# RMI

Distributed Application.pdf

```
import
Interface -import java.rmi.*;
Implementation-import java.rmi.*;
            import java.rmi.server.*;
Server-import java.rmi.registry.*;
Client-import java.rmi.registry.*;
     import java.util.*;
```

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.
● The RMI provides remote communication between the applications using two objects stub and skeleton.
● Core feature of Java’s distributed computing model

## Benefits of RMI (Remote Method Invocation):

- **Simplified Remote Communication**: Allows seamless communication between Java objects running in different JVMs over a network as if they were in the same JVM.
- **Object-Oriented Approach**: Supports method calls on remote objects, maintaining an object-oriented design.
- **Built-in Java Security**: Leverages Java's built-in security features for secure communication between client and server.

- **Automatic Stub Generation**: Simplifies development with automatic generation of stubs and skeletons.
- **Platform Independence**: Works across platforms as it is written entirely in Java.
- **Built-in Serialization**: Handles object serialization for transmitting objects between JVMs.
- **Dynamic Class Loading**: Supports dynamic loading of classes from the server to the client.
- **Scalability**: Facilitates building distributed applications that can scale across multiple machines.
- **Automatic Garbage Collection**: RMI integrates with Java's automatic garbage collection, ensuring that remote objects no longer in use are cleaned up efficiently, just like local objects, reducing memory management overhead.

## Limitations of RMI (Remote Method Invocation):

- **Java-only**: RMI is specific to Java, making it incompatible with systems or applications written in other programming languages.
- **Complexity**: Setting up and managing RMI can be more complex compared to simpler communication protocols like HTTP or REST.
- **Performance Overhead**: Involves additional overhead for serialization, deserialization, and communication, which may impact performance.
- **Tight Coupling**: RMI tightly couples the client and server, requiring both to use Java and share compatible interfaces.
- **Firewall Issues**: RMI uses dynamic port allocation, which may cause difficulties when communicating across firewalls or NAT.
- **Limited Scalability**: Not ideal for high-load, large-scale distributed systems compared to modern frameworks like gRPC or RESTful services.
- **Deprecation Risk**: As newer technologies emerge, RMI is less commonly used and may not receive significant updates in the future.

Here is the revised list of RMI limitations including **latency**:

1. **Java-Only**: RMI is specific to Java, making it incompatible with systems or applications written in other programming languages.
2. **Serializability**: Objects passed between the client and server must implement `Serializable`, which means only serializable objects can be transferred over jvm .
3. **Large-Scale Project Limitations**: RMI can struggle with scalability in large systems due to challenges with managing numerous remote objects, network complexity, and maintaining system-wide performance.
4. **Performance Overhead**: RMI introduces performance overhead due to object serialization, deserialization, and the creation of stubs and skeletons, making it less efficient than local calls.
5. **Latency**: Network latency can significantly affect RMI performance, as remote method calls are subject to the inherent delays in communication across networks.
6. **Complex Error Handling**: RMI requires careful handling of network failures, remote exceptions, and distributed garbage collection issues, which can complicate the development process.

## Difference Between `bind()` and `rebind()` in RMI:

| **Aspect** | `bind()` | `rebind()` |
|---|---|---|
| **Functionality** | Binds a remote object to a name in the RMI registry. | Replaces an existing binding with a new one, or creates it if it doesn't exist. |
| **Behavior** | Throws an exception if the name is already bound. | Overwrites the binding if the name already exists. |
| **Use Case** | Used when the binding name is new and not already in use. | Used when the binding name may already exist or to update it. |
| **Exception Handling** | Throws `AlreadyBoundException` if the name is already bound. | No such exception is thrown, as it overwrites the binding. |
| **Example Code** | `Naming.bind("serviceName", remoteObject);` | `Naming.rebind("serviceName", remoteObject);` |

## Summary:

- Use `bind()` to ensure that the name being bound is unique in the registry.

- Use `rebind()` if you want to update or replace an existing binding without throwing an exception.

# Distributed Application

The term distributed applications, is used for applications that require two or more autonomous
processes to cooperate in order to run them.
● A distributed application consists of one or more local or remote clients that communicate with one or more servers on several machines linked through a network.

