Vidyavardhini's College of Engineering & Technology

Department of Computer Engineering

Name: Shivam Pandey
Roll No.: 28
Experiment No.2
Implement Client/Server using RPC/RMI
Date of Performance: 01/02/2024
Date of Submission: 08/02/2024

Aim: To implement Client/Server using RPC/RMI

Objective: Develop a program to implement Client/Server using RPC/RMI

Theory:

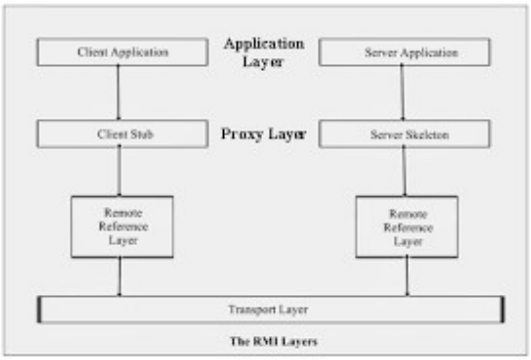
The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. It allows an object to invoke methods on an object running in another JVM. It provides remote communication between the applications using two objects *stub* and *skeleton*.

Understanding stub and skeleton:
RMI uses stub and skeleton object for communication with the remote object.
A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:
Stub
The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

Skeleton
The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.



Steps to write the RMI program

6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

Code and output:

SampleClient.java:-

```
import java.rmi.*;
import java.rmi.server.*;
public class SampleClient
{
    public static void main(String[]  args)
    {
       // set the security manager for the client
       System.setSecurityManager(new RMISecurityManager());
       //get the remote object from the registry
       try
         {
          System.out.println("Security Manager loaded");
          String url = "//localhost/SAMPLE-SERVER";
          SampleServer remoteObject = (SampleServer)Naming.lookup(url);
          System.out.println("Got remote object");
          System.out.println(" 1 + 2 = " + remoteObject.sum(1,2) );
         }
```

```
        catch (RemoteException exc) {
          System.out.println("Error in lookup: " + exc.toString()); }
        catch (java.net.MalformedURLException exc) {
          System.out.println("Malformed URL: " + exc.toString());   }
        catch (java.rmi.NotBoundException exc)  {
          System.out.println("NotBound: " + exc.toString());
        }
    }
}
```

SampleServerImpl.java:-

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class SampleServerImpl extends UnicastRemoteObject implements SampleServer
{
 SampleServerImpl() throws RemoteException
 {


super();
 }

        public int sum(int a,int b) throws RemoteException
        {
                return (a+b);
        }

 public static void main(String args[])
 {
    try
    {
     System.setSecurityManager(new RMISecurityManager());
     //set the security manager

     //create a local instance of the object
     SampleServerImpl Server = new SampleServerImpl();

     //put the local instance in the registry
     Naming.rebind("SAMPLE-SERVER" , Server);
```

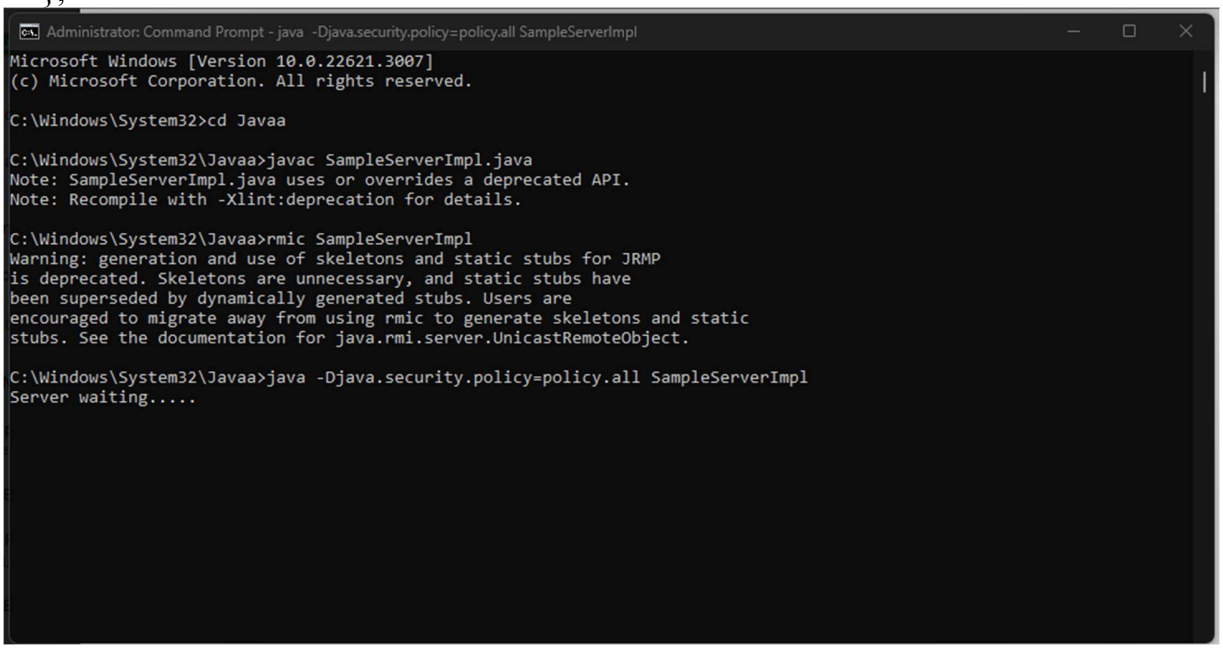
```
      System.out.println("Server waiting.....");
    }
    catch (java.net.MalformedURLException me)       {
      System.out.println("Malformed URL: " + me.toString());   }
    catch (RemoteException re)  {
       System.out.println("Remote exception: " + re.toString());  }
  }

