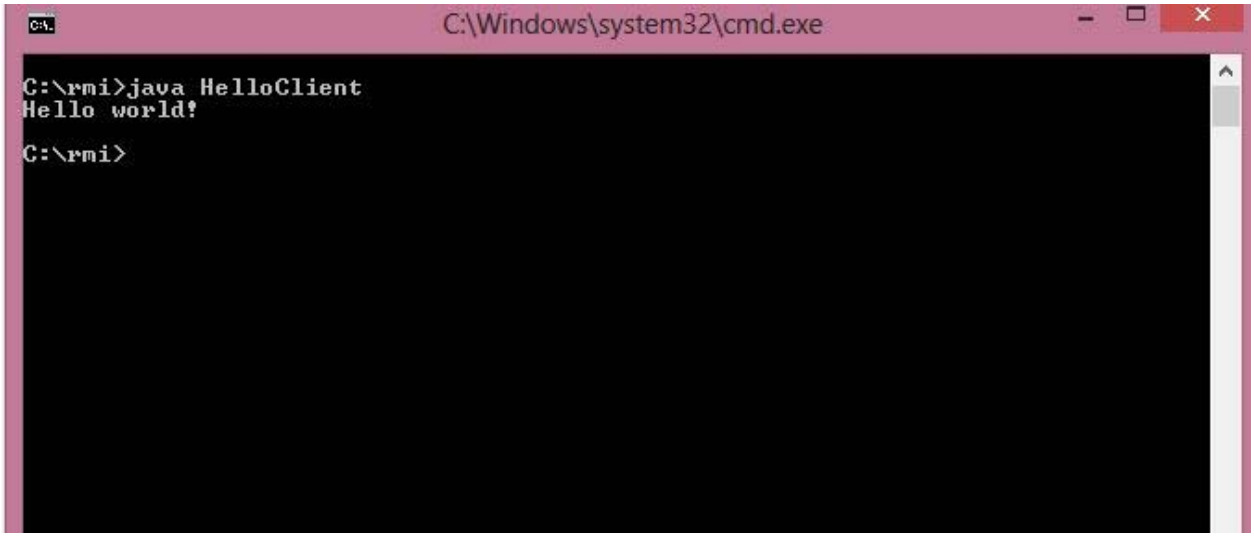

Figure 3.4: Running Client of Hello world

3.2 Java RMI: Adding 2 Numbers (Between 2 PCs)

In this example, the request specifies two numbers; the server adds these together and returns the sum. To create this application, we need to use four files. The first file is AdditionInterface.java. It defines the remote interface that is provided by the server. It contains one method that accepts two arguments and returns their sum to the client.

Write and Remote interface that declares remote methods

The first file AdditionalInterface.java defines the remote interface. It includes one method that accepts two arguments and returns their sum.

```
importjava.rmi.Remote;
importjava.rmi.RemoteException;

public interface AdditionalInterface extends Remote {
public int Add(int a, int b) throws RemoteException;
}
```

Implements the remote interface

The second source file which is Addition.java, implements the remote interface. All remote objects must extend UnicastRemoteObject, which provides functionality that is needed to make objects available from remote machines.

```
importjava.rmi.RemoteException;
importjava.rmi.server.UnicastRemoteObject;

public class Addition extends UnicastRemoteObject implements
AdditionalInterface {
private static final long serialVersionUID = 1L;

public Addition() throws RemoteException {
           }

publicint Add(int a, int b) {
return a + b;
    }
}
```

Implement the Server

The third source file AdditionServer.java contains the main program for the server machine.Its fundamental function of this Source file is to update the RMI registry on that machine. This is done by using the rebind( ) method of the Naming class which found in java.rmi.Naming.

```
importjava.net.MalformedURLException;
importjava.rmi.Naming;
importjava.rmi.RemoteException;
importjava.rmi.registry.LocateRegistry;
public class AdditionServer {
public static void main(String[] argv) throws RemoteException {
        Addition Hello = new Addition();

int port = 3001;

try {

LocateRegistry.createRegistry(port);
System.out.println("java RMI registry created.");
        } catch (RemoteException e) {
System.out.println("java RMI registry already exists.");
        }
String hostname = "192.168.109.50";

        String bindLocation = "//" + hostname + ":" + port + "/Hello";
```

```
try {
Naming.bind(bindLocation, Hello);
System.out.println("Addition Server is ready at:" + bindLocation);
        } catch (RemoteException e) {
e.printStackTrace();
        } catch (MalformedURLException e) {
e.printStackTrace();
        } catch (Exception e) {
System.out.println("Addition Serverfailed: " + e);
        }
    }
}
```

Develop client Side

The fourth source file, AdditionClient.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. Server uses port 3001 to define path. The client program illustrates the remote call by using the method Add(9,10) that will be invoked on the remote server machine from the local client machine where the client runs.

```
importjava.net.MalformedURLException;
import java.rmi.*;
public class AdditionClient {
public static void main(String[] args) {
        String remoteHostName = "192.168.1.2";
intremotePort = 3001;
        String connectLocation = "//" + remoteHostName + ":" +
remotePort
                + "/Hello";
AdditionalInterface hello = null;
try {
System.out.println("Connecting to client at : " + connectLocation);
hello = (AdditionalInterface) Naming.lookup(connectLocation);
        } catch (MalformedURLException e1) {
            e1.printStackTrace();
        } catch (RemoteException e1) {
e1.printStackTrace();
        } catch (NotBoundException e1) {
e1.printStackTrace();
        }
int result = 0;
try {
result = hello.Add(9, 10);
        } catch (RemoteException e1) {
            e1.printStackTrace();}
System.out.println("Result is :" + result);
    }
}
```

Compile the Java source files

To compile the Java source files, run the javac command as follows:

```
Javac Hello.java HelloImpl.java HelloClient.java
```

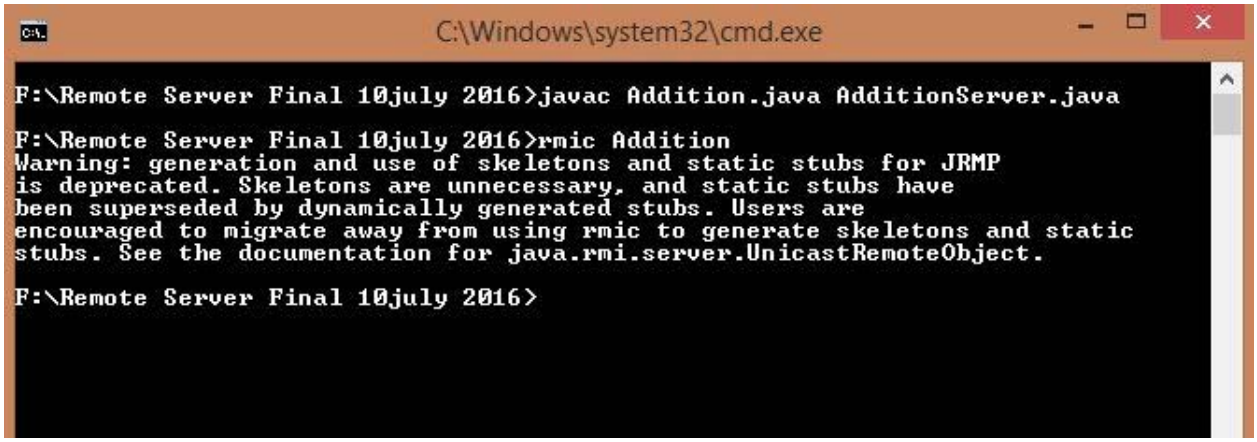

Figure 3.5: Creating class and stub file of two numbers addition

Generate stub

Compile all Java source files and keep all generated .class files in the same directory. We must generate the necessary stub. Before use the client and the 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.To generate stubs, we use a tool called the RMI compiler, which is invoked from the command line, as shown here:

```
rmic Addition
```

When using rmic, ensure that class path is set-to include the current directory. By default, rmic generates both stub and skeleton file.

Start the RMI Registry on the Server

We have to go to the directory on the server machine where we keep source files and then check that the class path environment variable includes the directory in which files are located .

