

RPC/RMI (Remote Procedure Call / Remote Method Invocation) ,

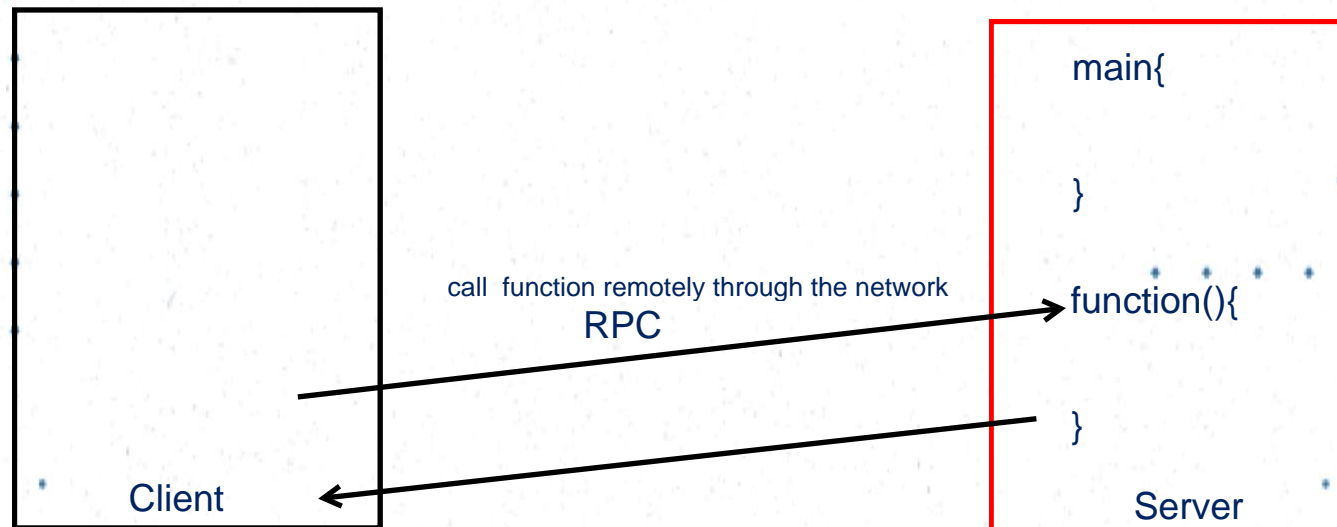
allow communication between processes running on different machines or network nodes.

Lecture 4 – RPC/RMI

Feature	Socket Programming	RPC / RMI
Abstraction Level	Low-level (manages data transmission, connection handling)	High-level (abstracts communication details, like calling a local function)
Complexity	Requires manual handling of connection, data serialization, and parsing	Automatically handles communication and data exchange
Ease of Use	Requires explicit socket creation, message handling, and error handling	Looks like a normal function/method call from the client side
Data Transfer	Raw bytes or structured data using protocols like TCP/UDP	Objects (RMI) or structured messages (RPC)
Language Support	Language-agnostic (supports any language but needs custom serialization)	RMI is Java-specific, while RPC supports multiple languages
Performance	High control over network performance and efficiency	Slightly more overhead due to abstraction and serialization

Topics

- Introduction to Distributed Objects and Remote Invocation
- Remote Procedure Call
- Java RMI



Two main ways to do DC (apart from socket programming)

Distribute computing - DC

➤ Remote Method Invocation (RMI)

Example :
Distributed applications (e.g., banking systems, cloud services)
Microservices communication
Remote file access and database queries

- ❖ Local object invokes methods of an object residing on a remote computer
- ❖ Invocation as if it was a local method call

➤ Event-based Distributed Programming

- ❖ Objects receive asynchronous notifications of events happening on remote computers/processes

Example Use Cases:
IoT networks (e.g., smart home devices reacting to sensor data)
Stock trading systems (e.g., real-time market price updates)
Social media notifications
Distributed monitoring systems

Remote Procedure Call (RPC)

RPC is a mechanism that allows a client to call a procedure (function) on a remote computer as if it were a local procedure.

- Objects that can receive remote method invocations are called remote objects and they implement a remote interface.
- Programming models for distributed applications are:

➤ Remote Procedure Call (RPC)

- ❖ Client calls a procedure implemented and executing on a remote computer
- ❖ Call as if it was a local procedure

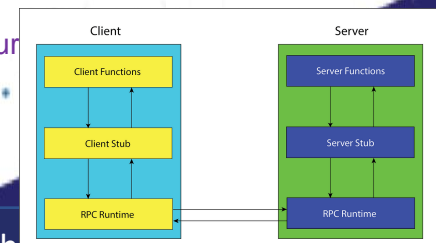
How it Works:

A client calls a procedure implemented on a remote server.

The call is forwarded to a client stub, which serializes the request.

The request is sent over the network to the server stub, which deserializes and calls the actual procedure.

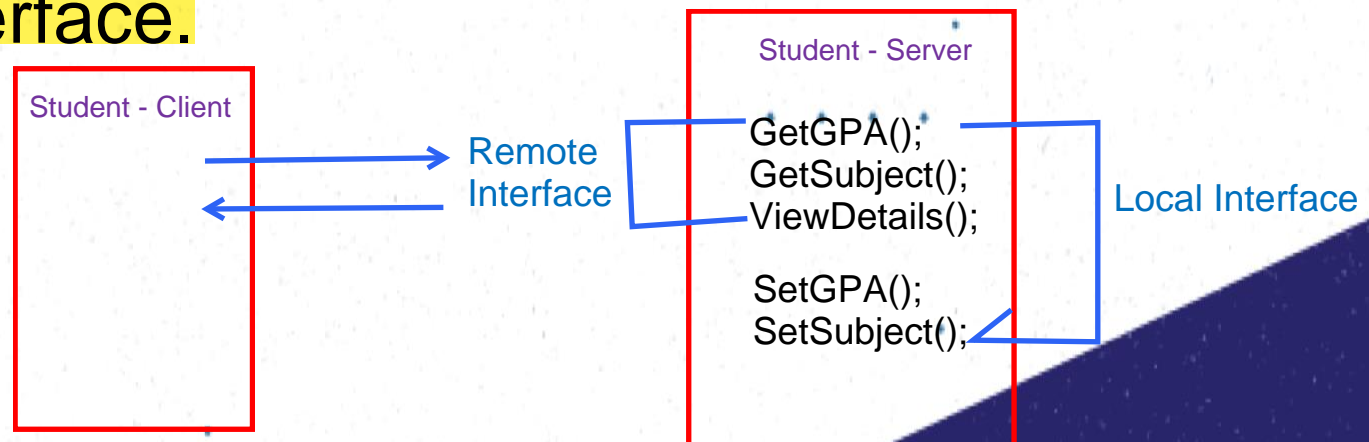
The response is sent back to the client.



