TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Lab Report

On

Implementation of JAVA RMI Mechanism

Submitted By:

Anuj Rayamajhi (THA076BCT007)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

27th June, 2023

1. INTRODUCTION

This report contains the implementation of Remote Method Invocation mechanism in Java Virtual Machine. RMI method is used to mediate communication between client and server program. In the previous lab work, the communication between client and server was established using Socket programming.

Some of the reasons to used RMI method could be:

- It enables distributed computing by providing a transparent way for objects to communicate and invoke methods across different JVMs.
- It provides a registry service that allows clients to look up remote objects by their names and obtain a reference to the remote object to invoke its methods.
- It supports both synchronous and asynchronous communication between client and server, allowing for flexibility in designing distributed applications.
- It uses Java's serialization mechanism to marshal and unmarshal objects and method parameters, allowing them to be transferred across the network.
- It provides a high-level, object-oriented approach to remote communication, where objects on the client-side can invoke methods on remote objects as if they were local objects.

There are also some downs using RMI method, which will not be discussed in this report.

2. OBJECTIVES

- To use RMI mechanism in client-server architecture using JVM
- To create remote methods to modify and view database records

3. THEORY

RMI (Remote Method Invocation) is a mechanism in Java that enables communication between a client and a server by allowing the client to invoke methods on objects residing in the server's Java Virtual Machine (JVM). RMI provides a transparent and object-oriented approach to remote communication, making it easier for developers to build distributed applications.

The role of RMI in client-server communication can be summarized as follows:

- **Object Remote Interfaces:** RMI requires the definition of interfaces that extend the ‘`java.rmi.Remote`’ interface. These interfaces specify the methods that can be invoked remotely by clients on server-side objects. Remote interfaces act as contracts between the client and server, ensuring that both parties understand the available methods and their signatures.
- **Remote Object Implementation:** On the server side, objects are created that implement the remote interfaces defined in step 1. These objects are often referred to as remote objects or remote servants. They expose methods that can be invoked remotely by clients.
- **RMI Registry:** The RMI registry provides a centralized lookup service for clients to locate remote objects. The server registers its remote objects with the registry, associating them with a unique name. Clients can then query the registry by name to obtain a reference to the desired remote object.
- **Marshaling and Unmarshaling:** When a client invokes a remote method, RMI handles the marshaling (serialization) of the method parameters and the unmarshaling (deserialization) of the return value. This allows objects and data to be transferred between the client and server JVMs.

- **Stub and Skeleton:** RMI generates stub and skeleton classes for remote objects. The stub on the client side acts as a proxy for the remote object and intercepts method invocations, handling the network communication. The skeleton on the server side receives method invocations from the client stub, invokes the corresponding methods on the actual remote object, and returns the result to the client.
- **Remote Method Invocation:** Clients invoke remote methods on the stub objects as if they were invoking local methods. The stub handles the communication details, including establishing a network connection with the server, marshaling method parameters, and sending the request. The server's skeleton receives the request, unmarshals the parameters, invokes the corresponding method on the remote object, and sends back the result.
- **Exception Handling:** RMI supports exception handling between the client and server. When an exception occurs during a remote method invocation, it is marshaled and propagated back to the client, allowing for proper error handling.

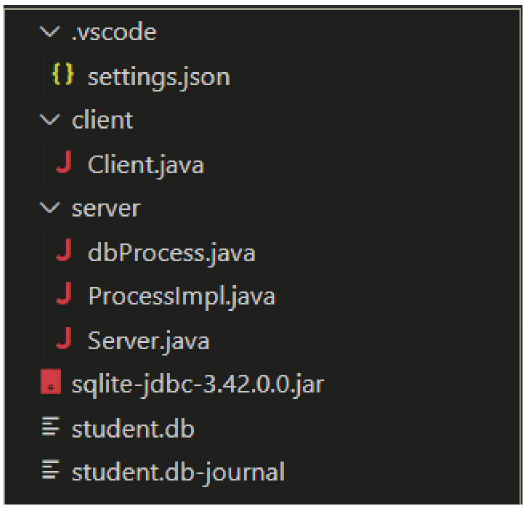
Through these steps, RMI enables seamless communication between client and server, abstracting away the complexities of network programming and providing a familiar object-oriented programming paradigm for distributed applications.