Start the RMI Registry from the command line

```
startrmiregistry 3001
```

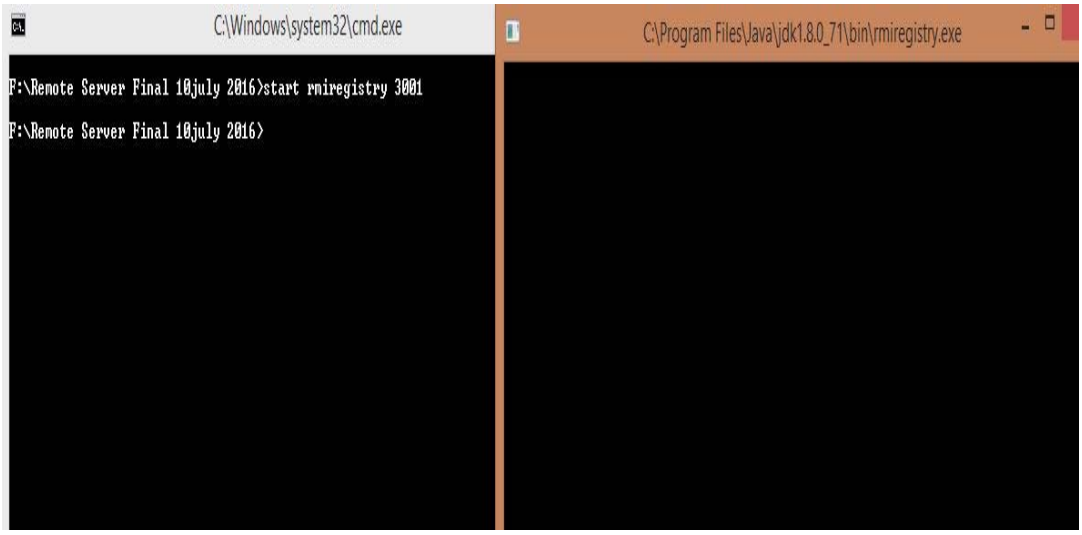

Figure 3.6: Start the rmiregistry

When this command returns, We see a new window has been created.

Start the Server

The server code is started from the command line, as shown here:

```
java AdditionServer
```

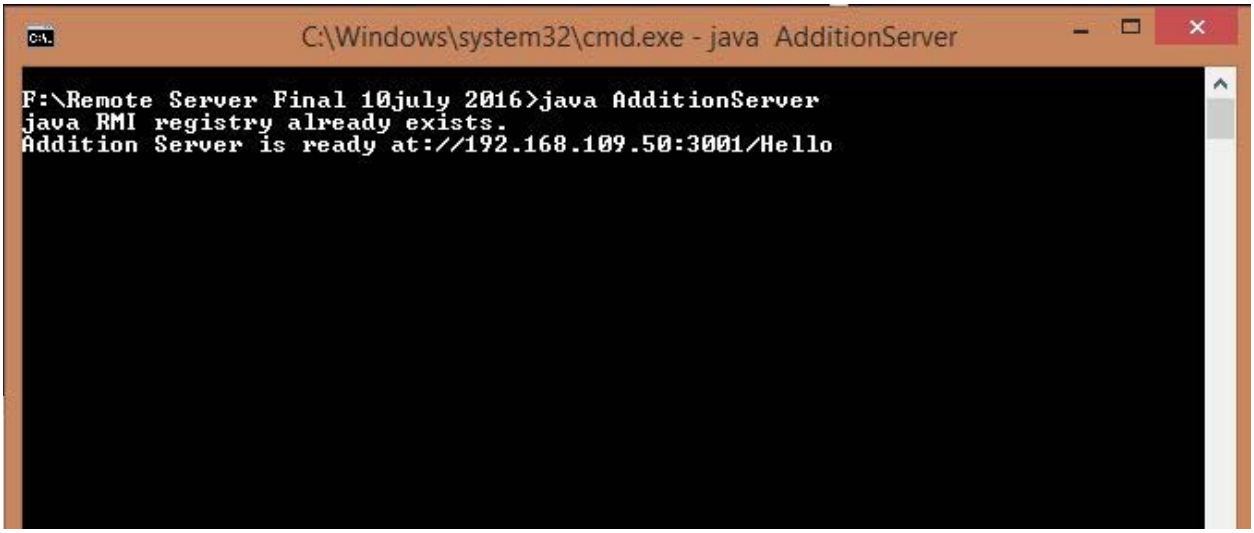

Figure 3.7: Starting server of two number addition

Start the Client

The AdditionClient software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. The client code is started from the command line, as shown here:

java Addition client

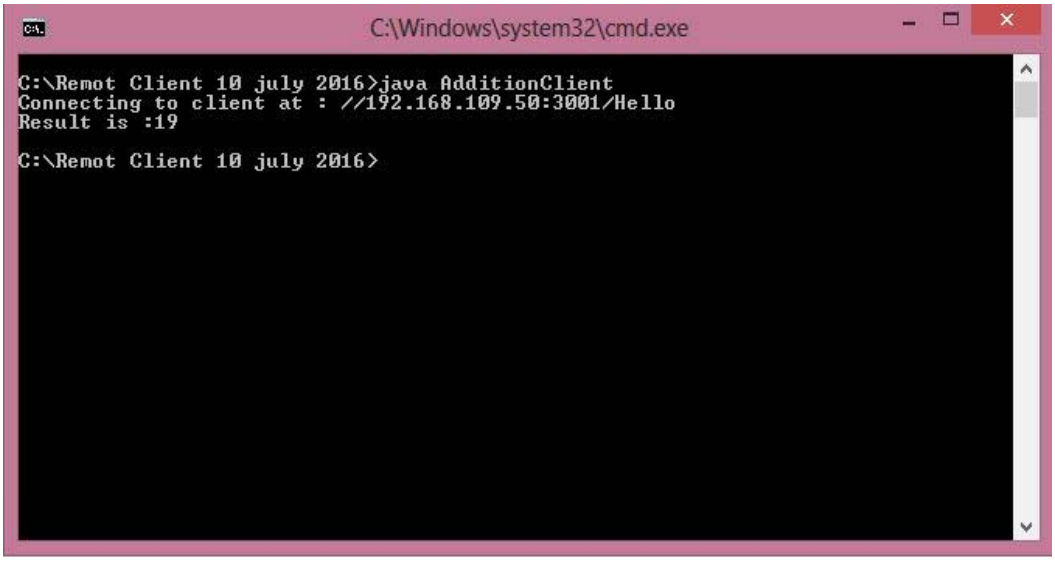

Figure 3.8: Output of two numbers addition

3.3 Java RMI: String Passing (Between 2 PCs)

In this example, we try to pass a string with java RMI. It is a distributed version using Java RMI. The distributed Hello Client example makes a remote method call to the server, so that it could retrieve the message "Hello". When the client runs, The server pass a massage and the receives the "Hello" message from the server.

Write a Remote interface that declares remote methods

The first file AdditionalInterface.java defines the remote interface. It includes one method that accepts string argument and returns their massage.

```
importjava.rmi.Remote;
importjava.rmi.RemoteException;

public interface AdditionalInterface extends Remote {
public String Add(String str) throws RemoteException;
}
```

Implements the remote interface

The Addition.java files which implements the remote interface. All remote objects must extend UnicastRemoteObject, which provides functionality that is needed to make objects available from remote machines. When we extend java.rmi.server. UnicastRemoteObject, class is automatically exported. This can be done as follows:

```
importjava.rmi.RemoteException;
importjava.rmi.server.UnicastRemoteObject;

public class Addition extends UnicastRemoteObject implements
AdditionalInterface {
private static final long serialVersionUID = 1L;

public Addition() throws RemoteException {
    }
public String Add(String str) {
return "Hello Client";
    }
}
```

Implement the server

Since RMI is network based, it is essential to use networking terminology .As we know, Networking is based on the notion of hosts, servers, and clients. Server class has a main method that creates an instance of the remote object implementation, exports the remote object, and then binds that instance to a name in a Java RMI registry. We can treat each computer as a host. Hosts have names and we use network address as name= "103.230.5.14". the host which provide services are called servers. Clients are machines which use these services. This is done by having a registry service. With a registry service, servers can register services and clients can lookup services and find the corresponding servers.

```
importjava.net.MalformedURLException;
importjava.rmi.Naming;
importjava.rmi.RemoteException;
importjava.rmi.registry.LocateRegistry;

public class AdditionServer {
public static void main(String[] argv) throws RemoteException {
        Addition Hello = new Addition();
int port = 3001;
try {
LocateRegistry.createRegistry(port);
System.out.println("java RMI registry created.");
        } catch (RemoteException e) {
System.out.println("java RMI registry already exists.");
        }
```

```
        String hostname = "192.168.109.50";
        String bindLocation = "//" + hostname + ":" + port + "/Hello";
try {
Naming.bind(bindLocation, Hello);
System.out.println("Addition Server is ready at:" + bindLocation);
        } catch (RemoteException e) {
e.printStackTrace();
        } catch (MalformedURLException e) {
e.printStackTrace();
        } catch (Exception e) {
System.out.println("Addition Serverfailed: " + e);
        }
    }
}
```

Implement the client

The fourth source file, AdditionClient.java, implements the client side of this distributed application. AdditionClient.java requires three command line arguments. The first is the IP address or name of the server machine. The second arguments is the two string that are to be passed. Server uses port 3001 to communicate with Client desktop.

```
importjava.net.MalformedURLException;
importjava.rmi.*;
public class AdditionClient {
public static void main(String[] args) {
        String remoteHostName = "192.168.109.50";
intremotePort = 3001;
        String connectLocation = "//" + remoteHostName + ":" +
remotePort
                + "/Hello";
AdditionalInterface hello = null;
try {
System.out.println("Connecting to client at : " + connectLocation);
hello = (AdditionalInterface) Naming.lookup(connectLocation);
        } catch (MalformedURLException e1) {
e1.printStackTrace();
        } catch (RemoteException e1) {
e1.printStackTrace();
        } catch (NotBoundException e1) {
e1.printStackTrace();
        }

String result="0";
try {
result = hello.Add("Hello");
        } catch (RemoteException e1) {
```

```
e1.printStackTrace();
        }
System.out.println("Result is :" + result);
    }
}
```

Generate stub

In this step , we have to compile all Java source files and keep all generated **.class** files in the same directory and generate the necessary stub before use the client and the server. The function of stub is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub.