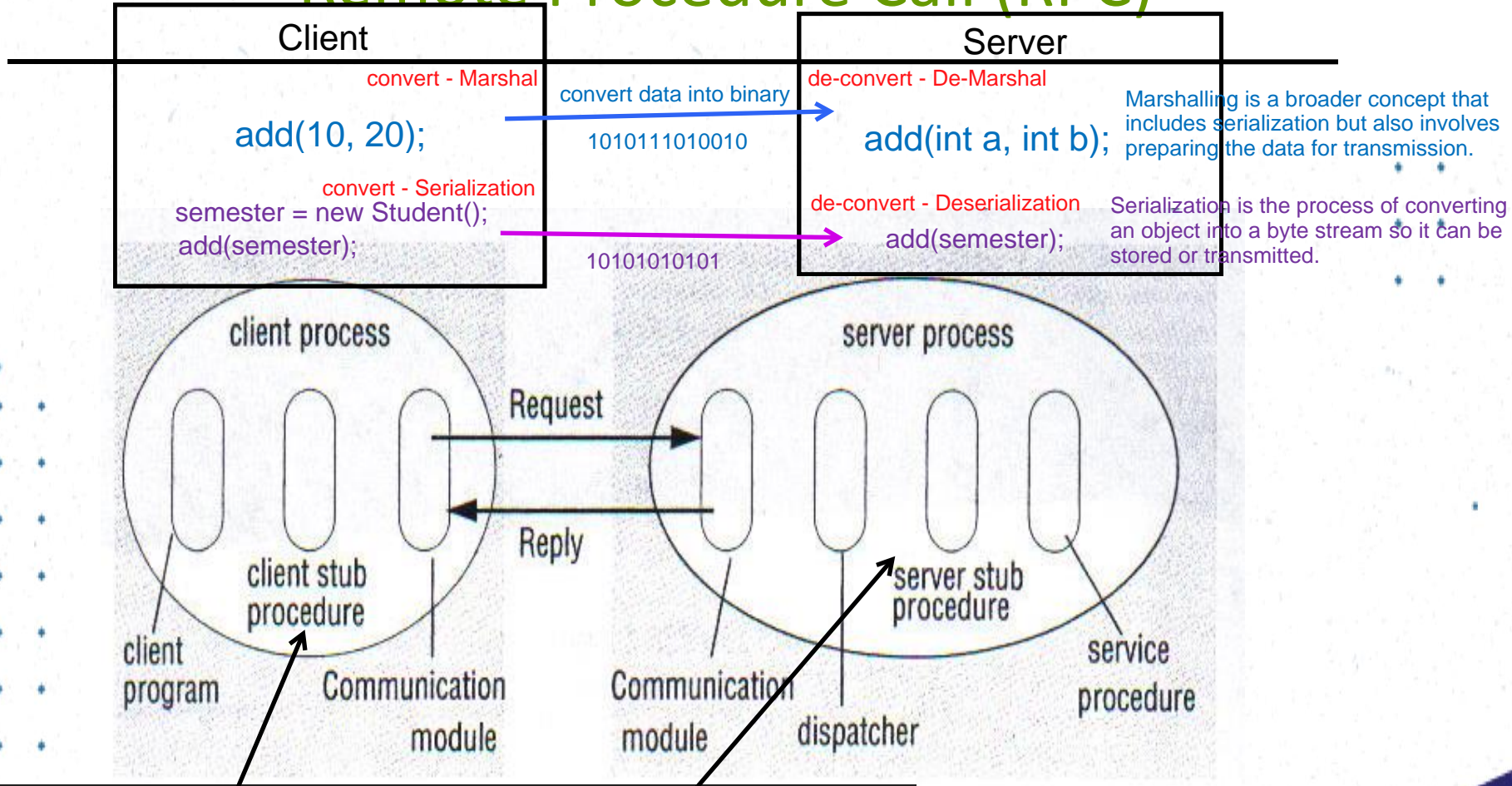
RPC Interfaces

■ Interfaces for RPC

- An explicit interface is defined for each module.
- An Interface hides all implementation details.
- Accesses the variables in a module can only occur through methods specified in interface.



Remote Procedure Call (RPC)



Feature	Client Stub	Server Stub
Location	Runs on client-side	Runs on server-side
Purpose	Acts as a proxy for the remote method	Acts as a proxy for the actual implementation
Tasks	Marshals method arguments & sends request	Unmarshals request, calls method, and marshals response
Communication	Sends data to server stub	Sends result back to client stub

e context of a procedural language

Edn. 4, Pearson Education 2005

shana Kasthurirathna

SUN ONC RPC

- RPC only addresses procedure calls.
- RPC is not concerned with objects and object references.
- A client that accesses a server includes one stub procedure for each procedure in the service interface. RPC (Remote Procedure Call), the client needs a stub (proxy function) for every remote procedure (function) that it wants to call on the server.
- A client stub procedure is similar to a proxy method of RMI (discussed later).
- A server stub procedure is similar to a skeleton method of RMI (discussed later).

Common in C/C++: Used in systems like SUN ONC RPC (Open Network Computing RPC).

Couloris, Dollimore and Kindberg *Distributed Systems: Concepts & Design* Edn. 4, Pearson Education 2005

Strength and Weaknesses of RPC

- RPC (or even RMI) is not well suited for adhoc query processing. (e.g. SQL queries)
- It is not suited for transaction processing without special modification.
- A separate special mode of querying is proposed – Remote Data Access (RDA).
- RDA is specially suited for DBMS.
- In a general client_server environment both RPC and RDA are needed.

RPC is technology specific (server is in java also client is in java language)



RMI

RMI allows method invocations to occur between objects residing in different Java Virtual Machines (JVMs). RMI provides a means of invoking methods on remote objects as if they were local, simplifying the development of distributed applications.

