

## Department of Computer Engineering

Academic Term : January - May 2024

Class: B.E. (Computer) Semester VIII

Division: B

Subject Name: Distributed Systems CSC 801

<b>Name</b>	<b>Riya Rajesh Patil</b>
<b>Roll No.</b>	<b>9282</b>
<b>Experiment No.</b>	<b>3</b>
<b>Experiment Title</b>	<b>To implement Remote Method Invocation</b>

**Aim** - To implement Remote Method Invocation

### **Adder.java**

```
import java.rmi.*;  
  
public interface Adder extends Remote {  
  
    public int add(int x,int y)throws RemoteException;  
  
}
```

### **AdderRemote.java**

```
// Implementing the remote interface  
  
public class AdderRemote implements Adder {  
  
    // Implementing the interface method  
  
    public int add(int x, int y) {  
  
        return x+y;  
  
    }  
  
}
```

### **Client.java**

```
import java.rmi.registry.LocateRegistry;  
  
import java.rmi.registry.Registry;  
  
public class Client {
```

```

private Client() {
}

public static void main(String[] args) {
    try {
        // Getting the registry
        Registry registry = LocateRegistry.getRegistry(null);
        // Looking up the registry for the remote object
        Adder stub = (Adder) registry.lookup("Hello");
        // Calling the remote method using the obtained object
        int result = stub.add(Integer.parseInt(args[0]), Integer.parseInt(args[1]));

        System.out.println("Result From Server: " + result);
        // System.out.println("Remote method invoked");
    } catch (Exception e) {
        System.err.println("Client exception: " + e.toString());
        e.printStackTrace();
    }
}
}
}

```

### **Server.java**

```

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class Server extends AdderRemote {
    public Server() {
    }

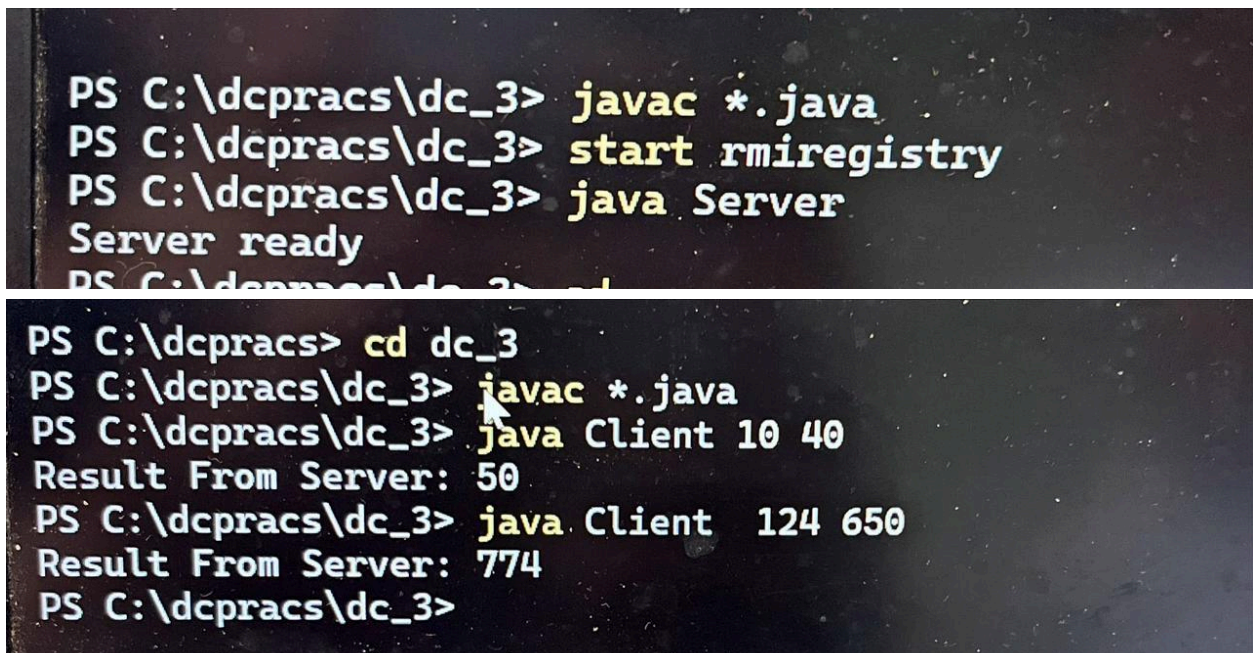
    public static void main(String args[]) {
        try {

