TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Lab Report Of Distributed System On Implementation of RMI using Java

Submitted By:

Raj Kumar Dhakal
THA076BCT033

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

June, 2023

TITLE: RMI IMPLEMENTATION USING JAVA

THEORY:

RMI (Remote Method Invocation) in Java allows objects located in different Java virtual machines (JVMs) to remotely invoke methods on each other. It facilitates distributed computing and communication between client and server applications.

RMI consists of several key components. The remote interface defines the methods that can be invoked remotely and extends the 'java.rmi.Remote' interface. The remote object implements the remote interface and executes the remotely invoked methods. The remote object either extends 'java.rmi.server.UnicastRemoteObject' or utilizes a custom subclass to enable remote method invocation.

The key components involved in facilitation communication between the client and server applications are:

a) Stub:
   A stub acts as a client-side proxy for the remote object. It resides in the client JVM and is responsible for handling the communication with the remote object located on the server side. The stub provides a local representation of the remote object and allows the client to invoke methods on it as if it were a local object. When a client invokes a method on the stub, the stub marshals the method parameters into a network-transmittable format, sends the request to the server, and awaits the response. After receiving the response, the stub decodes or deserializes the result and sends it back to the client. Essentially, the stub handles the details of network communication, parameter marshalling, and result unmarshalling on behalf of the client

b) Skeleton:
   A skeleton, also known as a server-side proxy, resides on the server side and acts as an intermediary between the stub and the actual remote object. It receives the method invocation requests from the client stub and is responsible for dispatching those requests to the appropriate remote object. The skeleton

marshals the method parameters received from the stub, invokes the corresponding method on the actual remote object, and obtains the result. It then marshals the result into a network-transmittable format and sends it back to the client stub. Similar to the stub, the skeleton handles the details of parameter marshalling and result marshalling, ensuring smooth communication between the client and the server.

ALGORITHM:

Here are the steps involved in using RMI in Java:

Step 1: Define the Remote Interface.

Step 2: Implement the Remote Object.

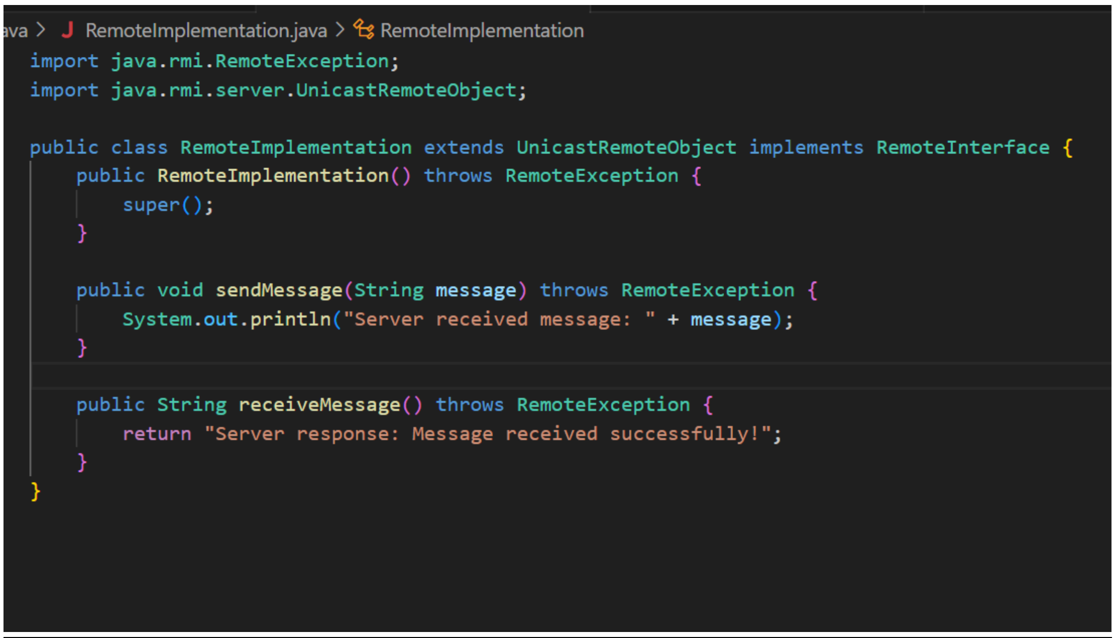
Step 3: Create the Server Application (including binding the remote object to the RMI registry).