Java Remote Method Invocation

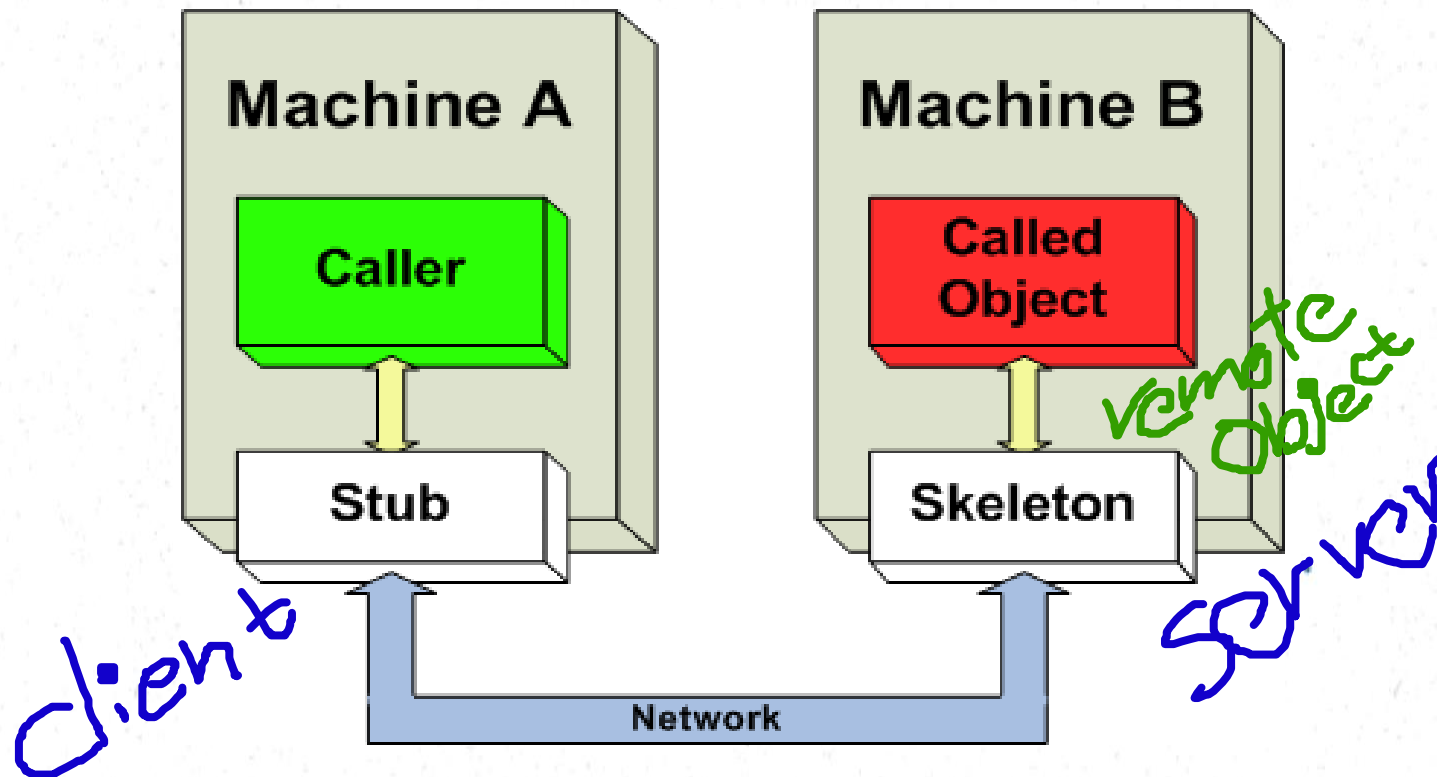


Fig: Distributed Object Technology

- **RMI Server, Client, Interface, Stubs, Skeletons:**
 - **RMI Server:** This hosts the remote objects that clients can interact with.
 - **RMI Client:** This invokes methods on remote objects hosted by the server.
 - **Remote Interface:** Defines the methods that can be invoked remotely.
 - **Stubs:** Client-side proxies that handle communication between the client and the server.
 - **Skeletons:** Server-side components that receive requests from clients and forward them to the actual remote object (though skeletons are deprecated since Java 2 SDK version 1.2).
- **RMI Registry:** A naming service for RMI where remote objects are registered and clients can look up and retrieve references to these objects.
- **Objects + RPC = RMI:** RMI combines the concepts of objects (as in object-oriented programming) with Remote Procedure Calls (RPC) to allow method calls on remote objects.
- **Method Invocation between Different JVMs:** RMI allows methods to be invoked across JVMs on different machines, enabling communication between distributed components of an application.
- **Java RMI API:** The core set of classes and interfaces that support RMI, including `Remote`, `UnicastRemoteObject`, and `Naming`.
- **JRMP (Java Remote Method Protocol):** The protocol used by RMI to enable communication between JVMs. It handles marshalling, communication, and unmarshalling of data.
- **Java Object Serialization:** Java objects are serialized to allow them to be transmitted over the network. Serialization converts objects into a byte stream so they can be sent across a network.
- **Parameter Marshalling:** The process of converting parameters to a form that can be transmitted over the network. This involves serializing the arguments passed to the remote method.

RMI System Architecture

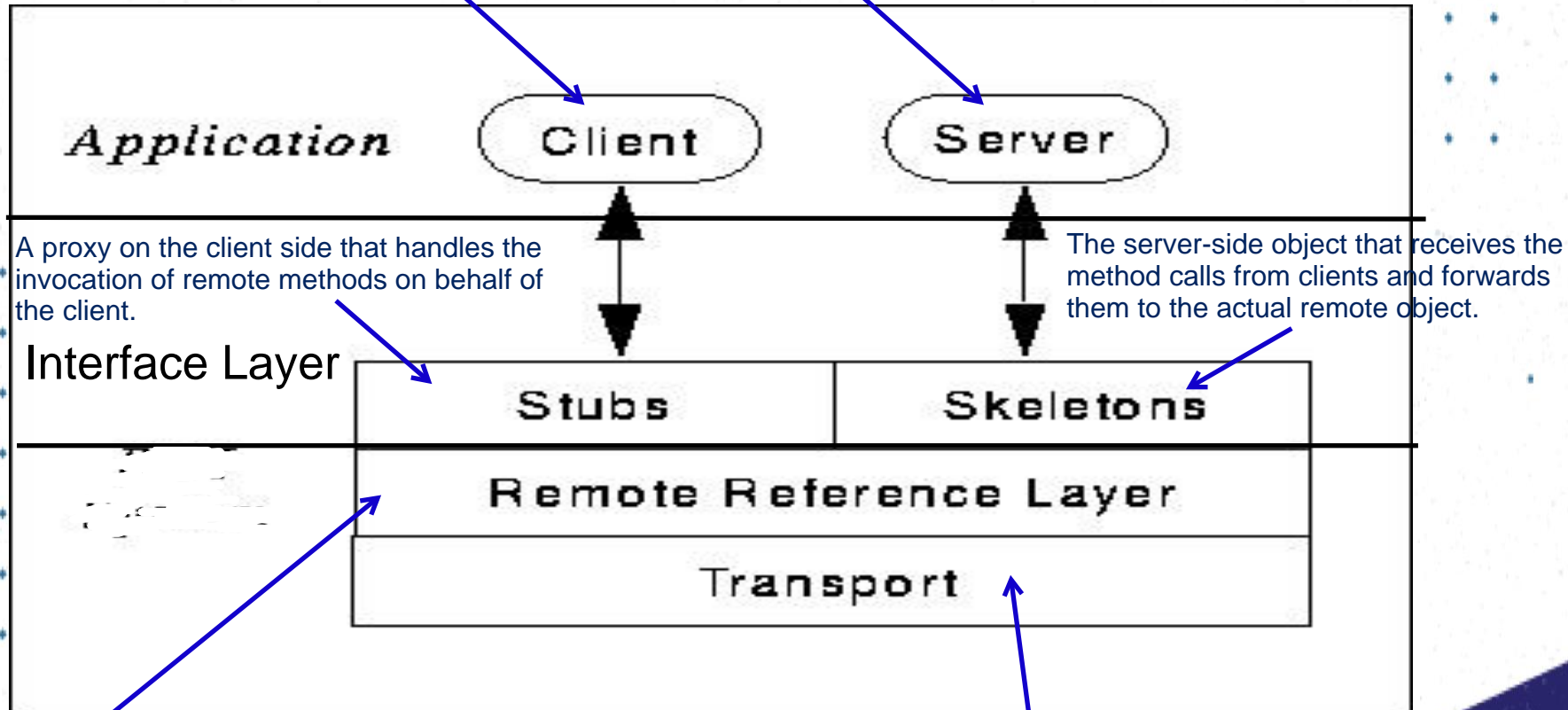
Lets divide into two perspectives:

