

# ASSIGNMENT NO: 1

## Title:

Multi-threaded client/server Process communication using RMI.

## Problem Statement:

Implement multi-threaded client/server Process communication using RMI.

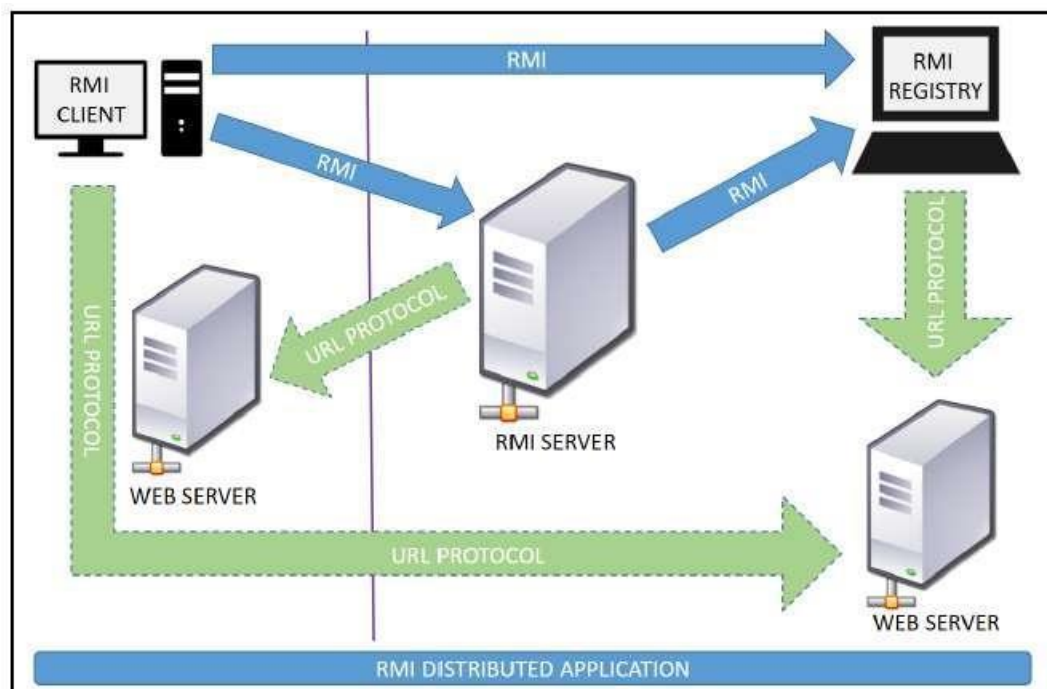
## Tools / Environment:

Java Programming Environment, RMI-Registry, JDK 1.8, Eclipse IDE.

## Theory:

RMI provides communication between java applications that are deployed on different servers and connected remotely using objects called **stub** and **skeleton**. This communication architecture makes a distributed application seem like a group of objects communicating across a remote connection. These objects are encapsulated by exposing an interface, which helps access the private state and behavior of an object through its methods.

The following diagram shows how RMI happens between the RMI client and RMI server with the help of the RMI registry:



**RMI REGISTRY** is a remote object registry, a Bootstrap naming service, that is used by **RMI SERVER** on the same host to bind remote objects to names. Clients on local and remote hosts then look up the remote objects and make remote method invocations.

### Key terminologies of RMI:

The following are some of the important terminologies used in a Remote Method Invocation. **Remote object:** This is an object in a specific JVM whose methods are exposed so they could be invoked by another program deployed on a different JVM.

**Remote interface:** This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.

**RMI:** This is a way of invoking a remote object's methods with the help of a remote interface. It can be carried with a syntax that is similar to the local method invocation.

**Stub:** This is a Java object that acts as an entry point for the client object to route any outgoing requests. It exists on the client JVM and represents the handle to the remote object.