We have to set class path include the current directory before using rmic.To generate stubs ,we use a tool called the RMI compiler , which isinvoked from the command line, as shown here:

rmic Addition

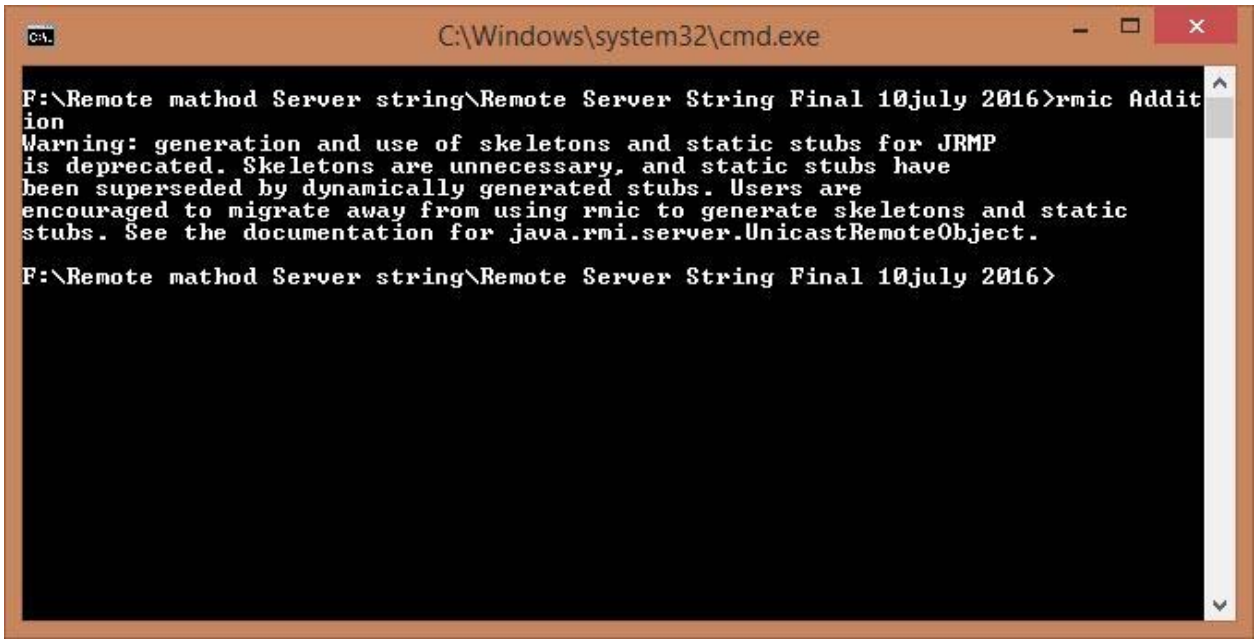

Figure 3.9: Creating Stub file of String pass program

Start the RMI Registry on the Server

We have to go to the directory on the server machine where we keep source files and then check that the class path environment variable includes the directory in which files are located . Then, Start the RMI Registry from the command line:

startrmiregistry 3001

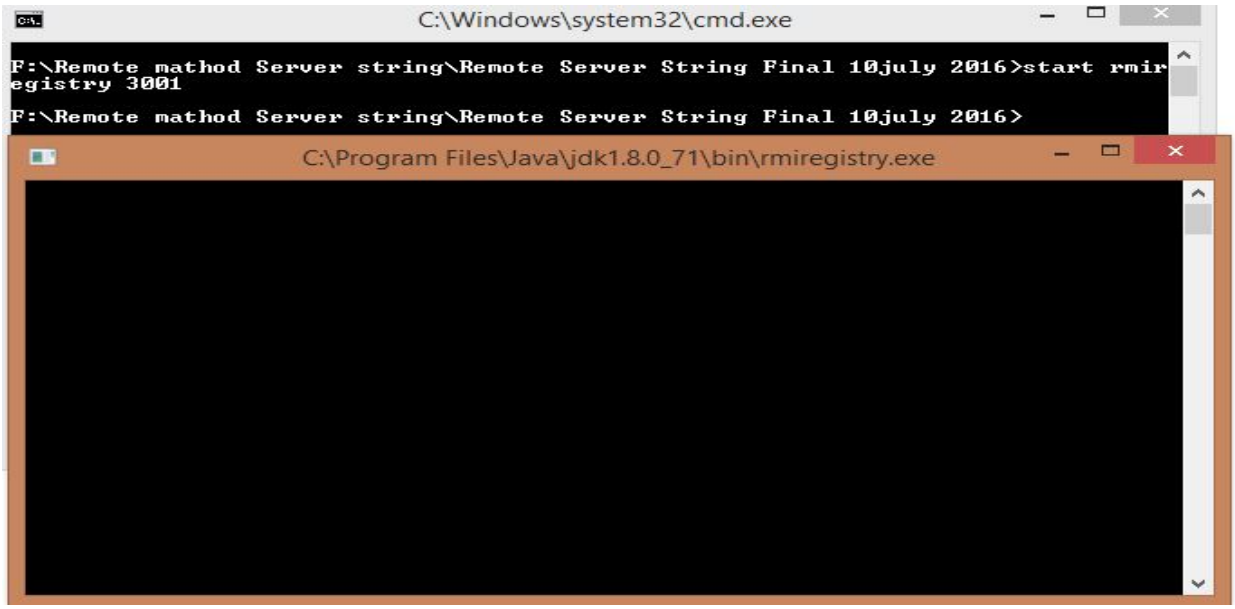

Figure3.10: Start the registry 3001

Start the Server

The server code is started from the command line, as shown here:

javaAdditionServer.

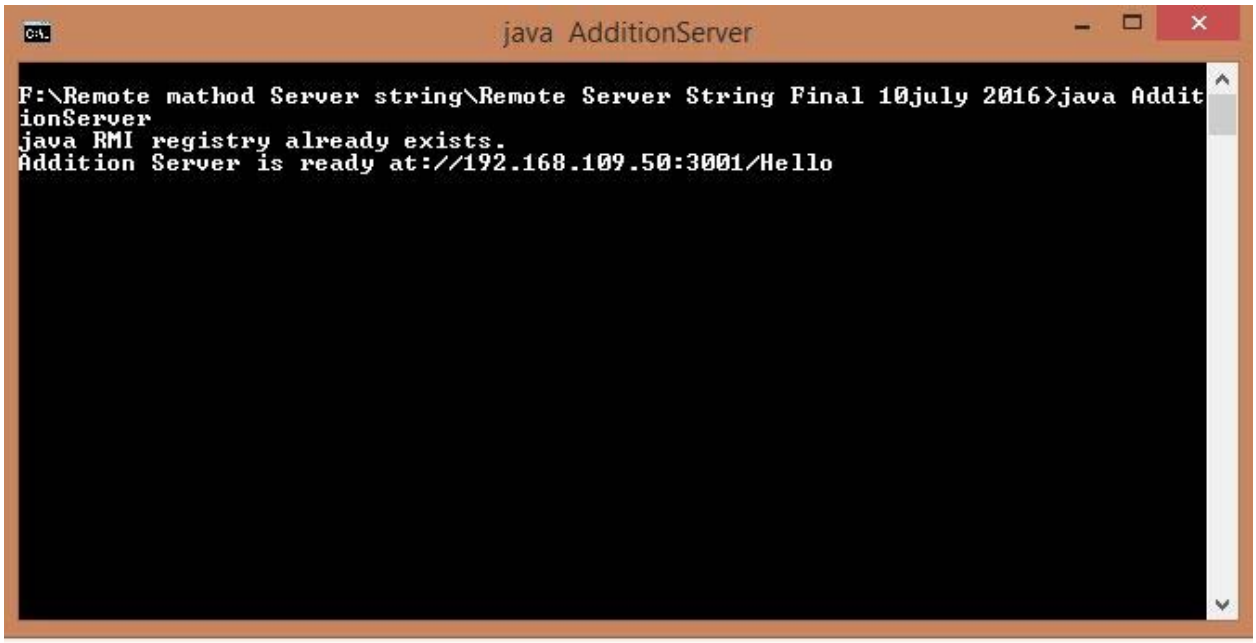

Figure 3.11: Starting the server

Start the Client

The AdditionClient software requires two arguments: the name or IP address of the server machine and the String .command is given below:

javac AdditionClient java AdditionClient

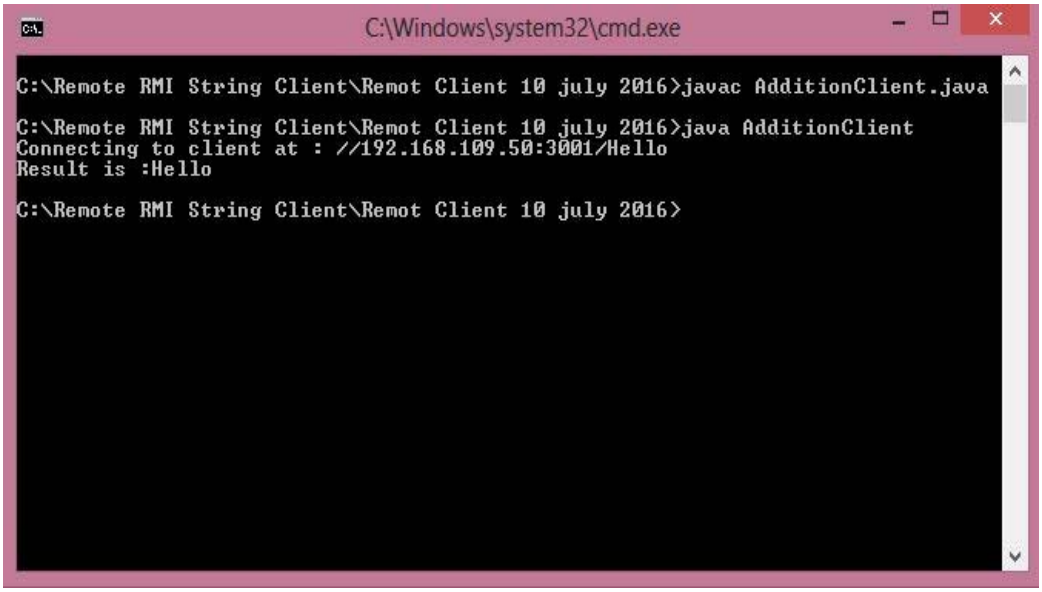

Figure 3.12: Create class file and run the client

Chapter 4
RMI with Database

Java RMI provides a simpler mechanism to invoke method remote. Here we will discuss the development of Java RMI with database application. This sample application is layered into 3tier: client, RMI server or middleware, and database.

4.1 Setting Local host connection with Database

We create an connection file called demo.java. As we are using the SQL package, we also need to declare java.sql.SQLException in the throws. We use user name= root and password =””. We use local host port 3306.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class demo{
     public static void main(String[] args) {
          try{
          Class.forName("com.mysql.jdbc.Driver");
           Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/rmi","roo
t","");
          if(con!=null)
               System.out.println("connection successful");
     }catch(ClassNotFoundException | SQLException e){
          System.err.println(e);
     }
     }
}
```

4.2 Define the Remote Interfaces

In this project, we are trying to run SQL queries using distributed Java objects. To manipulate a remote server object, client code needs to know what it can do with that object. Therefore, an interface is shared between the client and server. It is this interface that exposes the methods of the remote object to the client.

Create the Remote Interface

Our interface file is called AdditionalInterface.java. In order to create a remote interface. We import the RMI package .we use SQL and also import the SQL package. To expose the remote methods to a client, the interface must extend java.rmi.Remote. Each exposed method must declare java.rmi.RemoteException in its throws.

Define the remote methods in an interface (AdditionalInterface.java)

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

public interface AdditionalInterface extends Remote{
```

Compile and Locate the Interface

We compile the interface file using the javac command:

javac AdditionalInterface.java

This produces a file called AdditionalInterface.class

Figure 4.1: Create class file of Addition and AdditionServer

4.3 Define the Remote Classes

In this step, we create the remote server class. The remote server class implements all of the methods defined by the interface we created above stop by creating the actual Java methods to match the interface.

Create the Remote Server Class

Here we import the java.rmi.server package into the class. In this step ,we define the remote server by creating a class called Addition.class in Addition.java file. In order to define the class as containing methods that can be accessed remotely, we define Addition as a subclass of UnicastRemoteObject, implementing the Additionalinterface.

Remote server class Addition.java

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

public class Addition extends UnicastRemoteObject implements
        AdditionalInterface {
   private static final long serialVersionUID = 1L;

    public  Addition() throws RemoteException {
    }
```

Compile and Locate the Remote Server Class

Compile the remote server class using javac:

javac Addition.java

This produces a file called Addition.class.

With RMI, client program and remote objects use proxy called stub to handle the necessary networking between them. As Stub is a Java class. We place the compiled stub on the client .The RMI server host may cooperate as a client for another RMI server.

Generate the Stub Class

With RMI, client program and remote objects use proxy called stub to handle the necessary networking between them. As Stub is a Java class. We place the compiled stub on the client .The RMI server host may cooperate as a client for another RMI server.

In RMI, a stub is a java class that either resides on the client machine. We create stubs by using rmic (RMI compiler), which ships with the JDK. Before generating stub we compile the remote interface, AdditionalInterface.java, and the remote server, Addition.java. To generate stubs, we use a tool called the RMI compiler, which is invoked from the command line, as shown here:

rmic Addition

This creates class files

Addition_Stub.class

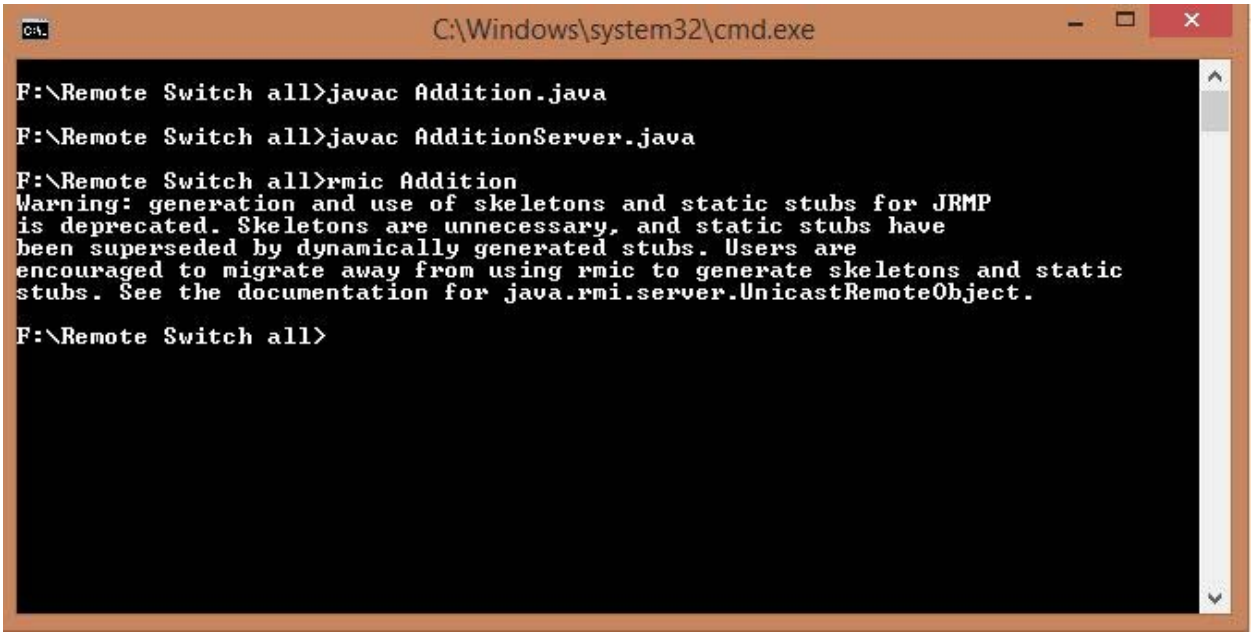

Figure 4.2: rmic Addition

4.4 Creating the Server

Server programs register remote objects with the bootstrap registry service. Once the remote object is registered, server programs can return references to the object. Since RMI is network based, it is essential to use networking terminology .As we know, Networking is based on the notion of hosts, servers, and clients. We can treat each computer as a host. Hosts have names and we use network address as name= "192.168.109.50". The class that contains the remote methods is instantiated in the following line:

```
Addition Hello = new Addition();
```

The object is then registered with the RMI registry service:

```
Naming.bind(bindLocation, Hello);
```

This is all we need to do to set up the remote factory to instantiate remote objects; RMI does everything else.

Start the RMI Registry Service on the Server

To access a remote object, a client program first gets a reference to the object by using the Naming class to look up the object using its registered name. Set your environment to point to the location of your Java installation.The Windows equivalent would be C:\ProgramFiles\Java\ jdk1.8.0_71 before starting registry, we should set class path.

Start the registry

start rmiregistry 3001

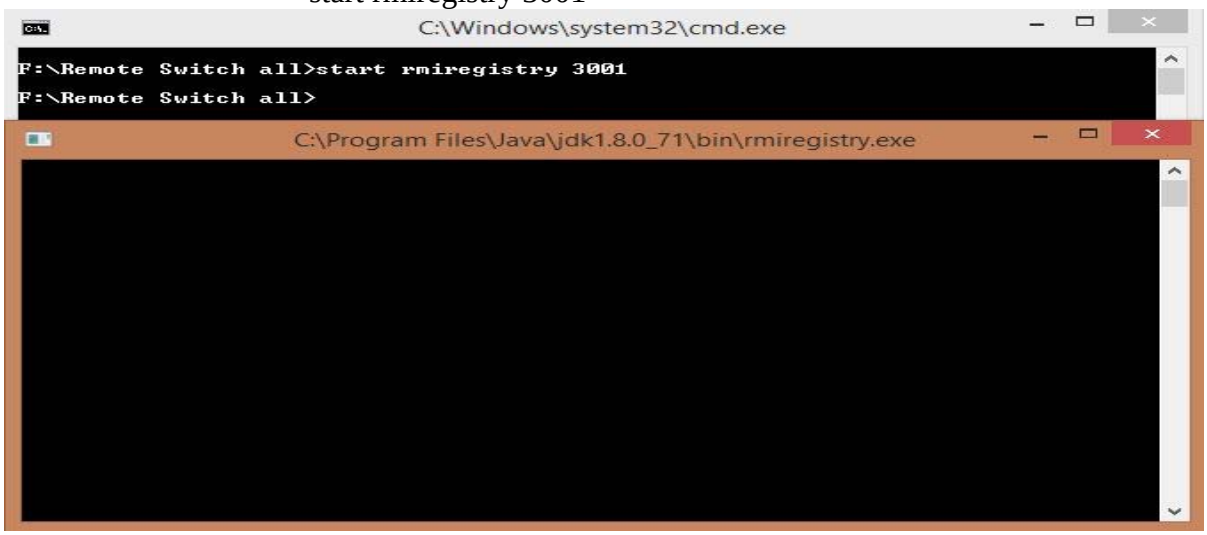

Figure 4.3: start rmiregistry

Start the Server

After setting class path and start the registry which creates an object that implements the methods accessed remotely by the client. We can start the server program as follows:

java AdditionServer

Figure 4.4: Execution of the server

4.5 Create the Java Client Program

Client programs use remote objects that have been exported by remote server programs. In order to do this, the client program must look up the remote object in the remote RMI registry. When the remote object is located, the stub of the remote object is sent to the client. The client invokes the methods in this stub as if the stub were the actual remote object in the local Java Virtual Machine.

Obtaining a Reference to the Remote Object

The client uses the `Naming.lookup` method to obtain a reference to the remote object. The returned value is actually a reference to a stub object.