- Layered Structure
- Working Principles

RMI Layered Structure

Makes method calls on remote objects hosted on the server.

Hosts remote objects and waits for client requests.



RMI Registry:

The registry that allows the client to look up and get references to remote objects.

Fig: RMI Layered Structure

TCP/IP: The protocol used to transport the data across the network

RMI Layered Structure

- Application layer: Server, Client
- Interface: Client stub, Server skeleton
- Remote Reference layer: RMI registry
- Transport layer: TCP/IP

Real-World Use Cases of RMI:

ATM Machines: Connecting to a remote bank database for transactions.

Online Shopping Portals: Fetching real-time product details from a central inventory.

Stock Market Applications: Getting real-time stock prices from a stock exchange server.

Remote File Management: Accessing and modifying files on a remote system.

RMI Working Principles

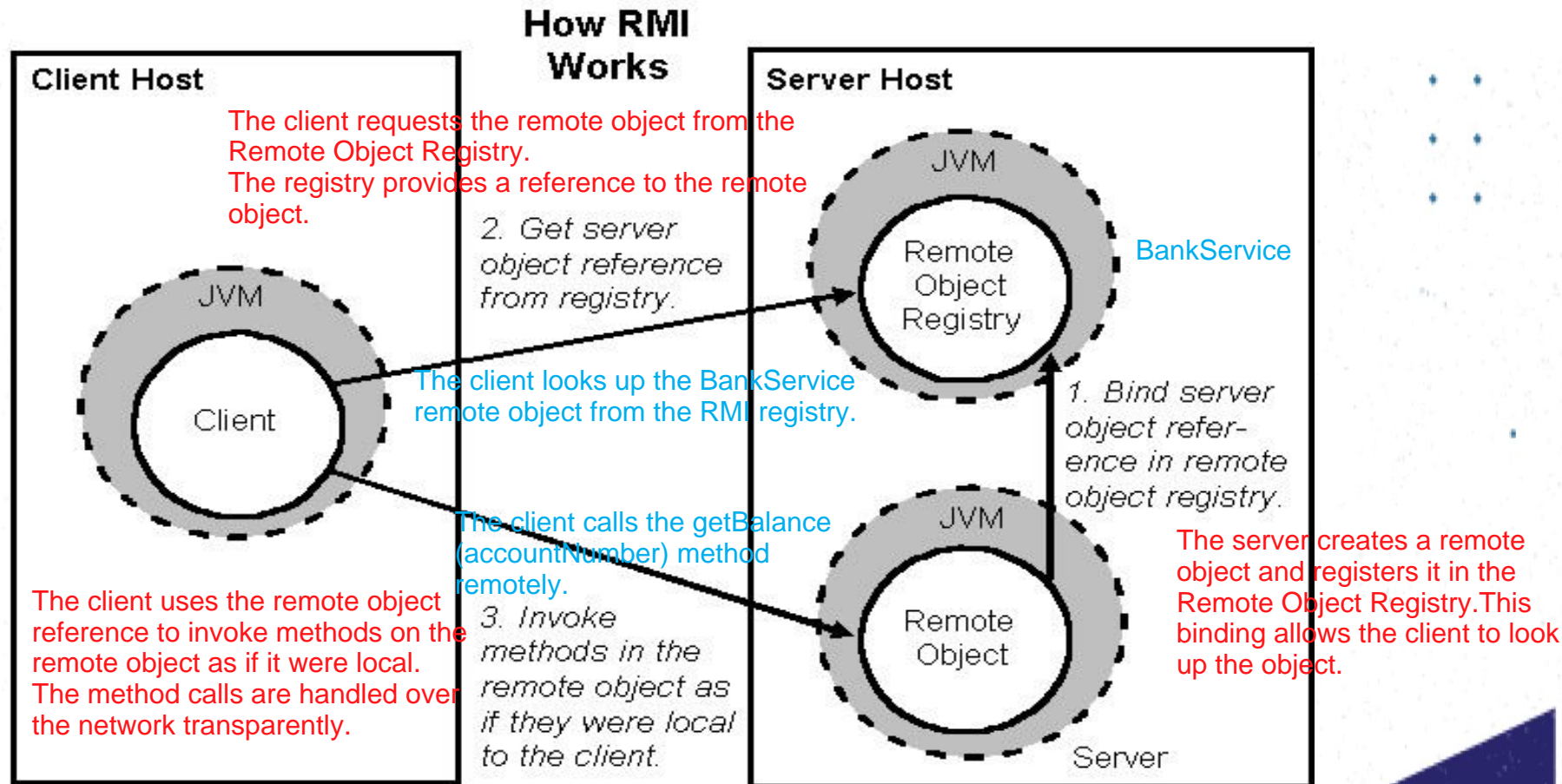


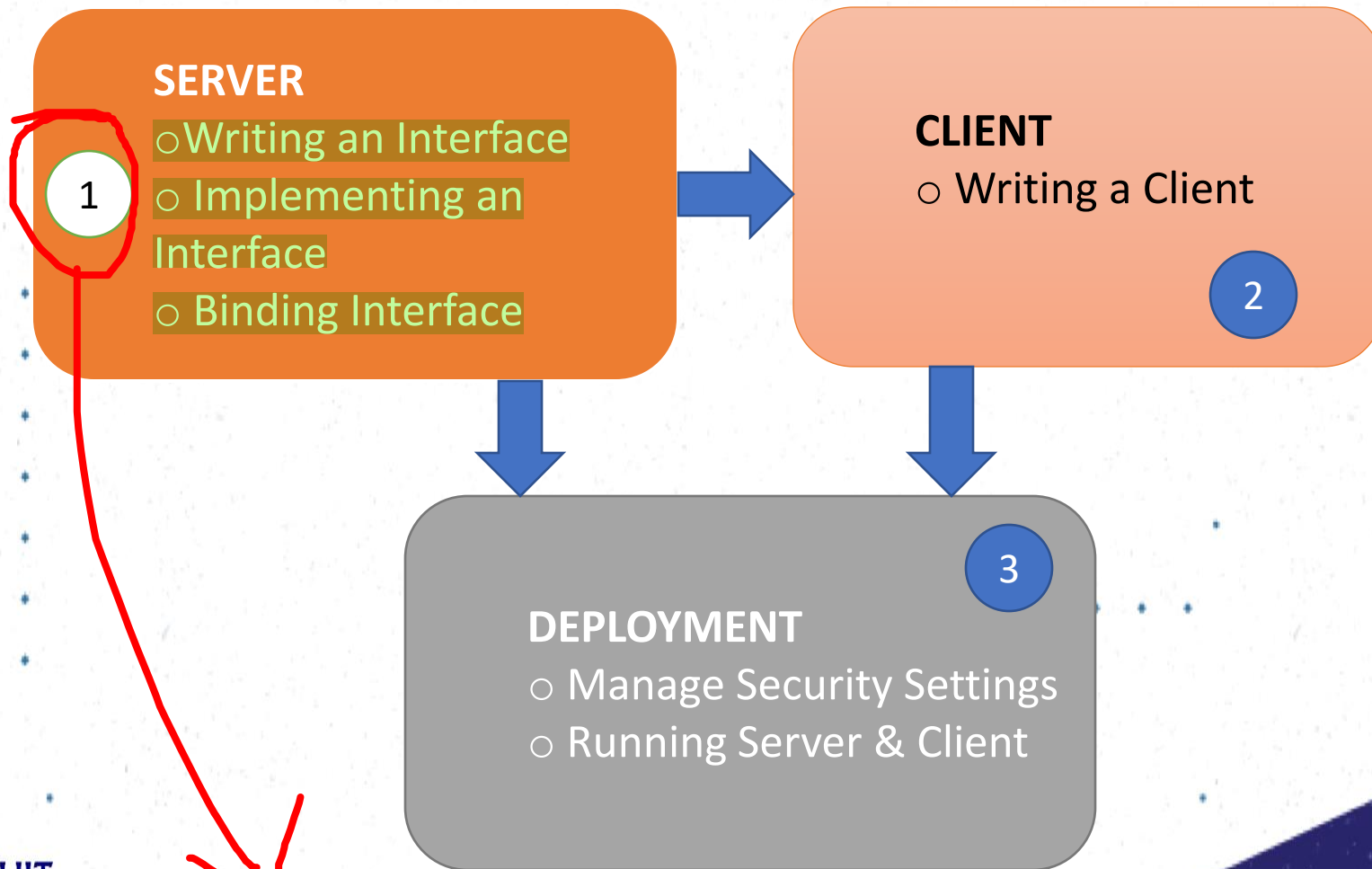
Fig: RMI Working principles

The bank server creates a BankService object (remote object). It binds this object in the RMI registry.

Ready to Develop One?



A Simple RMI Application



Service Interface: An Agreement Between Server & Client

- Factorial Operation

```
public long factorial(int number) throws  
    RemoteException;
```

- Check Prime Operation

```
public boolean checkPrime(int number) throws  
    RemoteException;
```

```
public BigInteger square(int number) throws  
    RemoteException;
```


Server Application: Writing a Service Interface

```
//interface between RMI client and server

import java.math.BigInteger;
import java.rmi.*;

public interface MathService extends Remote {

    // every method associated with RemoteException
    // calculates factorial of a number
    public long factorial(int number) throws
        RemoteException;

    // check if a number is prime or not
    public boolean checkPrime(int number) throws
        RemoteException;

    //calculate the square of a number and returns
    BigInteger
    public BigInteger square(int number) throws
        RemoteException;

}
```

Server

Writing an Remote Interface

Server Application: Implementing the Service Interface

Server Implementing the Service Interface

```
//MathService Server or Provider

import java.awt.font.NumericShaper;
import java.math.BigInteger;
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

public class MathServiceProvider extends UnicastRemoteObject implements
    MathService {

    // MathServiceProvider implements all the methods of MathService interface
    // service constructor
    public MathServiceProvider() throws RemoteException {
        super();
    }

    // implementation of factorial
    public long factorial(int number) {
        // returning factorial
        if (number == 1)
            return 1;
        return number * factorial(number - 1);
    }
}
```