## Application architecture

● Single tier
○
A single-tier application is an application where the  Presentation layer, Logical Layer
and Data Layer lies in the same machine.
● 2 tier
○
A software architecture in which a presentation layer runs on a client and Business Layer and Data Layer runs on a server.
● 3 tier
○ Presentation Layer, Business Layer and Data Layer runs in different machines
● N tier
○ There can be multiple middle layer for  multiple purpose.

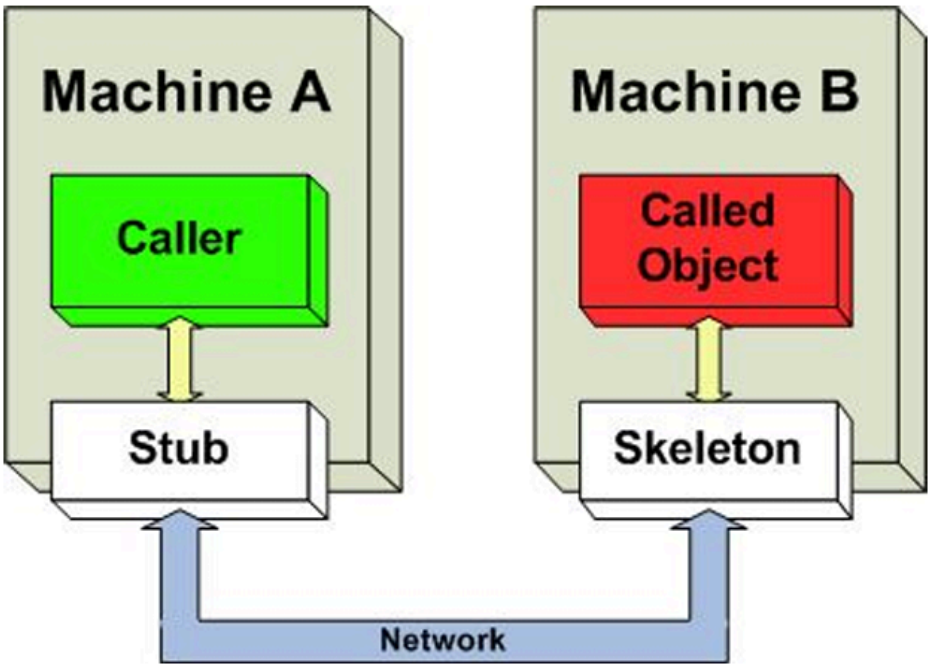
## Distributed Object

● Distributed objects are objects in a distributed computing environment that allow multiple systems or devices to interact with them as if they were local objects

● A distributed object behaves like a regular object, but the underlying infrastructure allows it to be used across different machines over a network.