Step 4: Create the Client Application (including obtaining a reference to the remote object from the RMI registry).

Step 5: Compile and Run the Applications.

CODE:

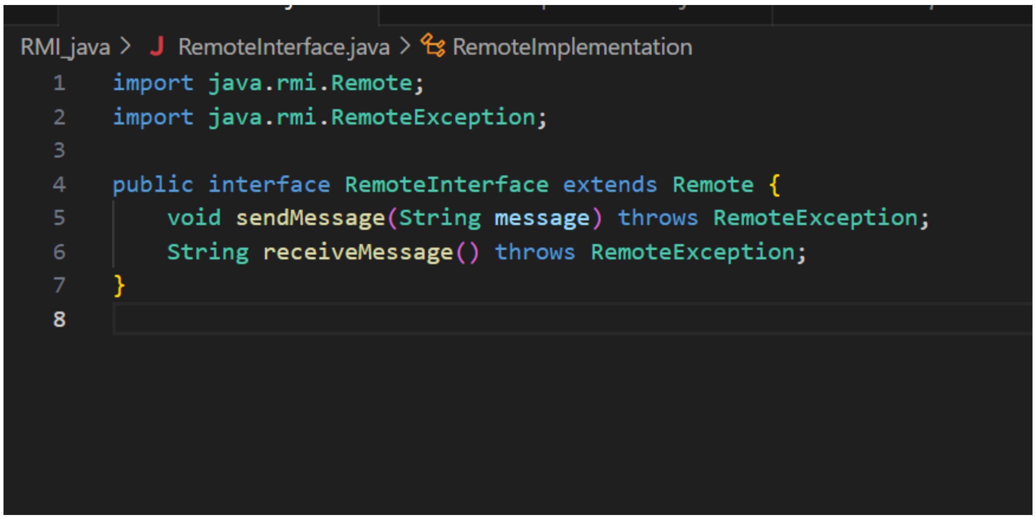
RemoteImplementation.java



ava > J RemoteImplementation.java > RemoteImplementation

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

public class RemoteImplementation extends UnicastRemoteObject implements RemoteInterface {
    public RemoteImplementation() throws RemoteException {
        super();
    }

    public void sendMessage(String message) throws RemoteException {
        System.out.println("Server received message: " + message);
    }

    public String receiveMessage() throws RemoteException {
        return "Server response: Message received successfully!";
    }
}
```

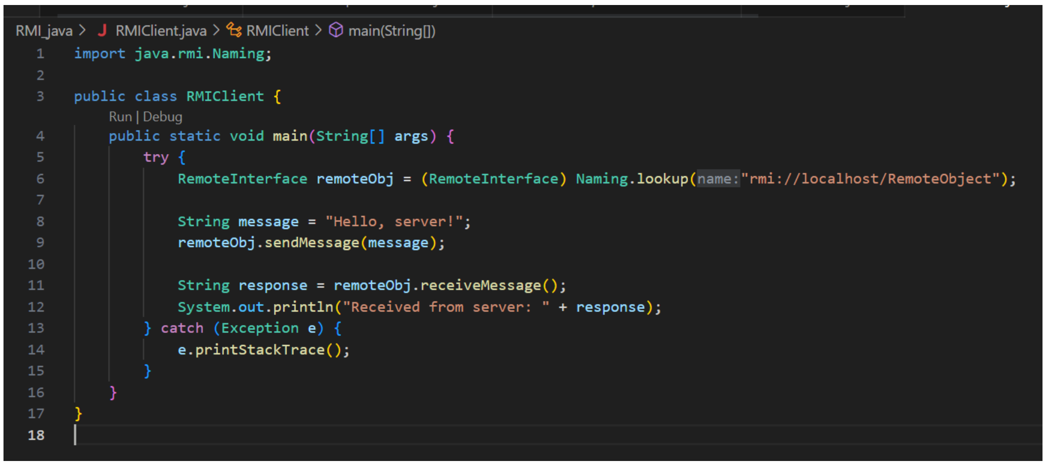
RemoteInterface.java



RMI_java > RemoteInterface.java > RemoteImplementation

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

public interface RemoteInterface extends Remote {
    void sendMessage(String message) throws RemoteException;
    String receiveMessage() throws RemoteException;
}
```