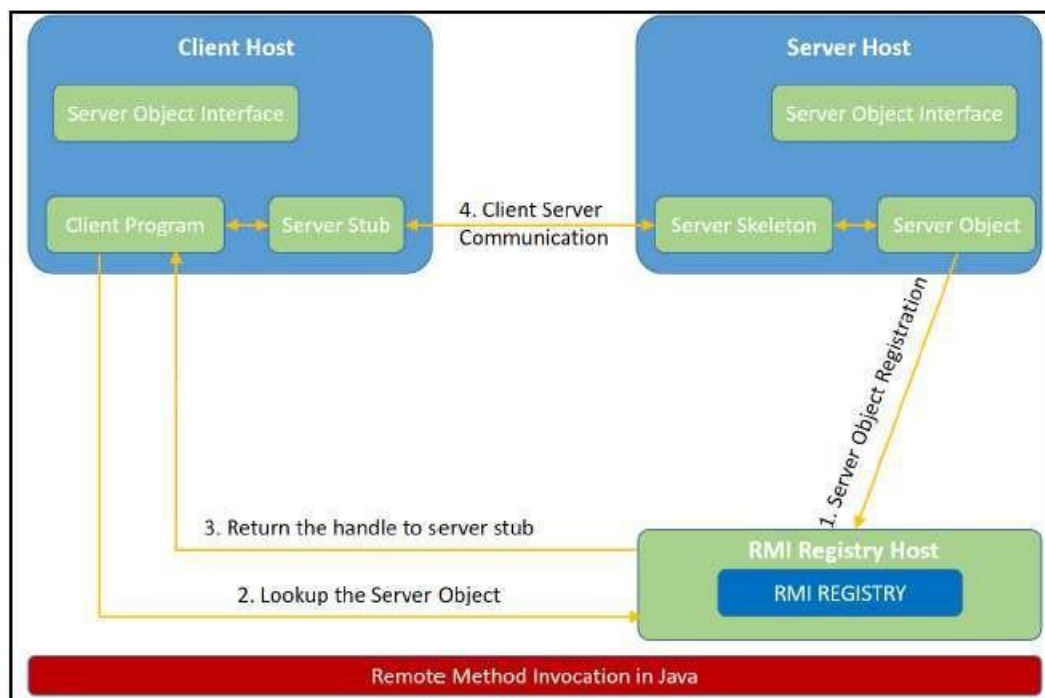
If any object invokes a method on the stub object, the stub establishes RMI by following these steps:

1. It initiates a connection to the remote machine JVM.
2. It marshals (write and transmit) the parameters passed to it via the remote JVM.
3. It waits for a response from the remote object and unmarshals (read) the returned value or exception, then it responds to the caller with that value or exception.

**Skeleton:** This is an object that behaves like a gateway on the server side. It acts as a remote object with which the client objects interact through the stub. This means that any requests coming from the remote client are routed through it. If the skeleton receives a request, it establishes RMI through these steps:

1. It reads the parameter sent to the remote method.
2. It invokes the actual remote object method.
3. It marshals (writes and transmits) the result back to the caller (stub).

The following diagram demonstrates RMI communication with stub and skeleton involved:



## **Designing the solution:**

The essential steps that need to be followed to develop a distributed application with RMI are as follows:

1. Design and implement a component that should not only be involved in the distributed application, but also the local components.
2. Ensure that the components that participate in the RMI calls are accessible across networks.
3. Establish a network connection between applications that need to interact using the RMI.

**Remote interface definition:** The purpose of defining a remote interface is to declare the methods that should be available for invocation by a remote client.

Programming the interface instead of programming the component implementation is an essential design principle adopted by all modern Java frameworks, including Spring. In the same pattern, the definition of a remote interface takes importance in RMI design as well.

**2. Remote object implementation:** Java allows a class to implement more than one interface at a time. This helps remote objects implement one or more remote interfaces. The remote object class may have to implement other local interfaces and methods that it is responsible for. Avoid adding complexity to this scenario, in terms of how the arguments or return parameter values of such component methods should be written.

**3. Remote client implementation:** Client objects that interact with remote server objects can be written once the remote interfaces are carefully defined even after the remote objects are deployed.