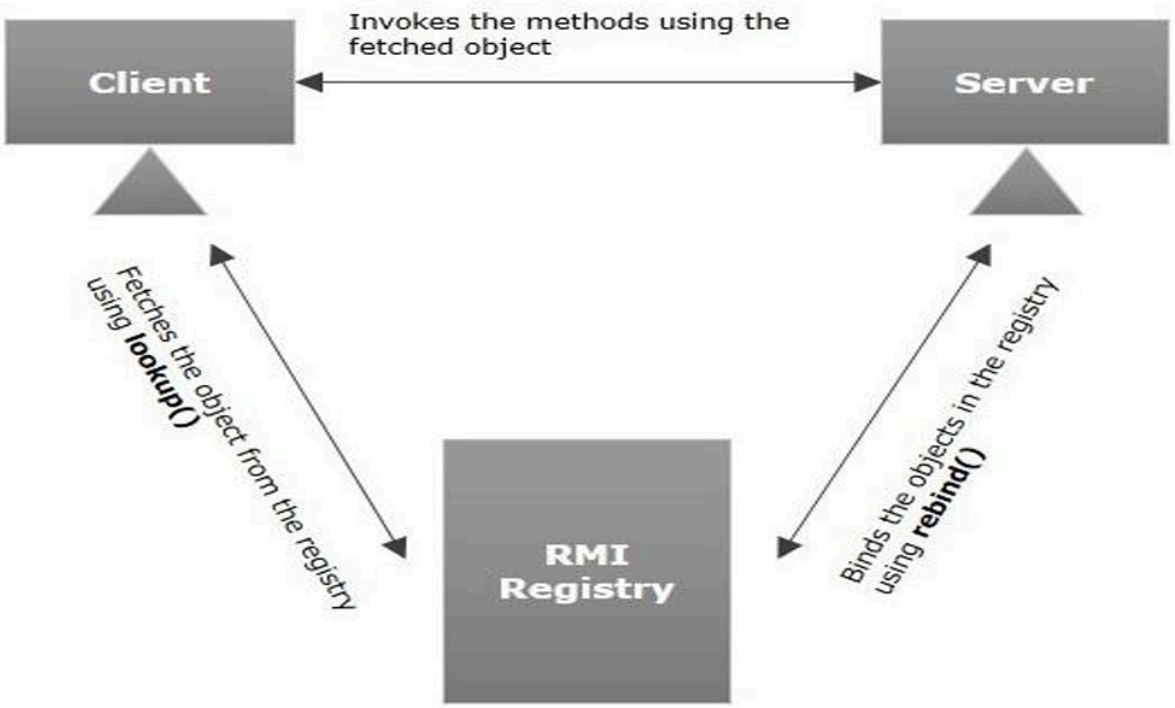
## How Distributed Objects Work:

● Client-Server Communication:
○ A client calls methods on an object as if it were local, but the object resides on a remote
server.
○ The middleware handles the communication between the client and the server.
● Object References:
○ A distributed object reference allows a client to communicate with an object located elsewhere.
○ The reference is usually created via a registry or directory service.
● Stubs and Skeletons:
○ When a client calls a method on a remote object, a stub (proxy object) on the client side
forwards the call to the skeleton (server-side object) on the remote machine.
○ The skeleton processes the call and sends the result back to the client through the stub.




## RMI Registry

● RMI registry is a namespace on which all server objects are placed.
● Each time the server creates an object, it registers this object with the RMIregistry (using bind() or reBind() methods). These are registered using a unique name known as bind name.
● To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using lookup() method)

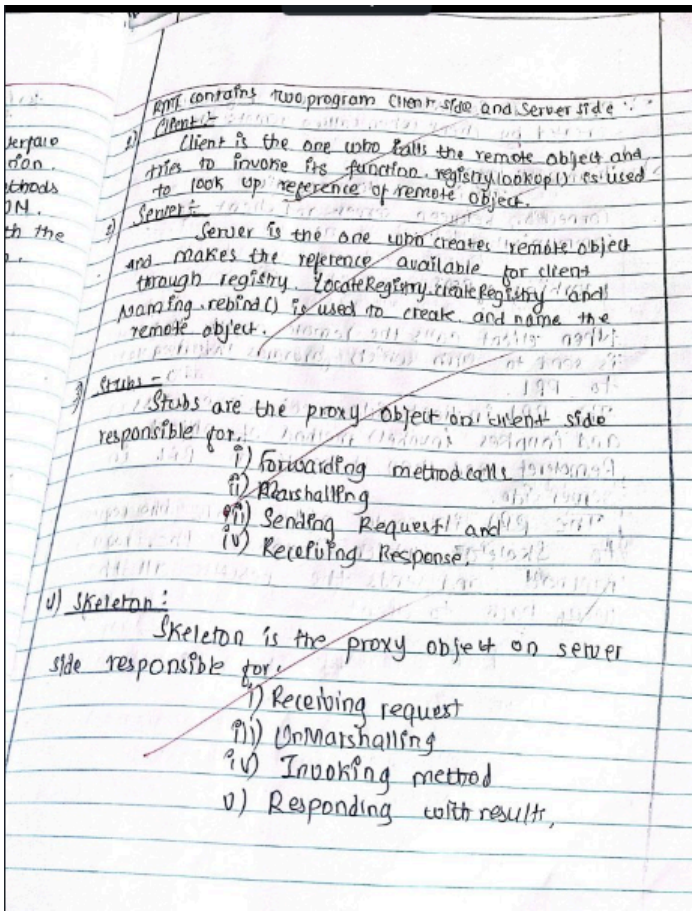



## ● Creating Server

- Step 1: Create interface that extends Remote
- Step 2: Provide implementation that extends UniCastRemoteObject
- Step 3: Create Server class with main method
- Step 4 Create registry using: LocateRegistry.createRegistry(1099)
- Step 5: Add the remote object to registry using Naming.rebind("name", object);