JDBC (Java Database Connectivity) is a Java API that provides a standardized way to interact with relational databases. It allows Java applications to connect to and manipulate databases using SQL (Structured Query Language) statements. JDBC provides a set of interfaces and classes that enable developers to perform various database operations such as establishing connections, executing queries, retrieving results, and managing transactions.

4. CODE

This section contains the code, that successfully realizes the objective of this lab work.

4.1 File Structure overview

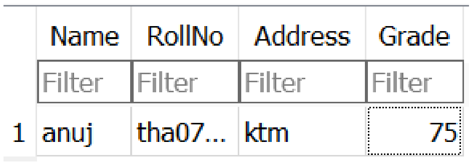


```
.vscode
  settings.json
client
  Client.java
server
  dbProcess.java
  ProcessImpl.java
  Server.java
sqlite-jdbc-3.42.0.0.jar
student.db
student.db-journal
```

Figure 1 : Code Filing structure

From the above figure, it can be seen, Client and Server are created as two separate JVMs. And the communication is achieved between 'client' and 'server' java projects, which is mediated through interface code: *'dbProcess.java'* and interface implementation code: *'ProcessImpl.java'*.

SQLite is used to create database *'student.db'*, which contains a table named *'student'* with columns: Name, RollNo, Address and Grade. Client program is able to invoke the methods to add and view records in this database available in *'ProcessImpl.java'* through *'dbProcess.java'*.



	Name	RollNo	Address	Grade
	Filter	Filter	Filter	Filter
1	anuj	tha07...	ktm	75

Figure 2 : Student Table

4.2 dbProcess.java

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

public interface dbProcess extends Remote {
   // Defining behaviors/methods
   // writing methods

   public void addRecord(String name, String address, String roll,
int grade) throws RemoteException;

   // reading methods

   public String getRecord(String roll) throws RemoteException;

}
```

4.3 ProcessImpl.java

```java
package server;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
// import java.sql.Statement;


public class ProcessImpl extends UnicastRemoteObject implements
dbProcess {

   protected ProcessImpl() throws RemoteException{
      super();
      establishConnection();
   }
   private static final long serialVersionUID = 1L;
   private Connection connection = null;

   private void establishConnection() {
      try {
         Class.forName("org.sqlite.JDBC");

         String dbFile = ".\\student.db";

         connection =
DriverManager.getConnection("jdbc:sqlite:"+dbFile);

         System.out.println("Connection Established to the
Database");

      }
      catch(ClassNotFoundException | SQLException e) {
         System.out.println("Failed to Establish Database
Connection: " + e.getMessage() );
      }
   }

   // @Override
   public void addRecord(String name, String address, String roll,
int grade) throws RemoteException {
      try {

         PreparedStatement prepStatement = null;
         // Statement statement = connection.createStatement();
         String query = "INSERT INTO student (Name, RollNo, Address,
Grade) VALUES (?,?,?,?)";
         prepStatement = connection.prepareStatement(query);

         prepStatement.setString(1,name);
         prepStatement.setString(2, roll);
         prepStatement.setString(3, address);
         prepStatement.setInt(4, grade);

         prepStatement.executeUpdate();

         System.out.println("Record Successfully Added.");

      } catch (SQLException e) {

         System.out.println("Failed to add into database:
"+e.getMessage());

      }
   }

   // @Override
   public String getRecord(String roll) throws RemoteException {
      String result = "";
      try {

         PreparedStatement prepStatement = null;
         // Statement statement = connection.createStatement();
         String query = "SELECT * FROM student WHERE RollNo = ?";
         prepStatement = connection.prepareStatement(query);
         prepStatement.setString(1, roll);
         ResultSet resultSet = prepStatement.executeQuery();

         while(resultSet.next()){
            String name = resultSet.getString("Name");
            String address = resultSet.getString("Address");
            int grade = resultSet.getInt("Grade");

            result += "Name: " + name + ", Roll No: " + roll + ", 
Address: " + address + ", Grade: " + grade + "\n";
         }
      } catch (SQLException e) {
         System.out.println("Failed to read data from the database: 
"+e.getMessage());
      }
      return result;
   }
}
```

4.4 Server.java