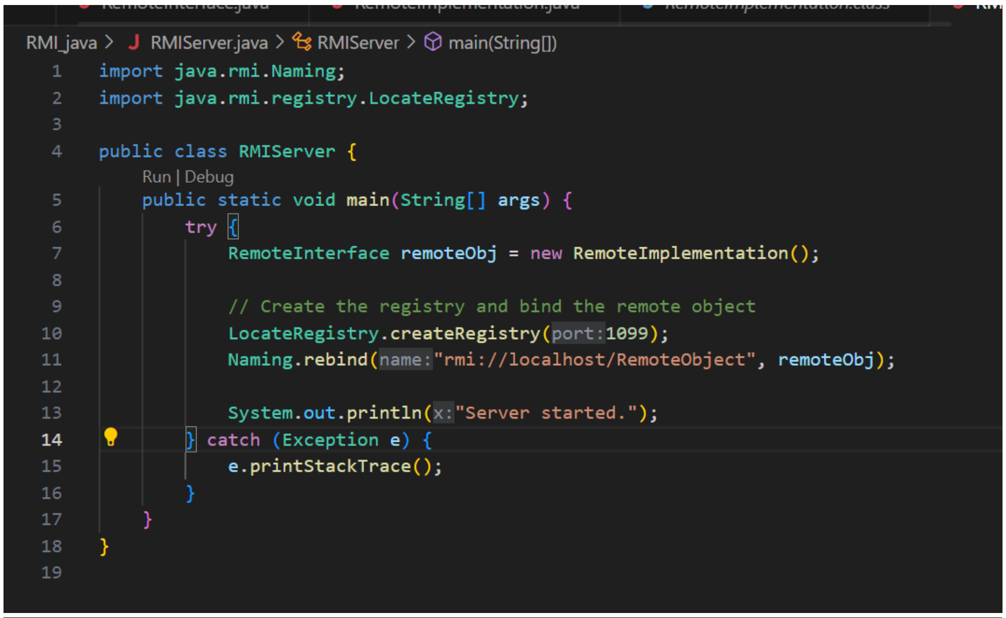
RMIClient.java



RMI_java > RMIClient.java > RMIClient > main(String[])

```java
import java.rmi.Naming;

public class RMIClient {
    Run | Debug
    public static void main(String[] args) {
        try {
            RemoteInterface remoteObj = (RemoteInterface) Naming.lookup(name:"rmi://localhost/RemoteObject");

            String message = "Hello, server!";
            remoteObj.sendMessage(message);

            String response = remoteObj.receiveMessage();
            System.out.println("Received from server: " + response);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

RMIServer.java

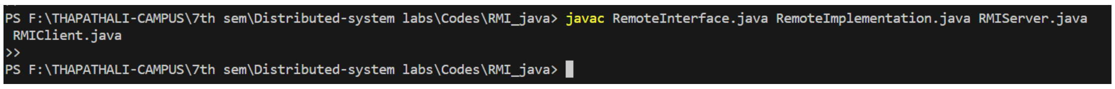


RMI_java > RMIServer.java > RMIServer > main(String[])

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

public class RMIServer {
    Run | Debug
    public static void main(String[] args) {
        try {
            RemoteInterface remoteObj = new RemoteImplementation();

            // Create the registry and bind the remote object
            LocateRegistry.createRegistry(port:1099);
            Naming.rebind(name:"rmi://localhost/RemoteObject", remoteObj);

            System.out.println(x:"Server started.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

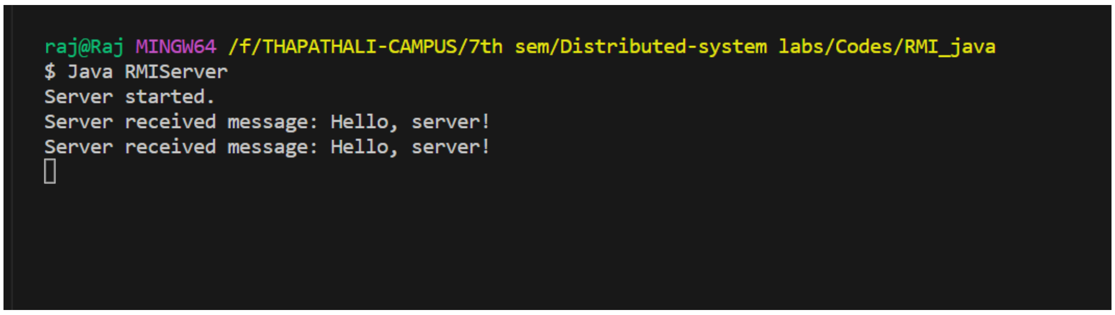
OUTPUT:

Code to compile all four java files.



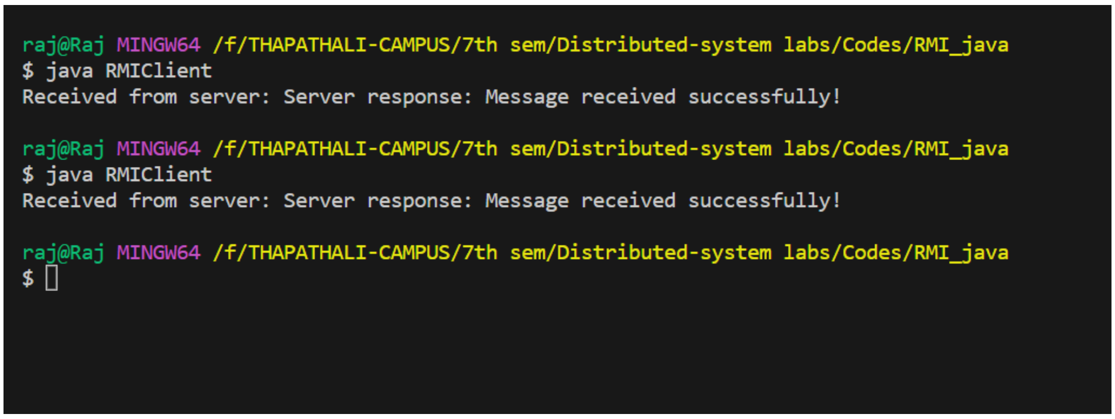
```
PS F:\THAPATHALI-CAMPUS\7th sem\Distributed-system labs\Codes\RMI_java> javac RemoteInterface.java RemoteImplementation.java RMIServer.java
 RMIClient.java
>>
PS F:\THAPATHALI-CAMPUS\7th sem\Distributed-system labs\Codes\RMI_java>
```

SERVER SIDE OUTPUT:



```
raj@Raj MINGW64 /f/THAPATHALI-CAMPUS/7th sem/Distributed-system labs/Codes/RMI_java
$ Java RMIServer
Server started.
Server received message: Hello, server!
Server received message: Hello, server!
```

CLIENT SIDE OUTPUT:



```
raj@Raj MINGW64 /f/THAPATHALI-CAMPUS/7th sem/Distributed-system labs/Codes/RMI_java
$ java RMIClient
Received from server: Server response: Message received successfully!

raj@Raj MINGW64 /f/THAPATHALI-CAMPUS/7th sem/Distributed-system labs/Codes/RMI_java
$ java RMIClient
Received from server: Server response: Message received successfully!

raj@Raj MINGW64 /f/THAPATHALI-CAMPUS/7th sem/Distributed-system labs/Codes/RMI_java
$
```

CONCLUSION:

The RMI implementation lab in Java offered me hands-on experience with remote method invocation. I built a server and client, exchanging messages and I tried to expand it to a chat application. This practical exercise deepened my understanding of RMI's role in distributed systems and robust application development.