## ● Creating Client

- Step 1: Create Client class with main method
- Step 2: Create Registry object using LocateRegistry.getRegistry(host, port);
- Step 3: Get object using registry.lookup("object name");
- Step 4: Typecast the object to Interface and invoke the method required



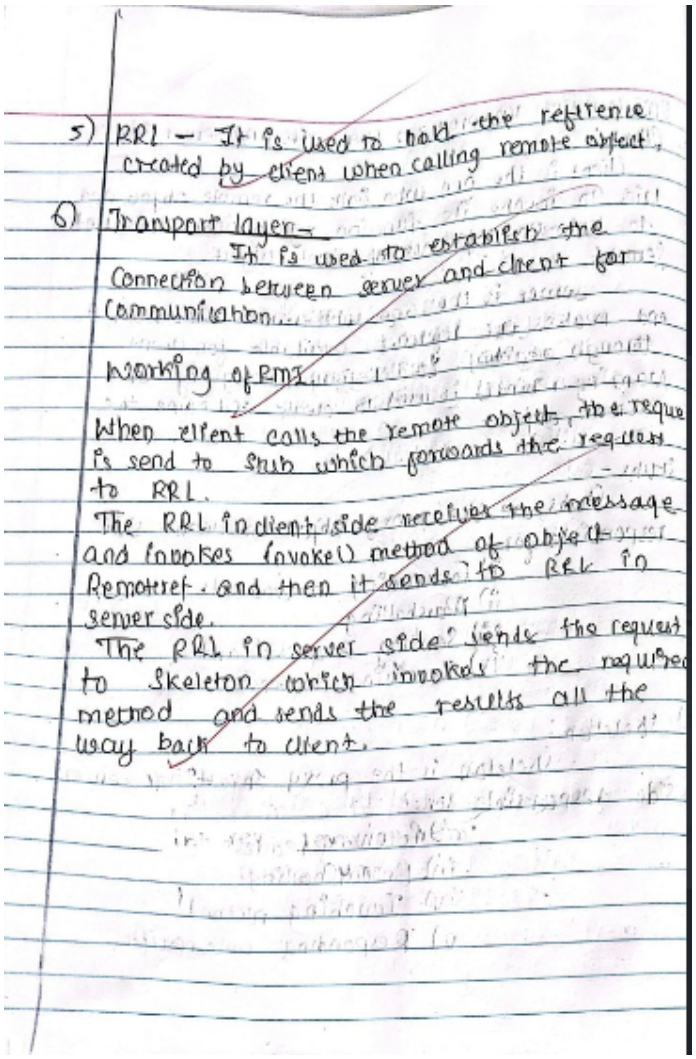
RMI contains two program client side and server side :-

1) Client :-
Client is the one who calls the remote object and tries to invoke its function. registry.lookup() is used to look up reference of remote object.

2) Server :-
Server is the one who creates remote object and makes the reference available for client through registry. LocateRegistry.createRegistry and Naming.rebind() is used to create and name the remote object.

3) Stubs :-
Stubs are the proxy object on client side responsible for.
i) Forwarding method calls
ii) Marshalling
iii) Sending Request and
iv) Receiving Response.

4) Skeleton :
Skeleton is the proxy object on server side responsible for.
i) Receiving request
ii) UnMarshalling
iv) Invoking method
v) Responding with result.