we can access through remotely



Server Application: Instantiating & Binding the Service

Server

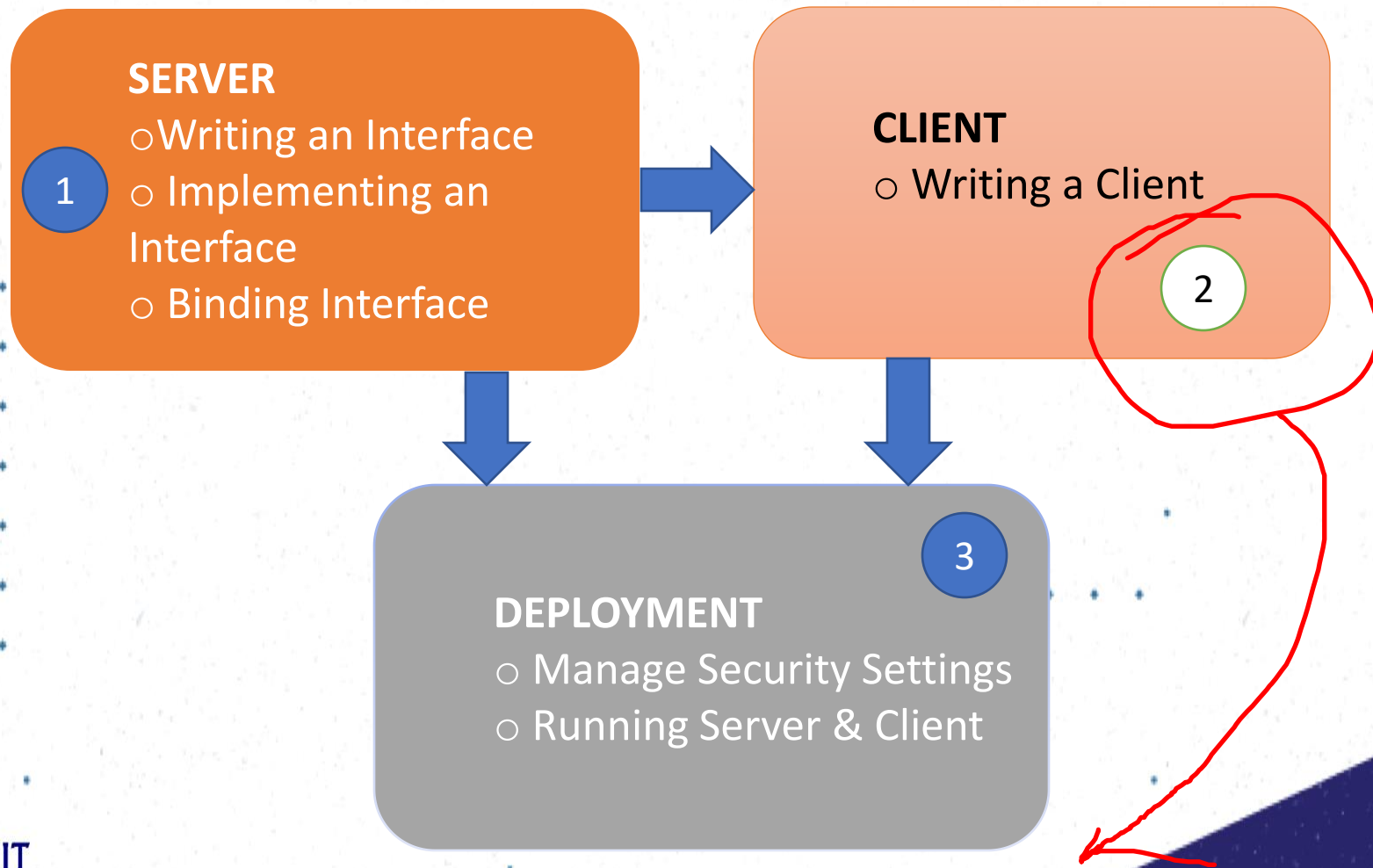
Instantiating & Binding the Service

```
public static void main(String args[]) {
    try {
        // setting RMI security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new
                RMISecurityManager());
        }

        // creating server instance
        MathServiceProvider provider = new
            MathServiceProvider();
        // binding the service with the registry
        LocateRegistry.getRegistry().bind("
            MathService", provider);
        System.out.println("Service is bound to RMI
            registry");
    } catch (Exception exc) {
        // showing exception
        System.out.println("Cant bind the service:
            " + exc.getMessage());
        exc.printStackTrace();
    }
}
```

any name

A Simple RMI Application



Client Application: Service Lookup

Client

```
// Assign security manager
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new
        RMISecurityManager());
}

// Accessing RMI registry for MathService
String hostName = "localhost"; //this can
    be any host
MathService service = (MathService) Naming.
    lookup("//" + hostName
        + "/MathService");
```

look service

Fig: Client locating *MathService* service

Client Application: Accessing Service

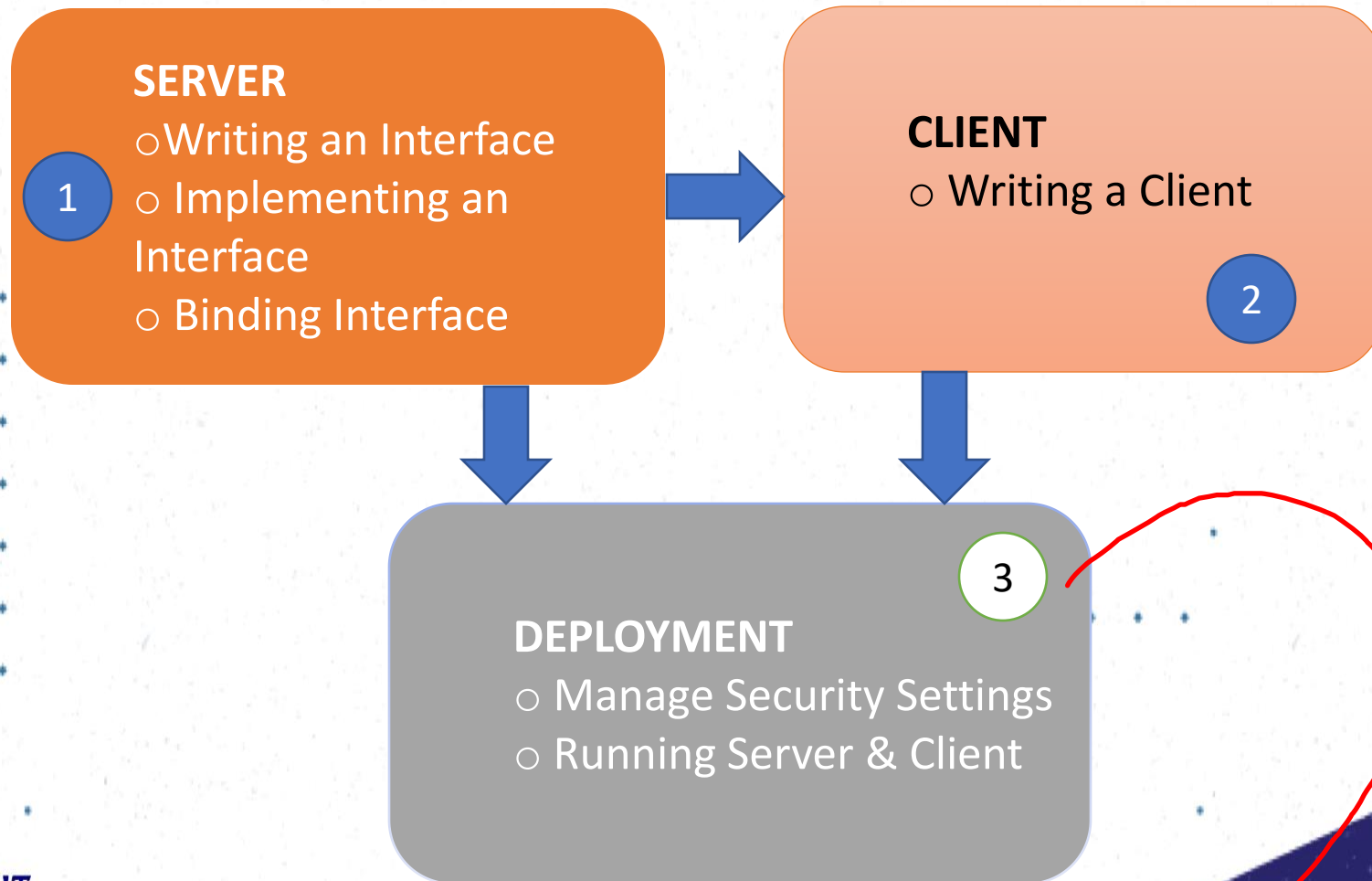
```
// Call to factorial method
System.out.println("The factorial of " + number +
    "=" + service.factorial(number));

// Call to checkPrime method
boolean isprime=service.checkPrime(number);

//Call to square method
BigInteger squareObj=service.square(number);
```