```
package server;
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class Server {
   public static void main(String[] args) {
      try {
         int port = 1099;

         dbProcess proc = new ProcessImpl();

         LocateRegistry.createRegistry(port);

         Naming.rebind("rmi://localhost:"+port+"/dbProcess", proc);

         System.out.println("Server is running and waiting for 
client requests");
```

```
      } catch (Exception e) {
         System.out.println("Server Exception: "+ e.getMessage());
      }
   }
}
```

4.5 Client.java

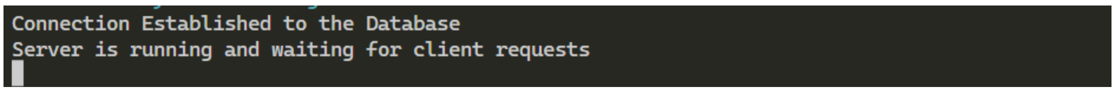
```
import java.rmi.Naming;
import java.util.Scanner;
import server.dbProcess;

public class Client {
   public static void main(String[] args) {
      try {
         int port = 1099;

         dbProcess proc = (dbProcess)
Naming.lookup("rmi://localhost:"+port+"/dbProcess");

         Scanner sc = new Scanner(System.in);

         boolean breakConnection = false;
         String name = "";
         String roll = "";
         String address = "";
         int grade = 0;
         int choice;
         String result;

         System.out.println("WELCOME TO Student Information
System");
         System.out.println("1. Add Record");
         System.out.println("2. Get Record");
         System.out.println("3. Exit");
         while(!breakConnection) {
            System.out.print("Enter Choice(1/2/3): ");
            choice = sc.nextInt();
            sc.nextLine();
            switch(choice){
               case 1:
               try {

                  System.out.print("Enter name: ");
                  name=sc.nextLine();
                  System.out.print("Enter Roll Number: ");
```

```
                    roll=sc.nextLine();
                    System.out.print("Enter Address: ");
                    address = sc.nextLine();
                    System.out.print("Enter Grade: ");
                    grade = sc.nextInt();
                    sc.nextLine();
                    proc.addRecord(name, address, roll, grade);
                    System.out.println("Successfully added to the 
database.");

                } catch (Exception e) {
                    System.out.println("Error while adding data: 
"+e.getMessage());
                }
                break;

                case 2:
                System.out.print("Enter Roll no. : ");
                roll = sc.nextLine();
                result = proc.getRecord(roll);
                System.out.println(result);
                break;

                case 3:
                breakConnection = true;
                break;

                default:
                System.out.println("Enter a valid choice.");
                break;
             }
          }

          sc.close();
          System.out.println("Client Program closed.");

       } catch (Exception e) {
          System.out.println("Client Exception: "+e.getMessage());
       }
    }
}
```

5. OUTPUT

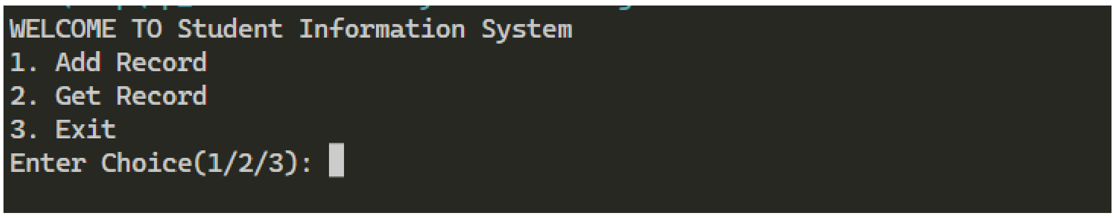
At first, the Server project is executed and then only Client project is executed.

Initially when server is executed:



```
Connection Established to the Database
Server is running and waiting for client requests
```

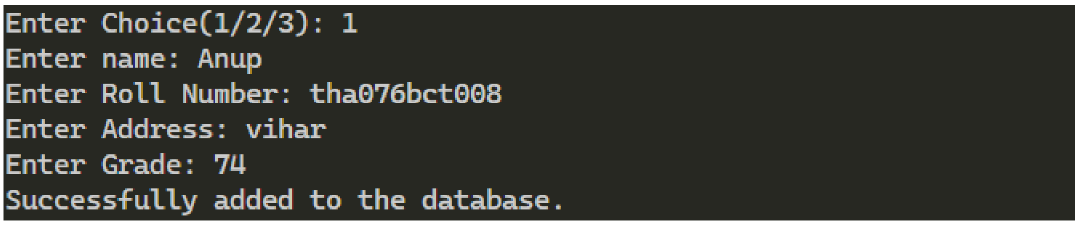
When Client is executed:



```
WELCOME TO Student Information System
1. Add Record
2. Get Record
3. Exit
Enter Choice(1/2/3):
```

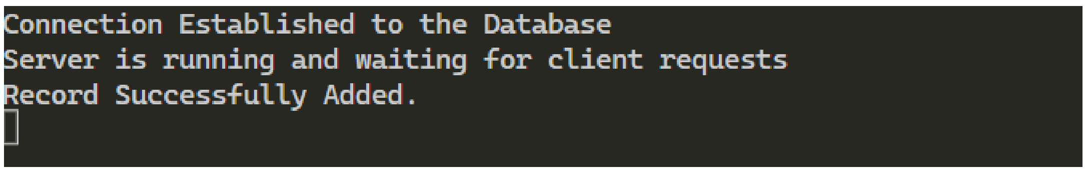
After adding some records in the database through client:

In Client console:



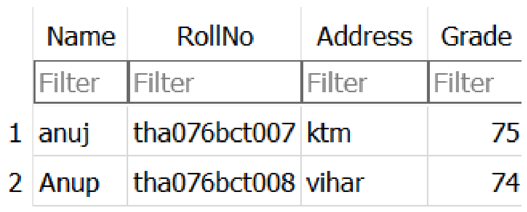
```
Enter Choice(1/2/3): 1
Enter name: Anup
Enter Roll Number: tha076bct008
Enter Address: vihar
Enter Grade: 74
Successfully added to the database.
```

In Server console:



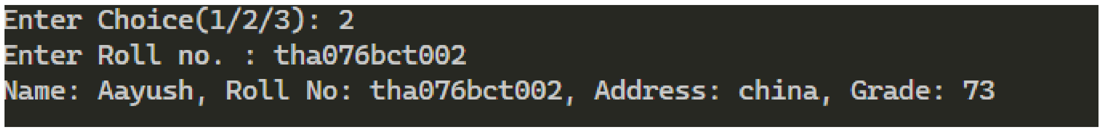
```
Connection Established to the Database
Server is running and waiting for client requests
Record Successfully Added.
```

In Database:



	Name	RollNo	Address	Grade
	Filter	Filter	Filter	Filter
1	anuj	tha076bct007	ktm	75
2	Anup	tha076bct008	vihar	74

After adding some more data and reading from the database through client program:



```
Enter Choice(1/2/3): 2
Enter Roll no. : tha076bct002
Name: Aayush, Roll No: tha076bct002, Address: china, Grade: 73
```

6. CONCLUSION

The objective of this project was to implement a client-server architecture using RMI (Remote Method Invocation). The achieved objectives include the successful development of a server component that exposes remote methods for adding data to a SQLite database and retrieving data from the database. The client component interacts with the server by invoking these remote methods, allowing clients to add records to the database and retrieve specific records based on criteria such as roll number. RMI facilitates seamless communication between the client and server, abstracting away the complexities of networking and providing a convenient and object-oriented approach to remote communication. Overall, the project has successfully demonstrated the use of RMI in a client-server architecture for database operations, providing a scalable and efficient solution for distributed applications

Github link: *https://github.com/anuj-raymajhi/Distributed-System*