}
```

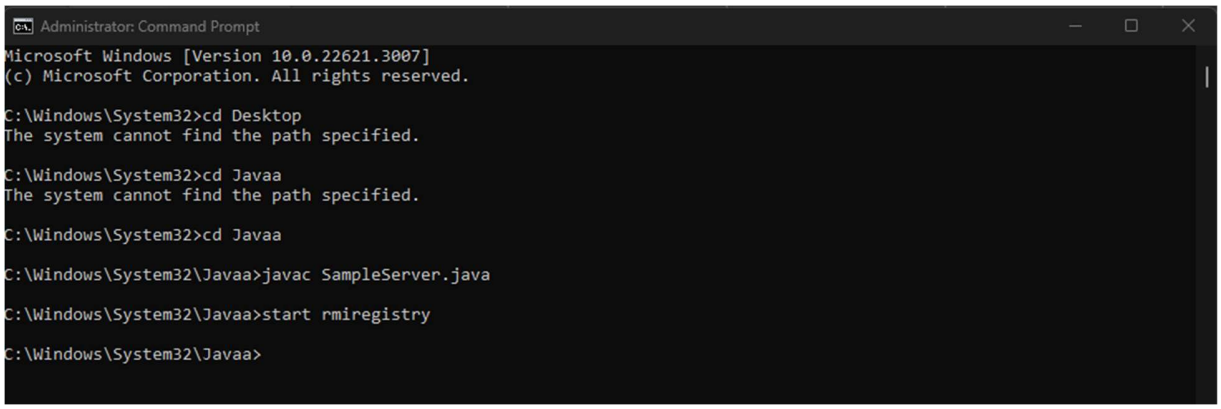
SampleServer.java

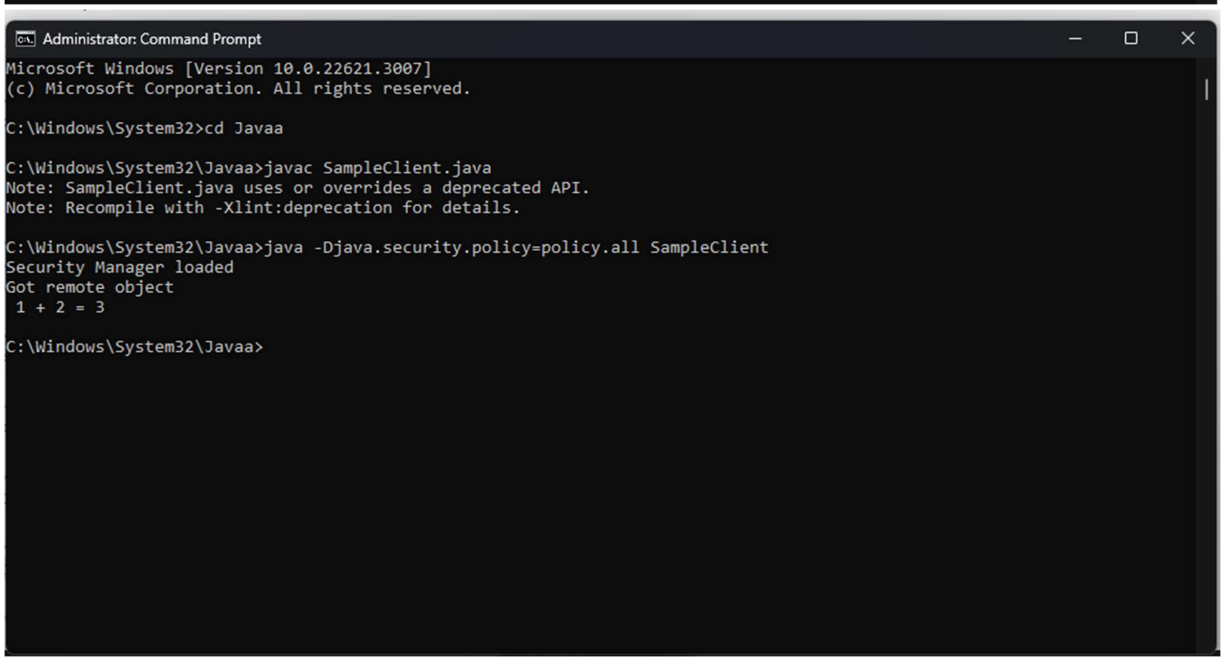
```
import java.rmi.*;

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

policy.all:-

```
grant {
      permission java.security.AllPermission;
 };
```







Conclusion: The client accessed the server's services through a socket-based communication model in a client-server architecture. Initially, the client initiated a connection request to the server, which responded by accepting the connection. Once the connection was established, the client sent requests to the server, specifying the desired service. The server processed these requests, executed the corresponding services, and returned the results to the client through the established socket connection. This bidirectional communication allowed seamless interaction, enabling the client to effectively utilize the services provided by the server.