TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Lab Report

of

Distributed Systems

on

Implementation of Java RMI mechanism

Submitted by:

Dipesh Bartaula

[THA076BCT015]

Submitted to:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

June 27, 2023

TITLE: TO IMPLEMENT JAVA RMI MECHANISM

THEORY:

The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM. RMI handles the underlying network communication, serialization of objects, and remote method invocation transparently, making it easier for developers to build distributed applications. It provides remote communication between the applications using two objects: stub and skeleton.

Stub:

A stub is a client-side proxy object that acts as a representative or placeholder for the actual remote object residing on the server. When a client makes a method call on the stub, the stub takes care of packaging the method parameters and sending them over the network to the server. It also handles the marshaling (serialization) and unmarshaling (deserialization) of objects between the client and server.

Skeleton:

A skeleton, also known as a server-side stub, resides on the server side and acts as an intermediary between the stub and the actual remote object. When a method call arrives from the client, the skeleton receives the request, unpacks the method parameters, and forwards the call to the remote object. After the remote object processes the method call and returns the result, the skeleton packages the result and sends it back to the client via the network.

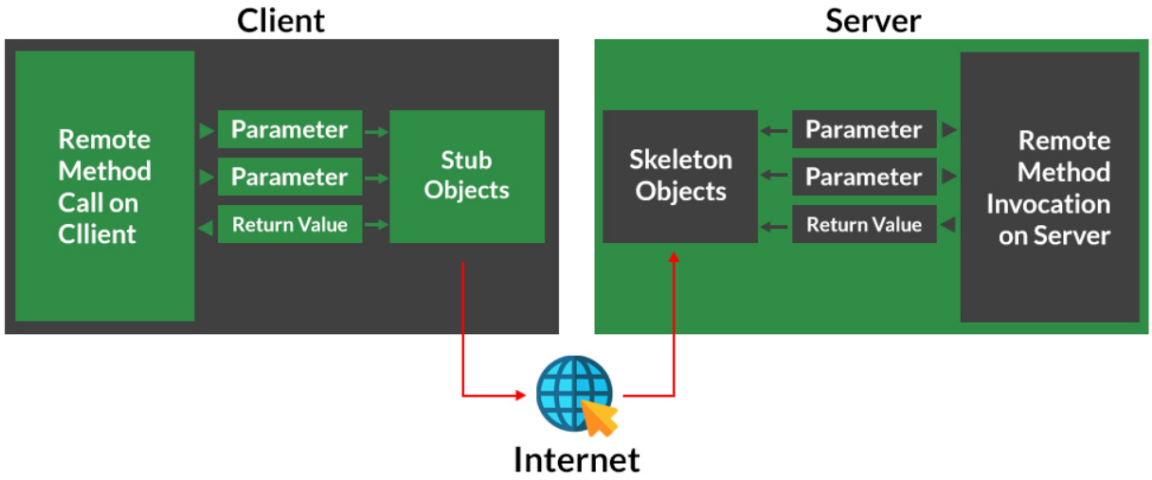

Fig: Working of RMI

Steps involved in using RMI are as follows:

1. Define the remote interface: This interface declares the methods that the client can invoke on the remote object. Both the client and the server must have access to this interface and its implementation.
2. Implement the remote object: The remote object is an implementation of the remote interface. It defines the actual behavior of the methods declared in the interface.
3. Start the RMI registry: The RMI registry is a simple naming service that allows clients to look up remote objects by name. It acts as a central directory for remote objects.
4. Register the remote object with the RMI registry: The server program registers the remote object with a name in the RMI registry, making it available for clients to look up.
5. Run the server program: The server program creates an instance of the remote object, binds it to a specific name in the RMI registry, and starts listening for incoming client requests.
6. Run the client program: The client program looks up the remote object in the RMI registry using its name, obtains a reference to the remote object, and then invokes methods on it as if it were a local object.