Example of RMI:

```
package Rmi;

import java.rmi.*;
public interface Pal extends Remote {
   public String reverse(String num) throws RemoteException;
}
```

```
package Rmi;
import java.rmi.*;
import java.rmi.server.*;
```

```java
public class PalImpl extends UnicastRemoteObject implements Pal {
   PalImpl() throws RemoteException{
      super();
   }
   public String reverse(String num){
      StringBuilder obj = new StringBuilder(num);
      return obj.reverse().toString();
   }
}
```

```java
package Rmi;
import java.rmi.registry.*;
import java.util.*;
import java.rmi.*;

public class PalClient {
   public static void main(String[] args) {
      try {
         Registry  rg = LocateRegistry.getRegistry("localhost", 9999);
         Scanner sc = new Scanner(System.in);
         System.out.println("Enter a number :");
         String num = sc.nextLine();
         System.out.println("Rmi.Number :"+ num);
         Pal remote = (Pal)rg.lookup("reverse");
         String rev = remote.reverse(num);
         System.out.println("Reverse : "+ rev);
      } catch (Exception e) {
         throw new RuntimeException(e);
      }
   }
```

```
}
```

```
package Rmi;
import java.rmi.registry.*;
public class PalServer {
   public static void main(String[] args){
      try{
         PalImpl obj = new PalImpl();
         Registry rg = LocateRegistry.createRegistry(9999);
         rg.rebind("reverse",obj);
         System.out.println("Server is running");
      }catch(Exception e){
         System.out.println(e);
      }
   }
}
```

In Java RMI (Remote Method Invocation), **stubs** and **skeletons** play a critical role in facilitating communication between the client and the remote server. Here's a detailed explanation of both, along with an example.

## 1. Stubs (Client-Side Proxy):

A **stub** is a client-side proxy or representation of the remote object. It acts as a gateway for the client to access the remote server object as if it were a local object. The stub hides the underlying network communication details and manages the invocation of remote methods.

## What the Stub Does:

- Receives method calls from the client (just like a local object).
- Marshals (serializes) the method parameters into a format that can be sent across the network.
- Sends the serialized request to the server.

- Waits for a response from the server, unmarshals (deserializes) the returned result, and passes it back to the client.

## 2. Skeleton (Server-Side Dispatcher):

A **skeleton** is a server-side component that receives requests from the stub, processes them, and invokes the appropriate method on the remote object. However, starting from Java 2 (JDK 1.2), skeletons are no longer required as they have been replaced by reflection-based mechanisms. But in older versions, the skeleton was responsible for the following tasks.

## What the Skeleton Does:

- Receives the request from the client stub.
- Unmarshals the method parameters.
- Invokes the appropriate method on the actual remote object.
- Marshals the result and sends it back to the stub.

## How the Interaction Works (Pre-Java 2):

1. The client invokes a method on the stub.
2. The stub marshals the method call and sends it to the skeleton.
3. The skeleton unmarshals the request and invokes the method on the server-side object.
4. The result is marshaled by the skeleton and sent back to the stub.
5. The stub unmarshals the response and returns it to the client.

## Example in RMI:

Let's consider a simple RMI example where a client requests a remote method `addNumbers(int a, int b)` from a server.

## 1. Define the Remote Interface

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

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

## 2. Implement the Remote Interface on the Server

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

public class CalculatorImpl extends UnicastRemoteObject implements Calcula
tor {

    // Constructor
    protected CalculatorImpl() throws RemoteException {
        super();
    }

    // Implement the remote method
    @Override
    public int addNumbers(int a, int b) throws RemoteException {
        return a + b;
    }
}
```

## 3. Server Program (Registers the Remote Object)

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

public class RMIServer {
    public static void main(String[] args) {
        try {
            // Create an instance of the remote object
            Calculator calculator = new CalculatorImpl();
```

```
            // Bind the remote object in the registry
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.rebind("CalculatorService", calculator);

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

## 4. Client Program

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

public class RMIClient {
    public static void main(String[] args) {
        try {
            // Get the registry from the server
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);

            // Look up the remote object
            Calculator calculator = (Calculator) registry.lookup("CalculatorServic
e");

            // Call the remote method
            int result = calculator.addNumbers(5, 10);
            System.out.println("Result of addNumbers(5, 10): " + result);

        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }
}
```

## How Stubs and Skeletons Work in this Example:

- **Client Side (Stub)**:
  - When the client calls `calculator.addNumbers(5, 10)`, the **stub** (which is automatically generated) handles the network communication. It serializes the method parameters (`5` and `10`) and sends the request to the server.
- **Server Side (Skeleton)**:
  - The server receives the request, and the **skeleton** (in pre-Java 2) would deserialize the parameters and invoke the `addNumbers` method on the actual `CalculatorImpl` object. After execution, it sends the result (`15`) back to the client.
  - In modern Java (post-Java 2), there's no explicit skeleton, but the server runtime uses reflection to accomplish this task without the need for separate skeleton code.