Let's design a project that can sit on a server. After that different client projects interact with this project to pass the parameters and get the computation on the remote object execute and return the result to the client components.

## **Implementing the solution:**

1. Creating remote interface, implement remote interface, server-side and client-side program and compile the code.
2. Generate a Stub.
3. Install Files on the Client and Server Machines.
4. Start the RMI Registry on the Server Machine.
5. Start the Server.
6. Start the Client.

## **Conclusion:**

Remote Method Invocation (RMI) allows you to build Java applications that are distributed among several machines. Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows you to build distributed applications.

## Code:

### BankAccount.java

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

public interface BankAccount extends Remote {
    void deposit (int amount) throws RemoteException;
    void withdraw (int amount) throws RemoteException;
    double getBalance() throws RemoteException;
}
```

### BankAccountImpl.java.

```
import java.rmi.RemoteException;

public class BankAccountImpl implements BankAccount{

    private double amount=50;

    @Override
    public void deposit(int amount) throws RemoteException {
        // TODO Auto-generated method stub
        this.amount+=amount;
    }

    @Override
    public void withdraw(int amount) throws RemoteException {
        // TODO Auto-generated method stub
        this.amount-=amount;
    }

    @Override
    public double getBalance() throws RemoteException {
        // TODO Auto-generated method stub
        return amount;
    }
}
```

## **Server.java**

```
import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.UnknownHostException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class Server {
    public static void main(String[] args) {
        try {
            BankAccountImpl bankAccount= new BankAccountImpl();
            BankAccount
exportObject=(BankAccount)UnicastRemoteObject.exportObject(bankAccount, 0);
            Registry registry = LocateRegistry.createRegistry(52369);
            String
url="rmi://" +InetAddress.getLocalHost().getHostAddress()+":52369/BankAccount";
            Naming.rebind(url, exportObject);
            System.out.println("Waiting for the client's call");

        } catch (RemoteException | UnknownHostException | MalformedURLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            System.out.println("Error while trying to connect to BankAccount object");
        }
    }
}
```

## **Client.java**

```
import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.UnknownHostException;
import java.rmi.Naming;
```

```

import java.rmi.NotBoundException;
import java.rmi.Remote;
import java.rmi.RemoteException;

public class Client {
    public final static int amount=10;
    public static void main(String[] args) {
        try {
            String url= "rmi://" +InetAddress.getLocalHost().getHostAddress() +
":52369/BankAccount";
            BankAccount bankAccount =(BankAccount) Naming.lookup(url);
            System.out.println("Current amount in the bank
account"+bankAccount.getBalance());

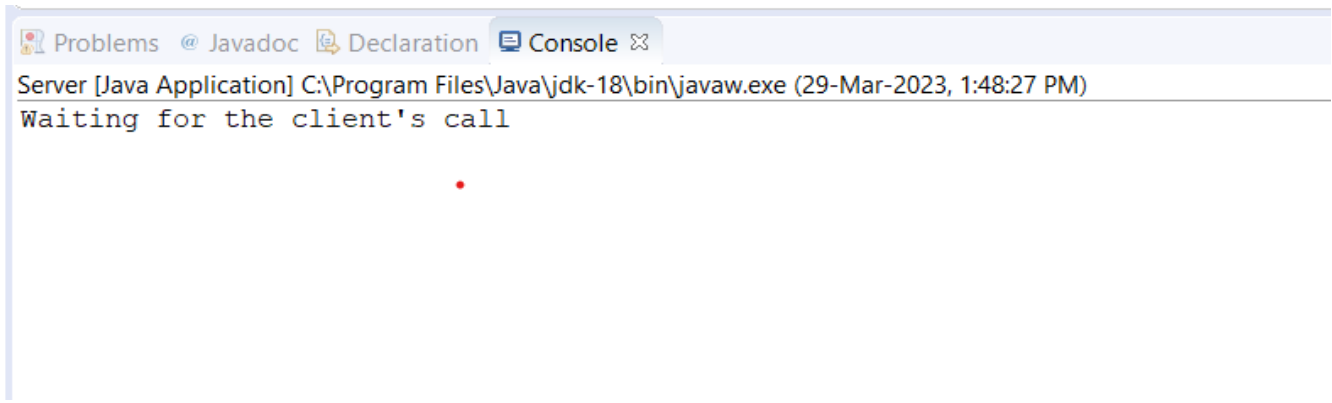
            System.out.println("Deposit----->" + amount);
            bankAccount.deposit(amount);