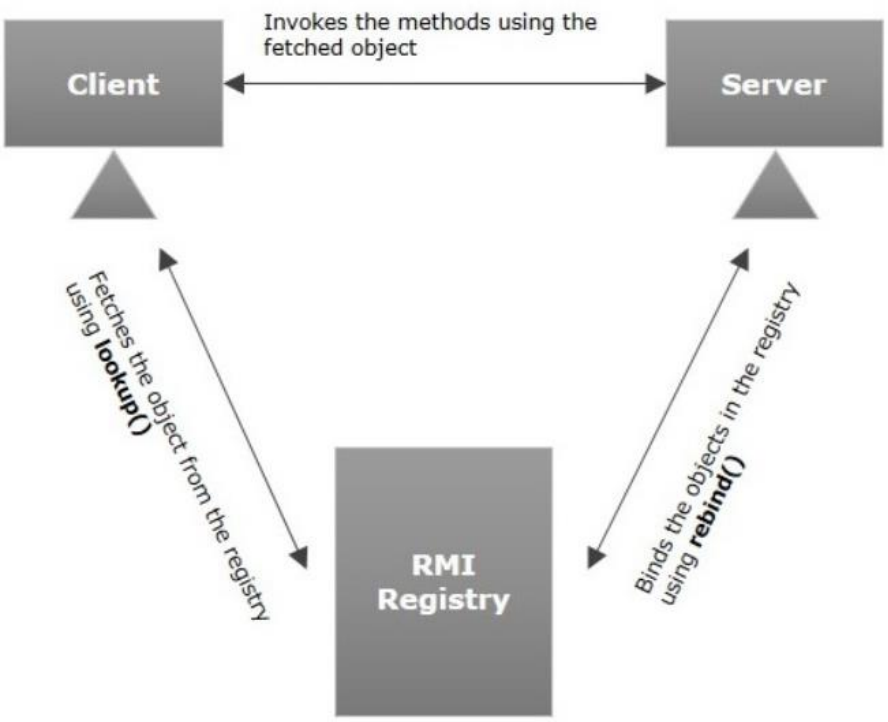

Fig: RMI registry

CODE:

Step 1: Creation of remote interface

RemoteInterface.java > ...

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

public interface RemoteInterface extends Remote {
    int addNumbers(int a, int b) throws RemoteException;
}

```

Step 2: Implementation of remote interface

RemoteImplementation.java > ...

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

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

    @Override
    public int addNumbers(int a, int b) throws RemoteException {
        return a + b;
    }
}
```

Step 3: Server-side class

Server.java > ...

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

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

            // Create and start the RMI registry on port 1099
            LocateRegistry.createRegistry(port:1099);

            // Bind the remote object to the registry with the name "RemoteObject"
            Naming.rebind(name:"RemoteObject", remoteObj);

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

Step 4: Client-side class

```java
import java.rmi.Naming;

public class Client {
    Run | Debug
    public static void main(String[] args) {
        try {
            // Lookup the remote object by its name in the RMI registry
            RemoteInterface remoteObj = (RemoteInterface) Naming.lookup(name:"RemoteObject");

            // Invoke the remote method
            int result = remoteObj.addNumbers(a:10, b:20);
            System.out.println("Result: " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

RESULT:

Step 1: Compiling all the java class and starting the server

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\dell\OneDrive\Desktop\DS program> javac *.java
PS C:\Users\dell\OneDrive\Desktop\DS program> java .\Server.class
Error: Could not find or load main class .\Server.class
Caused by: java.lang.ClassNotFoundException: /\Server/class
PS C:\Users\dell\OneDrive\Desktop\DS program> java Server
Server is ready.
```

Step 2: Running client-side and final result

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\dell\OneDrive\Desktop\DS program> java Client
Result: 30
PS C:\Users\dell\OneDrive\Desktop\DS program>
```

As the program is all about adding the two numbers provided from client which in our case are 10 and 20. Since the program run successfully, we got our final result as 30.

DISCUSSION:

The ‘RemoteInterface’ defines the remote methods, ‘RemoteImplementation’ provides the implementation of those methods, ‘Server’ starts the RMI registry and binds the remote object, and ‘Client’ looks up the remote object and invokes its methods. Together, these classes enable communication between the client and server using the Java RMI mechanism.

CONCLUSION:

By running the server and client programs, we can observe how the client invokes a method on a remote object residing in the server JVM through RMI. This demonstrates the basic functioning of Java RMI for distributed communication and method invocation between Java programs.