## Post-Java 2: No Skeletons

In Java 2 and later versions, **skeletons were replaced** by dynamic proxy generation on the server side, reducing complexity. Instead of requiring a separate skeleton class, the Java RMI runtime uses reflection to dispatch method calls on the server. However, **stubs** still exist and are used for client-side communication.

## Stub Generation:

In older versions, stubs and skeletons were generated using the `rmic` (RMI compiler) tool. For example:

```
rmic CalculatorImpl
```

However, in modern versions of Java (Java 5 and later), **dynamic proxies** make it unnecessary to generate stub classes explicitly. Stubs are dynamically created at runtime.

## Summary:

- **Stubs** act as the client-side proxy, handling network communication and marshaling/unmarshaling data.
- **Skeletons** were used to dispatch remote method calls to the server object, but were removed in Java 2 and replaced by dynamic proxy mechanisms.

Here's a comparison between **RMI (Remote Method Invocation)** and **CORBA (Common Object Request Broker Architecture)** in a tabular form:

| Feature | **RMI** | **CORBA** |
|---|---|---|
| **Language Dependency** | Java-specific; both client and server must be in Java | Language-independent; supports multiple languages like C++, Java, Python, etc. |
| **Communication Protocol** | Uses JRMP (Java Remote Method Protocol) | Uses IIOP (Internet Inter-ORB Protocol) for communication |
| **Object Passing** | Passes objects by reference or by value (Java objects) | Passes objects by value using IDL (Interface Definition Language) |
| **Stubs and Skeletons** | Stubs generated automatically; skeletons removed after Java 2 | Requires both client-side stubs and server-side skeletons |
| **Interoperability** | Limited to Java-to-Java communication | High interoperability across different platforms and languages |
| **Security** | Built-in Java security manager | Customizable security models; more flexible but more complex |
| **Ease of Use** | Easier for Java developers as it integrates naturally with Java | More complex due to the need for IDL and language mapping |
| **Object Model** | Purely object-oriented (Java's object model) | Supports both procedural and object-oriented paradigms |
| **Performance Overhead** | Lesser overhead in Java-based systems (simpler setup) | Higher due to language neutrality and more general architecture |
| **Portability** | Portable within Java environments | Highly portable across multiple platforms and languages |

| | | |
|---|---|---|
| **Naming Service** | Uses Java RMI Registry for naming and locating objects | Uses CORBA Naming Service (part of its standard services) |
| **Garbage Collection** | Integrated with Java's automatic garbage collection | No automatic garbage collection; manual memory management needed |
| **Concurrency Handling** | Managed within the Java environment | Requires explicit concurrency handling, varies by implementation |
| **Middleware Complexity** | Simpler to set up, especially for Java applications | More complex setup due to support for multiple languages and platforms |
| **Adoption** | Mostly used in Java-based applications | Widely adopted in systems requiring cross-language communication |
| **IDL (Interface Definition Language)** | Not required in RMI (uses native Java interfaces) | Requires IDL to define the interfaces, which are then mapped to various programming languages |

## Summary:

- **RMI** is simpler and more tightly integrated with Java, making it ideal for Java-only applications.
- **CORBA** is language-agnostic and supports more complex, distributed systems that involve multiple programming languages, but at the cost of added complexity.