            System.out.println("Amount after the deposit"+ bankAccount.getBalance());

            System.out.println("Withdraw----->" +amount);
            bankAccount.withdraw(amount);
            System.out.println("Amount after the withdraw"+ bankAccount.getBalance());
        }
        catch (UnknownHostException | MalformedURLException | RemoteException |
NotBoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

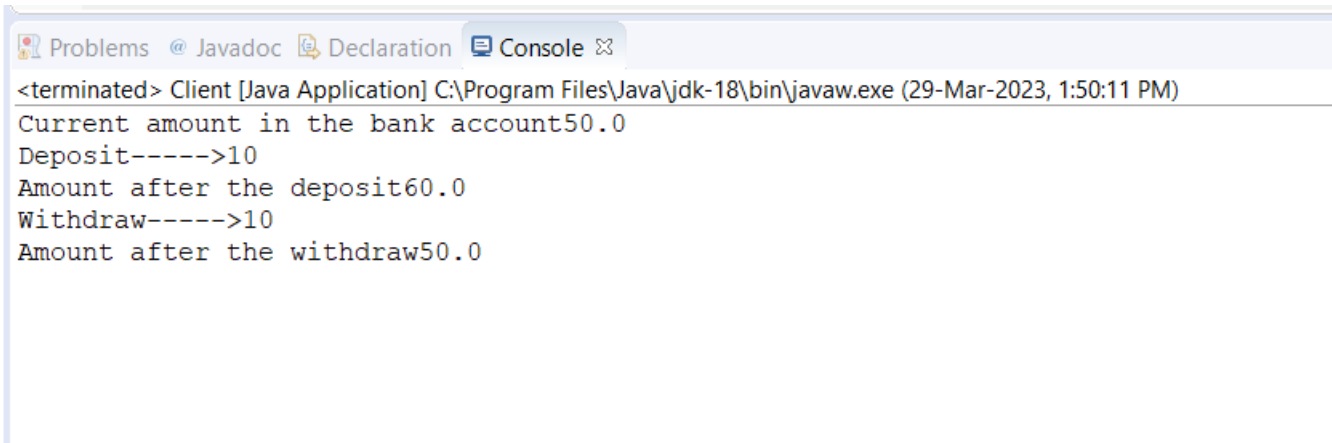
## Output:



The screenshot shows an IDE console window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of a Java application. The text in the console is: "Server [Java Application] C:\Program Files\Java\jdk-18\bin\javaw.exe (29-Mar-2023, 1:48:27 PM)" followed by "Waiting for the client's call". A small red dot is visible in the center of the console area.

```
Server [Java Application] C:\Program Files\Java\jdk-18\bin\javaw.exe (29-Mar-2023, 1:48:27 PM)
Waiting for the client's call
```

Server output



The screenshot shows an IDE console window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of a Java application. The text in the console is: "<terminated> Client [Java Application] C:\Program Files\Java\jdk-18\bin\javaw.exe (29-Mar-2023, 1:50:11 PM)" followed by "Current amount in the bank account50.0", "Deposit----->10", "Amount after the deposit60.0", "Withdraw----->10", and "Amount after the withdraw50.0".

```
<terminated> Client [Java Application] C:\Program Files\Java\jdk-18\bin\javaw.exe (29-Mar-2023, 1:50:11 PM)
Current amount in the bank account50.0
Deposit----->10
Amount after the deposit60.0
Withdraw----->10
Amount after the withdraw50.0
```

Client output