Fig: Client accessing *MathService* service

A Simple RMI Application



Server Deployment: Start RMI Registry

run - step 1

- To start RMI registry on **windows**

```
$ start rmiregistry
```

- To start RMI registry on **Linux**

```
$ rmiregistry \&
```

Server Deployment: Compile the Server

run - step 2

- Compile both **MathService** interface and **MathServiceProvider** class

```
$ javac MathService.java MathServiceProvider.  
    java
```

Security Deployment: Create Security Policy file (Both Client & Server)

run - step 3

- Create a security policy file called *no.policy* with the following content and add it to CLASSPATH
- This step implies for both server and client

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

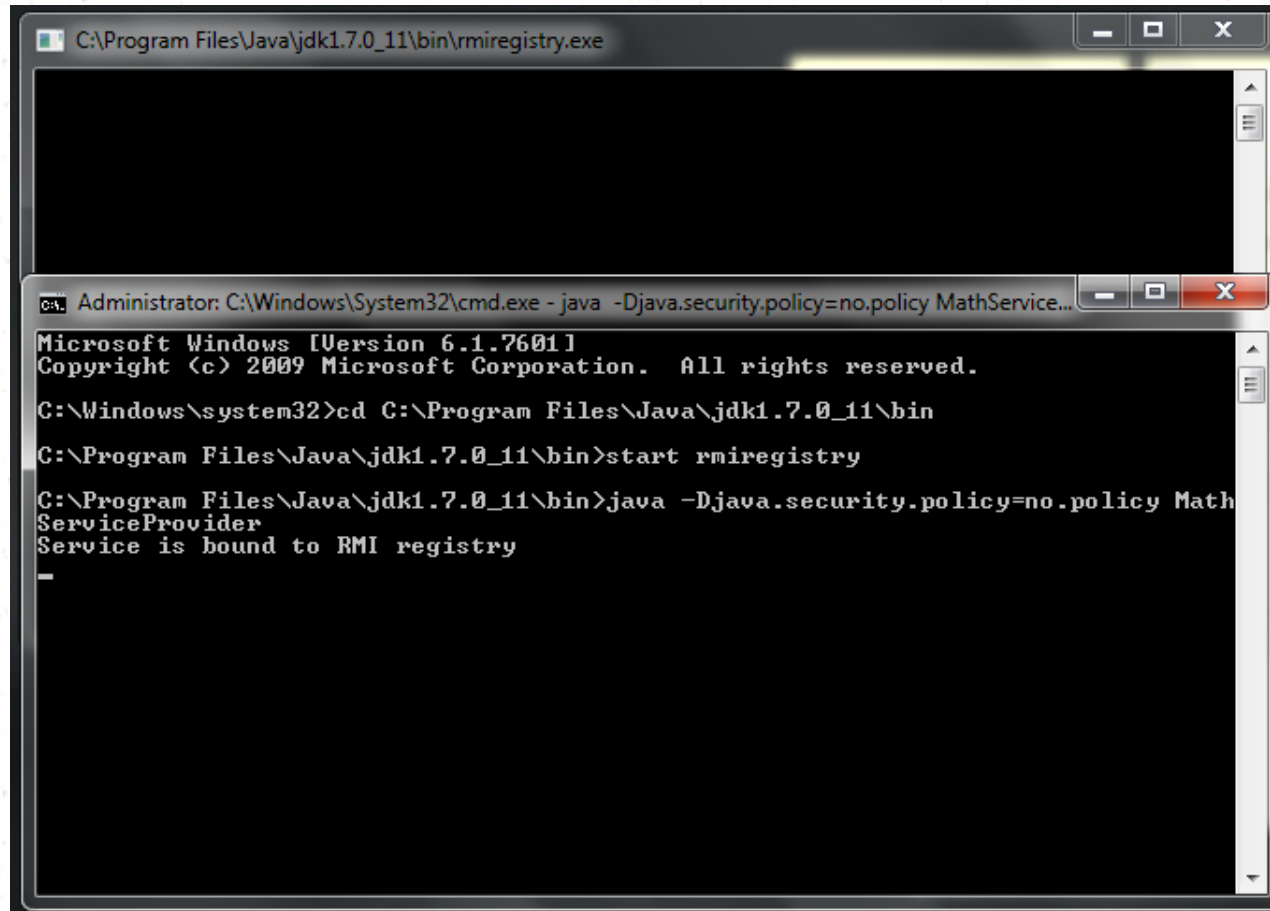

Start the Server

- Execute the command to **run server**

```
$ java -Djava.security.policy=no.policy  
    MathServiceProvider
```