```
hello=(AdditionalInterface) Naming.lookup(connectLocation);
```

Output of the project

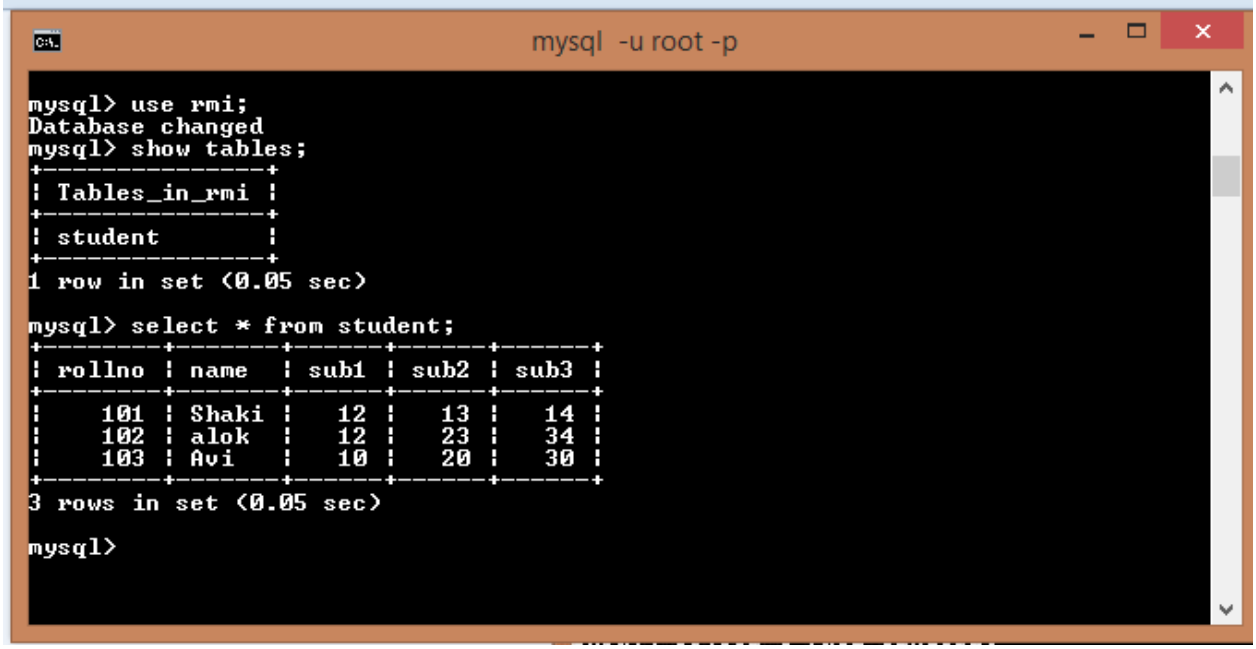

Figure 4.5: Database of Students

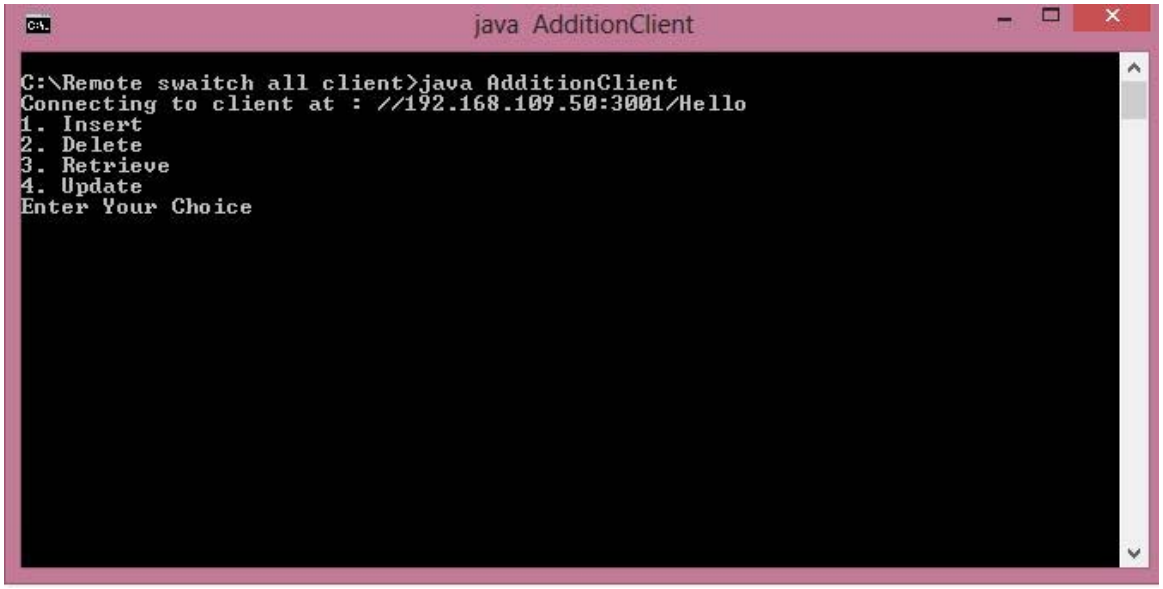

Figure 4.6: Execution of client

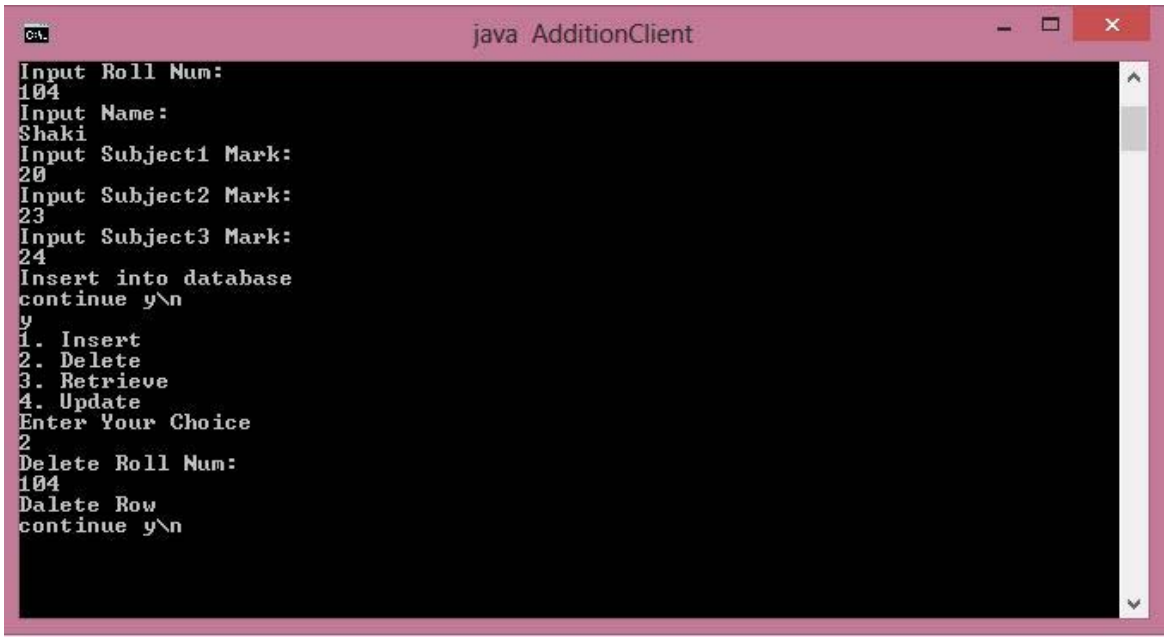

Figure 4.7: Option choice window

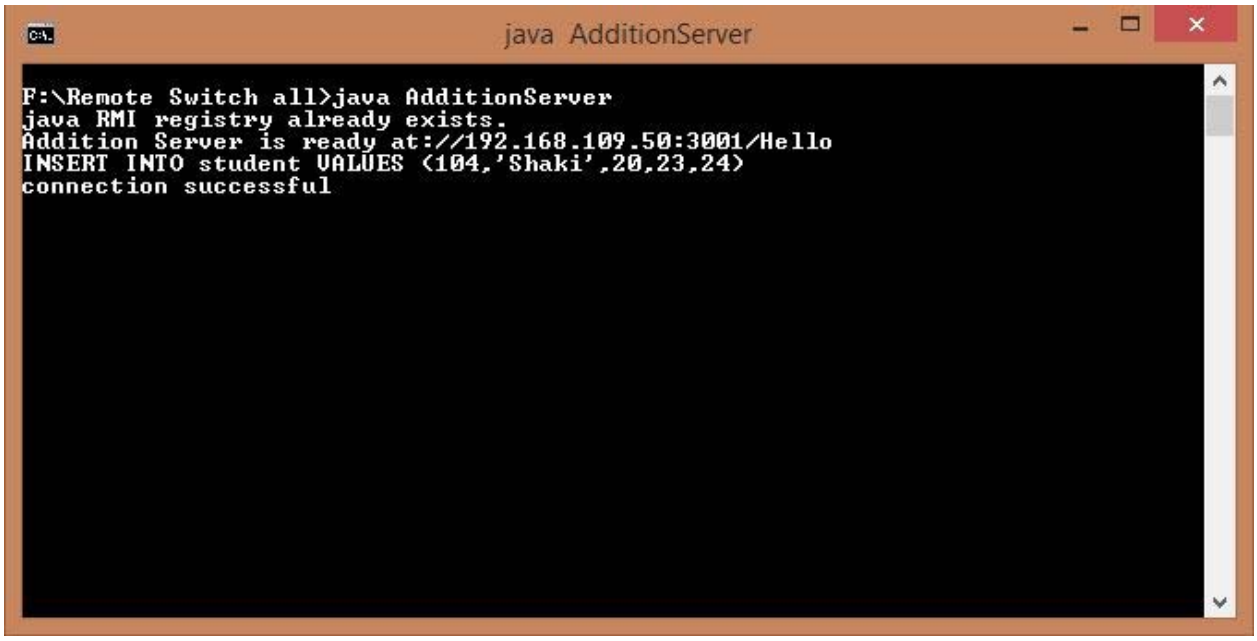

Figure 4.8: Add new record into database

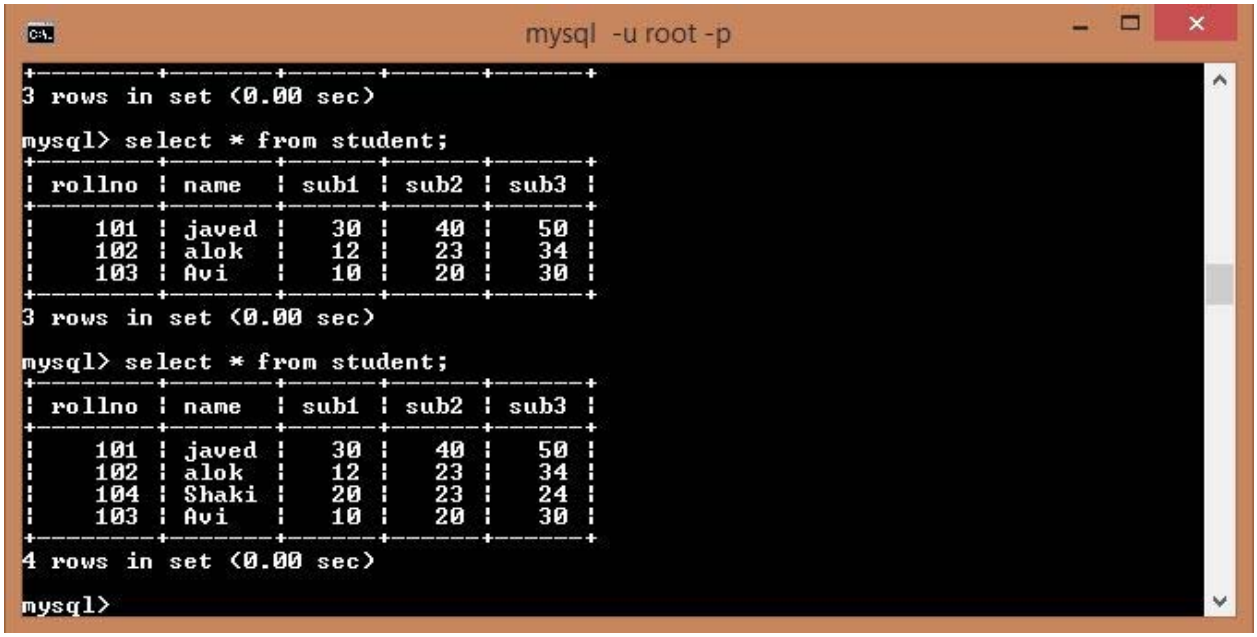

Figure 4.9: Inserted database

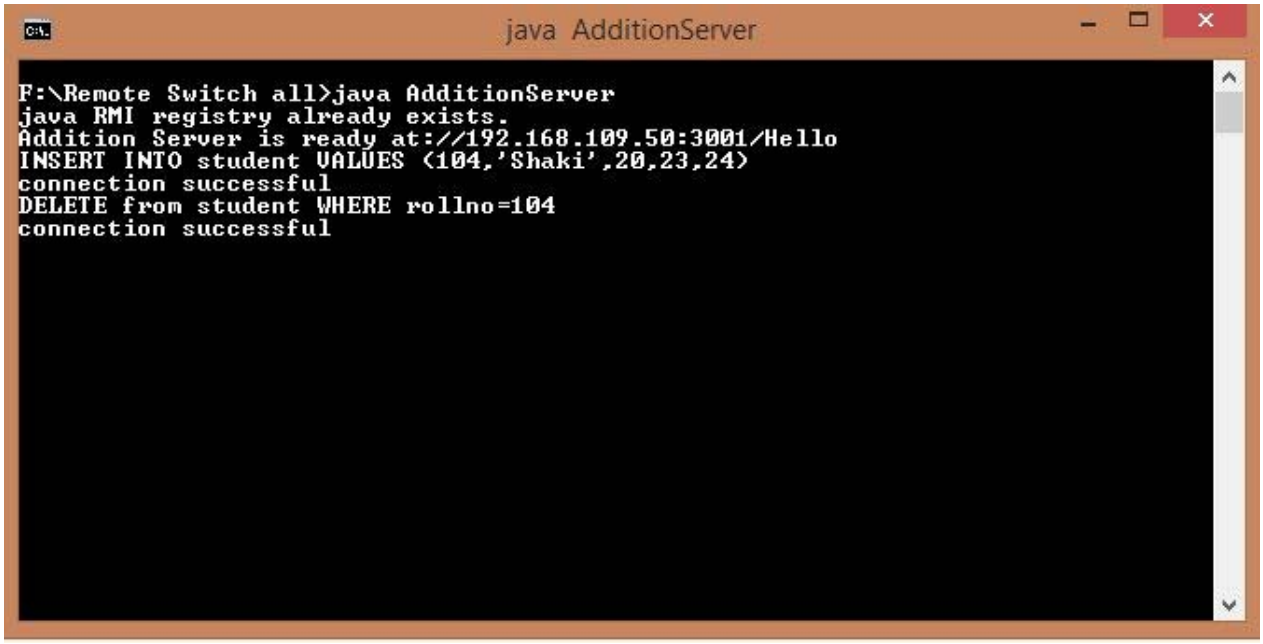

Figure 4.10: Command of delete a record from database

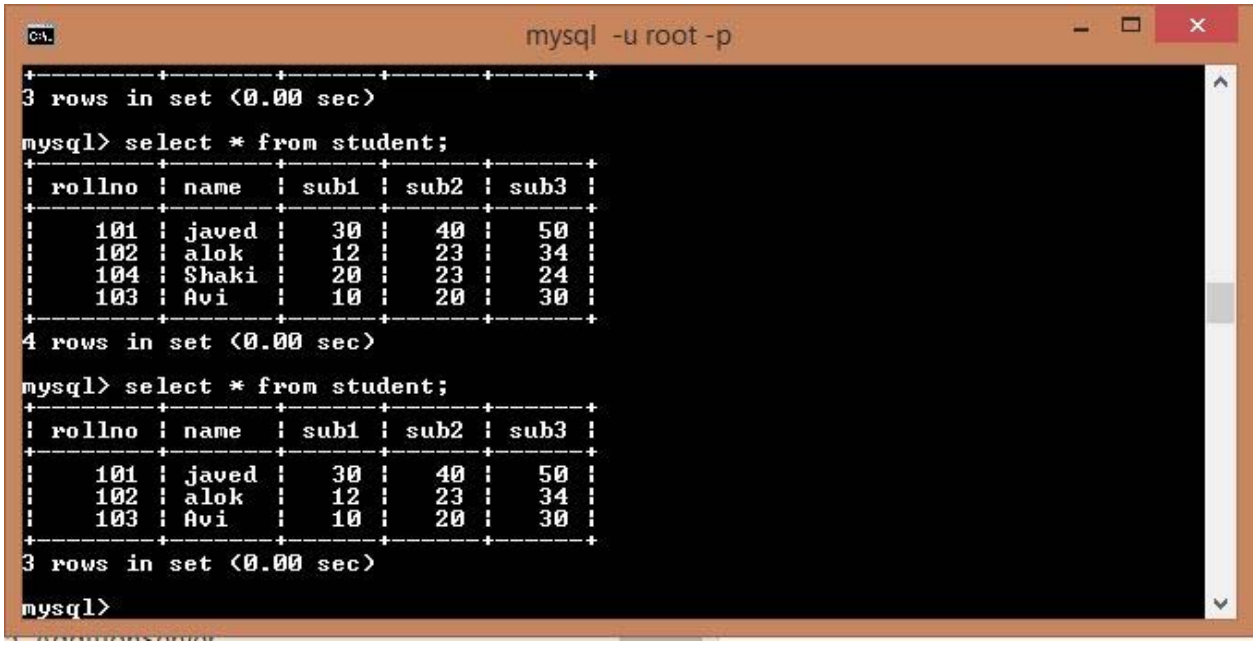

Figure 4.11: deleted Database

Figure 4.12: Choosing retrieve option

Figure 4.13: Command of retrieve a record from database

Figure 4.14: Retrieved result

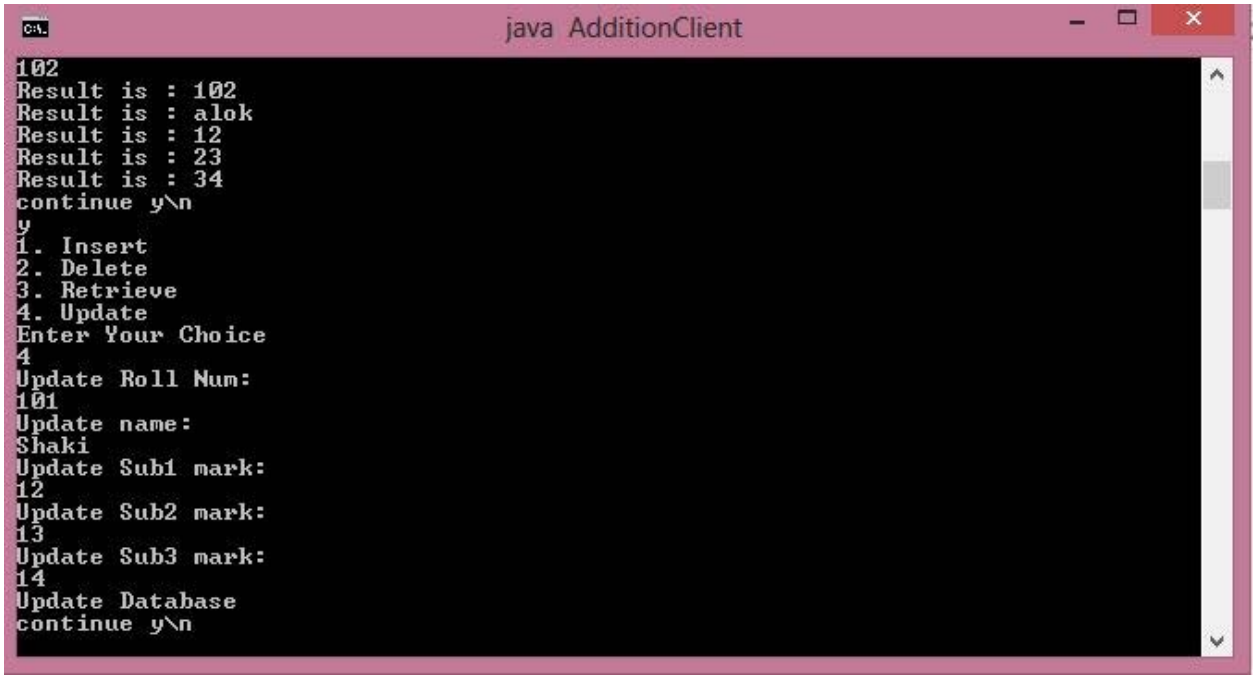

Figure 4.15: Choosing Update option

Figure 4.16: Command of update a record into database

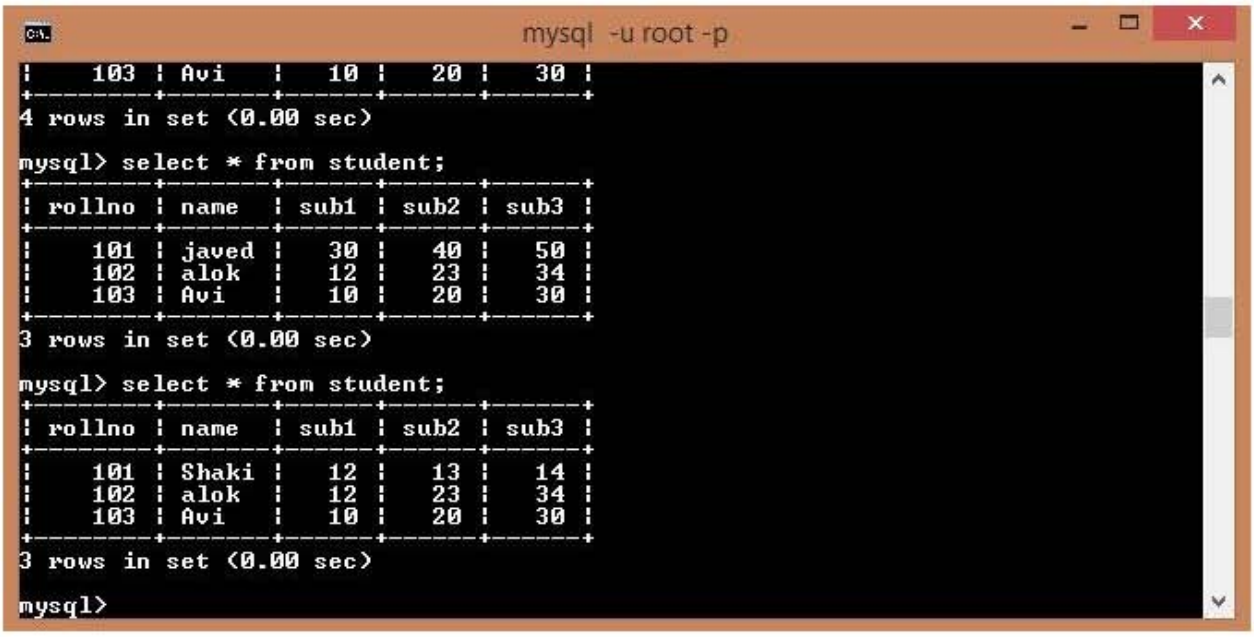

Figure 4.17: Updated databases

Chapter 5
Conclusion and Future Work

5.1 Conclusion

Java RMI has been introduced to reduce the complexity in developing protocol that relies on UDP and TCP. RMI allow programmers to develop distributed Java programs with the same syntax and semantic used for non-distributed program. RMI provides a simpler mechanism to invoke method remotely. Here we discussed the development of Java RMI with database application. RMI gives platform to expand Java into any part system in an incremental fashion, adding new Java servers and clients when it makes sense. After addition of Java, its full benefits flow through all the Java in system. RMI makes this easy, secure, and powerful.

5.2 Future Work

The future plan of this project is to improved design, implementation and documentation in such a way that anyone can use this project for better. Future work may include developing the application dynamically so that Client can access this application from anywhere. RMI based distributed query processing system will be enhanced with a web application. We will make a distributed web application which can run from any device. In future we will be add a web application on server and the user access this application through different type of devices like computer, mobile phone.

Appendix

Code for RMI Implementation with Database

AdditionalInterface.java

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

public interface AdditionalInterface extends Remote {
    public String[] Add(String str) throws RemoteException;
public String[] Add1(String str) throws RemoteException;
public String[] Add2(String str) throws RemoteException;
public String[] Add3(String str) throws RemoteException;
}
```