```

```

// Instantiating the implementation class
AdderRemote obj = new AdderRemote();
// Exporting the object of implementation class
// (here we are exporting the remote object to the stub)
Adder stub = (Adder) UnicastRemoteObject.exportObject(obj, 0);
// Binding the remote object (stub) in the registry
Registry registry = LocateRegistry.getRegistry();
registry.bind("Hello", stub);
System.err.println("Server ready");
} catch (Exception e) {
System.err.println("Server exception: " + e.toString());
e.printStackTrace();
}
}
}

```



The image consists of two screenshots of a Windows command prompt. The top screenshot shows the compilation and execution of a server program. The bottom screenshot shows the execution of a client program that interacts with the server.

```

PS C:\dcpracs\dc_3> javac *.java
PS C:\dcpracs\dc_3> start rmiregistry
PS C:\dcpracs\dc_3> java Server
Server ready
PS C:\dcpracs\dc_3>

PS C:\dcpracs> cd dc_3
PS C:\dcpracs\dc_3> javac *.java
PS C:\dcpracs\dc_3> java Client 10 40
Result From Server: 50
PS C:\dcpracs\dc_3> java Client 124 650
Result From Server: 774
PS C:\dcpracs\dc_3>

```

## Conclusion-

In conclusion, the provided code demonstrates a basic implementation of Remote Method Invocation (RMI) in Java, which allows clients to invoke methods on remote objects residing on a server.

**1. Adder Interface (Adder.java):** Declares a remote interface `Adder` with a method `add(int x, int y)`. This interface extends `Remote` and declares `RemoteException` for handling remote method invocation errors.

**2. AdderRemote Class (AdderRemote.java):** Implements the `Adder` interface, providing the implementation for the `add` method. This class acts as the remote object that clients interact with.

**3. Server Class (Server.java):** Exports the `AdderRemote` object, creates a stub, and binds it to the RMI registry. The server listens for client requests and executes the `add` method on the remote object.

**4. Client Class (Client.java):** Looks up the remote object in the RMI registry, obtains a stub, and invokes the `add` method remotely. It then receives and displays the result returned by the server.

From the output it is seen that the server code is executed to print the message server ready indicating that the server is ready. Before this process starts, `rmiregistry` commands help to start the rmi registry. The client code is run and the input is given in the form of parameters. The server processes this and the output is given back to the client.

## Postlab -

### 1. What are the different times at which a client can be bound to a server?

Some common times at which a client can be bound to a server include:

1. **Static Binding:** In static binding, the client is bound to a specific server at compile time or configuration time. This binding is typically established by specifying the server's network address or endpoint in the client's configuration files or code. Once bound, the client communicates exclusively with the specified server.

2. **Dynamic Binding:** In dynamic binding, the client binds to a server at runtime. This can occur in several ways:

3. **On Demand Binding:** The client binds to a server only when needed, rather than at startup. This approach is common in connection-oriented protocols like Remote Procedure Call (RPC) or Java RMI, where the client obtains a stub or proxy for the remote object from a registry or naming service when it needs to invoke remote methods.

4. **Session Binding:** The client is bound to a specific server for the duration of a session or interaction. Once the session ends, the binding is released. This approach is common in stateful protocols or applications where maintaining

session state on the server is necessary.

## **2. How does a binding process locate a server? Mention with diagram**

The binding process in a distributed computing environment typically involves locating a server, which can be achieved through various mechanisms such as service registries, naming services, or service discovery protocols. Here's how the binding process works, illustrated with a diagram:

### **Service Registration:**

Servers register their network address (IP address and port) and other metadata (such as service name, version, and capabilities) with a central service registry or naming service.

The service registry maintains a database or directory of registered services, allowing clients to look up and locate available servers.

### **Client Lookup:**

When a client needs to bind to a server, it queries the service registry or naming service to locate the desired server.

The client specifies criteria such as the service name or type, version, and any other relevant parameters to narrow down the search.

### **Service Discovery:**

The service registry or naming service responds to the client's query by providing a list of available servers that match the specified criteria.