run with policy

Server Running



The screenshot shows two overlapping windows. The top window is titled 'C:\Program Files\Java\jdk1.7.0_11\bin\rmiregistry.exe' and is currently empty. The bottom window is titled 'Administrator: C:\Windows\System32\cmd.exe - java -Djava.security.policy=no.policy MathService...' and contains the following text:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>cd C:\Program Files\Java\jdk1.7.0_11\bin
C:\Program Files\Java\jdk1.7.0_11\bin>start rmiregistry
C:\Program Files\Java\jdk1.7.0_11\bin>java -Djava.security.policy=no.policy Math
ServiceProvider
Service is bound to RMI registry
```

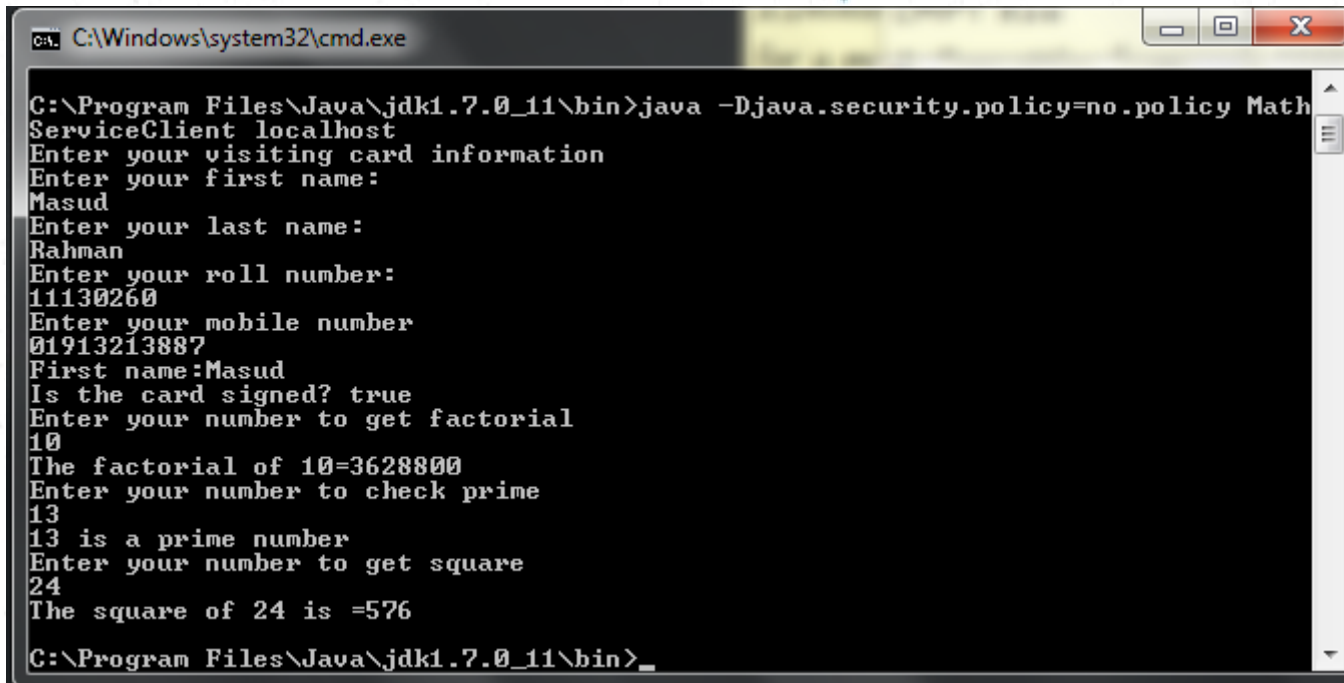
Start the Client

run with policy

- Execute the command to **run client**

```
$ java -Djava.security.policy=no.policy  
    MathServiceClient localhost
```

Client Interface



```
C:\Windows\system32\cmd.exe

C:\Program Files\Java\jdk1.7.0_11\bin>java -Djava.security.policy=no.policy Math
ServiceClient localhost
Enter your visiting card information
Enter your first name:
Masud
Enter your last name:
Rahman
Enter your roll number:
11130260
Enter your mobile number
01913213887
First name:Masud
Is the card signed? true
Enter your number to get factorial
10
The factorial of 10=3628800
Enter your number to check prime
13
13 is a prime number
Enter your number to get square
24
The square of 24 is =576

C:\Program Files\Java\jdk1.7.0_11\bin>_
```

Java RMI notes

- Java Object Serialization process of converting an object into a byte stream so it can be sent over a network or stored in a file.
- Parameter Marshalling & Unmarshalling
 - Marshalling - process of converting method arguments and return values into a format suitable for transmission over a network
 - Unmarshalling - process of reconstructing the received data back into Java objects..
- Object Activation
 - Singleton A single instance of the remote object serves all clients.State is shared between all clients.The object is created once and remains active.
Use Case: When multiple clients need access to shared state, like a counter or logging service.
 - Per client Each client gets a unique instance of the remote object.This allows each client to maintain its own state.Suitable when a client's data should be isolated from others.
Use Case: A shopping cart in an e-commerce system, where each user has their own cart.
 - Per call A new instance is created for each method call.No state is preserved between calls.This is best for stateless services where each request is independent.
Use Case: A calculator service where no state is needed between calls.

Strength Of Java RMI

- *Object Oriented*: Can pass complex object rather than only primitive types
- *Mobile Behavior*: Change of roles between client and server easily
- *Design Patterns*: Encourages OO design patterns as objects are transferred
- *Safe & Secure*: The security settings of Java framework used
- *Easy to Write /Easy to Use*: Requires very little coding to access service

Strengths Of Java RMI

- *Connects to Legacy Systems*: JNI & JDBC facilitate access.
- *Write Once, Run Anywhere*: 100% portable, run on any machine having JVM
- *Distributed Garbage Collection*: Same principle like memory garbage collection
- *Parallel Computing*: Through multi-threading RMI server can serve numerous clients
- *Interoperable between different Java versions*: Available from JDK 1.1, can communicate between all versions of JDKs

Weaknesses of Java RMI

- *Tied to Java System*: Purely Java-centric technology, does not have good support for legacy system written in C, C++, Ada etc.
- *Performance Issue* : Only good for large-grain computation not good for many method calling
- *Security Restrictions & Complexities*: Threats during downloading objects from server, malicious client request, added security complexity in policy file.

.NET Remoting

- .NET equivalent of Java RMI
- Uses Windows registry as the registry service
- Java RMI supports more communication protocols and .NET remoting
- .NET remoting creates proxies at runtime similar to RMI

✚ .NET Remoting (Java RMI Alternative)

.NET Remoting is the Microsoft equivalent of Java RMI, designed for distributed computing in C# and .NET environments.

◆ Key Differences:

Feature	Java RMI	.NET Remoting
Platform	Java-based	.NET-based (Windows)
Registry Service	Uses <code>rmiregistry</code>	Uses Windows Registry
Interoperability	Java-only	Works with multiple .NET languages (C#, VB.NET)
Communication Protocols	Supports multiple protocols (JRMP, IIOP)	Supports fewer protocols
Proxy Creation	Uses <code>stub & skeleton</code>	Creates proxies dynamically at runtime

Conclusion

- RMI/RPC makes it easier to build distributed systems with programmers not having to worry about network comm. (sockets, etc)
- .NET Remoting
- No interoperability (Java only due to binary messages used)

EJB / JEE
RMI / RPC
Socket programming
Network interface