Addition.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.*;
import java.sql.*;
import java.rmi.Remote;
import java.rmi.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Addition extends UnicastRemoteObject implements
        AdditionalInterface {
   private static final long serialVersionUID = 1L;


    public  Addition() throws RemoteException {
    }
    public String[] Add(String str) {
        System.out.println(str);
         ResultSet rs;
        Integer tot_rows = 0 ;
 String str1[]= new String[6];
try
        {
            java.sql.Connection rmiconn=null;
            Class.forName("com.mysql.jdbc.Driver");
            rmiconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/rmi","root","
");
```

```
               if(rmiconn!=null)
                 System.out.println("connection successful");
             Statement st=rmiconn.createStatement();
             st.executeUpdate(str);
            st.execute("commit");
        }
        catch (Exception e)
        {
          System.out.println("Not executed");
            System.out.println(e);
            }
return(str1);
}
    public String[] Add1(String str) {
        System.out.println(str);
         ResultSet rs;
        Integer tot_rows = 0 ;
             String str1[]= new String[6];
                       try
            {
                java.sql.Connection rmiconn=null;
                Class.forName("com.mysql.jdbc.Driver");
                rmiconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/rmi","root","
");
               if(rmiconn!=null)
                System.out.println("connection successful");
                Statement st=rmiconn.createStatement();
                st.executeUpdate(str);
            }
            catch (Exception e)
            {
                System.out.println(e);
            }
     return(str1);
    }
public String[] Add2(String str) {

        System.out.println(str);
         ResultSet rs;
        Integer tot_rows = 0 ;
String str1[]= new String[6];
        try
        {
            java.sql.Connection rmiconn=null;
            Class.forName("com.mysql.jdbc.Driver");
            rmiconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/rmi","root","
");
              if(rmiconn!=null)
                System.out.println("connection successful");
            Statement st=rmiconn.createStatement();
```

```
            rs=st.executeQuery(str);
            if(rs.next())
            {
                    str1[0]=Integer.toString(rs.getInt("rollno"));
                    str1[1]=rs.getString("name");
                    str1[2]=Integer.toString(rs.getInt("sub1"));
                    str1[3]=Integer.toString(rs.getInt("sub2"));
                    str1[4]=Integer.toString(rs.getInt("sub3"));
            }
            else
            {
                str1 = null;
            }

        }
        catch (Exception e)
        {
            System.out.println(e);
        }
        return(str1);
    }
   public String[] Add3(String str) {

        System.out.println(str);
         ResultSet rs;
        Integer tot_rows = 0 ;
 String str1[]= new String[6];
            try
            {
                java.sql.Connection rmiconn=null;
                Class.forName("com.mysql.jdbc.Driver");
                rmiconn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/rmi","root","
");
            if(rmiconn!=null)
                 System.out.println("connection successful");
                Statement st=rmiconn.createStatement();
                st.executeUpdate(str);

            }
            catch (Exception e)
            {
                System.out.println(e);
            }
       return(str1);
}
}
```

AdditionServer.java

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;

public class AdditionServer {
    public static void main(String[] argv) throws RemoteException {
        Addition Hello = new Addition();
        int port = 3001;
        try {
            LocateRegistry.createRegistry(port);
            System.out.println("java RMI registry created.");
        } catch (RemoteException e) {
            System.out.println("java RMI registry already exists.");
        }
        String hostname = "192.168.109.50";
        String bindLocation = "//" + hostname + ":" + port + "/Hello";
        try {
            Naming.bind(bindLocation, Hello);
            System.out.println("Addition Server is ready at:" +
bindLocation);
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (Exception e) {
            System.out.println("Addition Serverfailed: " + e);
        }
    }
}
```

AdditionClient.java

```
import java.net.MalformedURLException;
import java.rmi.*;
import java.io.*;
import java.util.*;
import static java.lang.System.exit;
import java.util.Arrays;
import java.util.Scanner;
public class AdditionClient {
    public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
        String remoteHostName = "192.168.109.50";
        int remotePort = 3001;
        String connectLocation = "//" + remoteHostName + ":" +
remotePort
                + "/Hello";
        AdditionalInterface hello = null;
```

```
        try {
            System.out.println("Connecting to client at : " +
connectLocation);
            hello=(AdditionalInterface)
Naming.lookup(connectLocation);
        } catch (MalformedURLException e1) {
            e1.printStackTrace();
        } catch (RemoteException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
        } catch (NotBoundException e1) {
            e1.printStackTrace();
        }
int  x,y;
int ch;
String choice="y";
        String[] result=new String[10];
             Scanner sc=new Scanner(System.in);
    try {
            do
         {

                System.out.println("1. Insert");
           System.out.println("2. Delete");
                System.out.println("3. Retrieve");
                System.out.println("4. Update");
                System.out.println("Enter Your Choice");
                ch=sc.nextInt();
       switch(ch)
                {
                    case 1:
                    System.out.println("Input Roll Num:");
                    int rollno = scanner.nextInt();
                    System.out.println("Input Name:");
                    String name = scanner.next();
                    System.out.println("Input Subject1 Mark:");
                    int sub1 = scanner.nextInt();
                    System.out.println("Input Subject2 Mark:");
                    int sub2 = scanner.nextInt();
                    System.out.println("Input Subject3 Mark:");
                    int sub3 = scanner.nextInt();
                    result= hello.Add("INSERT INTO student VALUES
("+rollno+",'"+name+"',"+sub1+","+sub2+","+sub3+")");
           System.out.println("Insert into database");
                      break;
                     case 2:
                         System.out.println("Delete Roll Num:");
                         int roll = scanner.nextInt();
           result = hello.Add1("DELETE from student WHERE
rollno="+roll+"");
                 System.out.println("Dalete Row");
                  break;
```

```
                case 3:
                         System.out.println("View all Roll Num:");
                                        int rol = scanner.nextInt();
              result = hello.Add2("SELECT * FROM student Where rollno
="+rol+"");
                      for (String v: result)
                            System.out.println("Result is : " + v);
                         break;
                        case 4:
                         System.out.println("Update Roll Num:");
                                         int ro = scanner.nextInt();
                         System.out.println("Update name:");
                                        String name1 = scanner.next();
                         System.out.println("Update Sub1 mark:");
                                        int su1 = scanner.nextInt();
                         System.out.println("Update Sub2 mark:");
                                        int su2 = scanner.nextInt();
                         System.out.println("Update Sub3 mark:");
                                        int su3 = scanner.nextInt();
             result = hello.Add3("UPDATE student SET
name='"+name1+"',sub1="+su1+",sub2="+su2+",sub3="+su3+" WHERE
rollno="+ro+" ");
                    System.out.println("Update Database");
                       break;
               default :
                System.out.println("wrong choice");
                break;

                 }
               System.out.println("continue y\\n");
                 choice=sc.next();

           }while(choice.equals("y"));


        } catch (RemoteException e1) {
            e1.printStackTrace();
        }

}
}
```

References

[1] Distributed Systems: Concepts and Design (5th Edition) by George Coulouris , Jean Dollimore , Tim Kindberg

[2] https://www.techopedia.com/definition/1311/remote-method-invocation-rmi.html

[3] http://www.javacoffeebreak.com/articles/javarmi/javarmi.html

[4] http://mrbool.com/how-to-create-rmi-client-and-server-to-invoke-remove-method-of-rmi-server-in-java/28320

[5] https://www.cs.ucsb.edu/~cappello/lectures/rmi/helloworld.shtml

[6] http://docs.oracle.com/javase/1.5.0/docs/guide/rmi/hello/hello-world.html

[7] http://javaexamples-shanavas.blogspot.com/2009/02/rmi-remote-methode-invocation-two.html

[8] https://cseweb.ucsd.edu/classes/wi00/cse130/rmi.html

[9] http://lycog.com/distributed-systems/java-rmi-database-application/1.html

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

[11] http://www.cs.ucsb.edu/~cappello/lectures/rmi/helloworld.shtml

[12] http://stackoverflow.com/questions/21495389/example-of-a-rmi-program-involving-2-machine-connected-via-lan