Alternatively, the client may use multicast DNS (mDNS), multicast-based discovery protocols, or other service discovery mechanisms to discover available servers dynamically within the local network.

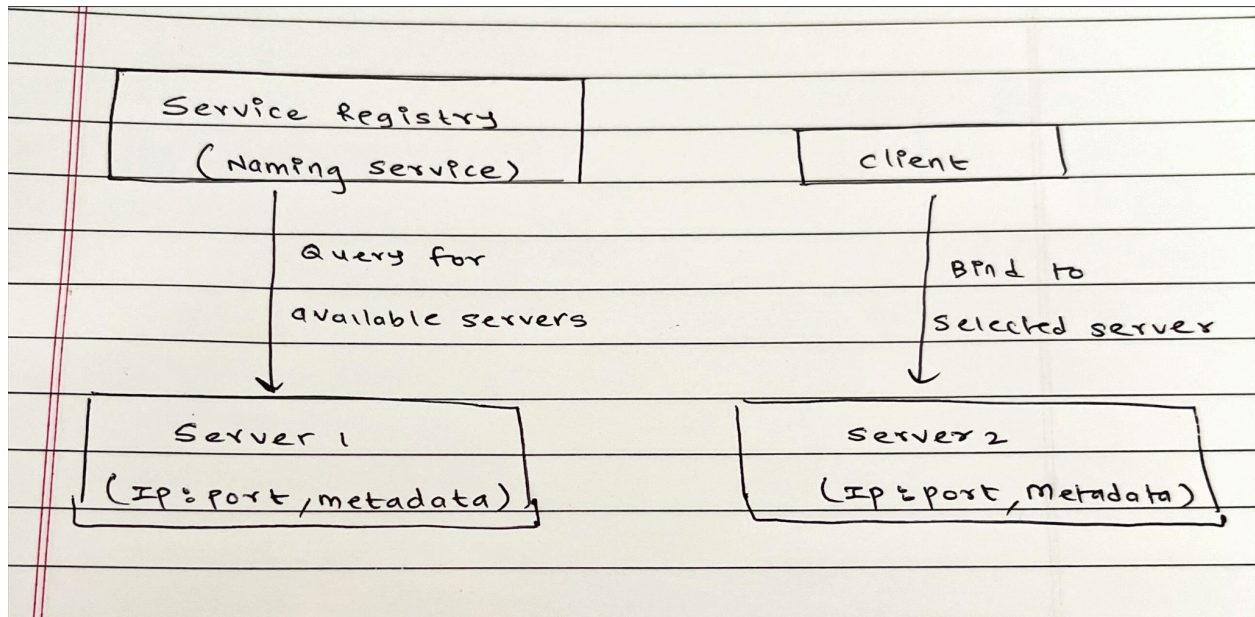
### **Selection and Binding:**

The client selects a server from the list of available servers based on criteria such as load, proximity, or other application-specific requirements.

Once the server is selected, the client establishes a connection or binds to the server by using its network address and port information obtained from the service registry or discovery process.

### **Communication:**

With the binding process complete, the client can now communicate with the selected server by sending requests and receiving responses over the established connection.



**3. Name some optimization methods adopted for better performance of distributed applications using RPC and RMI.**

To optimize the performance of distributed applications using Remote Procedure Call (RPC) and Remote Method Invocation (RMI), various techniques and strategies can be employed. Some optimization methods include:

1. **Batching:** Instead of sending individual RPC/RMI requests one at a time, batch multiple requests together and send them in a single network round-trip. This reduces the overhead associated with network communication and improves efficiency.
2. **Caching:** Cache frequently accessed data or results of remote invocations locally to avoid unnecessary network round-trips. This can reduce latency and improve response times, especially for repetitive operations.
3. **Asynchronous Communication:** Use asynchronous RPC/RMI calls to allow clients to continue processing while waiting for responses from remote servers. This can improve concurrency and throughput by overlapping computation with communication.
4. **Load Balancing:** Distribute incoming RPC/RMI requests across multiple server instances using load balancing techniques. This ensures optimal resource utilization and prevents overloading of individual servers, leading to better overall performance and scalability.

5. Parallelism: Parallelize RPC/RMI calls by invoking multiple remote procedures or methods concurrently. This leverages the processing power of multi-core systems and distributed environments to execute tasks in parallel, leading to faster execution and improved throughput.