Here's a comparison highlighting the key differences between **RMI (Remote Method Invocation)** and **Socket Programming** in terms of their functionality and use cases:

| Feature | **RMI (Remote Method Invocation)** | **Socket Programming** |
|---|---|---|
| **Abstraction Level** | Higher-level abstraction; allows invoking methods on remote objects, similar to local method calls | Lower-level abstraction; requires manually managing communication between client and server through sockets |
| **Communication Model** | Object-oriented model, where methods on remote objects can be called directly | Stream-oriented (TCP) or message-oriented (UDP) communication between processes |

| | | |
|---|---|---|
| **Language Dependency** | Java-specific, works only in Java environments | Language-independent; can be implemented in any language that supports sockets (e.g., Java, C++, Python) |
| **Ease of Use** | Easier to use for Java developers; handles complex details like serialization, communication, and threading automatically | Requires manual management of low-level networking details, such as creating sockets, reading/writing data streams, and handling connections |
| **Data Handling** | Supports object passing (objects are serialized and sent over the network) | Requires sending raw data (byte streams, strings, etc.), with no direct support for sending objects without manual serialization |
| **Communication Protocol** | Uses JRMP (Java Remote Method Protocol) over TCP/IP by default | Uses TCP/IP or UDP for communication, depending on socket type |
| **Scalability** | Easier to scale for object-based distributed systems | More flexible but requires manual handling of scalability (e.g., threading, load balancing) |
| **Automatic Stubs** | Stubs (proxies) are generated automatically in modern Java, simplifying remote method invocations | No automatic proxy generation; clients and servers must explicitly define how data is sent and received over sockets |
| **Method Invocation** | Allows remote method invocation (object-oriented communication) | Communicates through raw data transmission (byte or message-based communication) |
| **Error Handling** | Handles distributed computing errors such as remote exceptions and automatic retries | Requires manual handling of network errors (e.g., connection failures, timeouts) |
| **Security** | Built-in security mechanisms via Java's security manager (e.g., controlling access to resources) | Security must be implemented manually, including encryption and authentication if required |
| **Garbage Collection** | Integrated with Java’s automatic garbage collection; remote objects are cleaned up when no longer referenced | No automatic resource cleanup; sockets must be explicitly closed to free resources |

| | | |
|---|---|---|
| **Connection Management** | Automatically manages connections, object references, and method calls on remote objects | Requires explicit socket connection management, including opening, reading/writing data, and closing connections |
| **Concurrency Handling** | Java RMI provides built-in concurrency management for handling multiple client requests | Concurrency must be handled manually using threads or other mechanisms for each socket connection |
| **Use Case** | Best suited for distributed systems where Java objects and methods need to be invoked remotely | Best suited for custom communication protocols, simple client-server applications, or when low-level control over network communication is needed |
| **Interoperability** | Limited to Java-to-Java communication | Can be used for communication between systems written in different languages (e.g., a Java client with a C++ server) |
| **Performance** | Higher-level abstraction incurs performance overhead (due to serialization, deserialization, and method invocation protocols) | Generally faster due to direct access to network communication, though requires more effort to optimize |
| **Flexibility** | Less flexible due to Java-specific protocols and object-oriented model | More flexible, allowing custom protocols, cross-language communication, and fine-tuning of network operations |

## Summary:

- **RMI** provides a **higher-level** abstraction where developers can focus on calling remote methods on objects, making it simpler to use for Java-based distributed systems. It handles complex tasks like serialization, communication, and garbage collection automatically.
- **Socket Programming** offers a **lower-level** approach that requires developers to manually handle the communication between client and server. It is more flexible and language-independent, allowing custom protocols but requires more effort to manage connections, data streams, and error handling.

The choice between RMI and socket programming depends on the application requirements: **RMI** is ideal for Java-based distributed systems with remote object communication, while **socket programming** is better for scenarios that need custom, low-level control, or cross-language communication.

Here's a simple Java RMI program where the client sends two numbers to the server, and the server returns the greater of the two numbers.

## Steps to Implement:

1. Define a remote interface (`GreatestNumber`) for the RMI service.
2. Implement the remote interface on the server (`GreatestNumberImpl`).
3. Create an RMI server (`RMIServer`) that registers the remote object.
4. Create an RMI client (`RMIClient`) that sends two numbers to the server and receives the result.

---

## 1. Define the Remote Interface (`GreatestNumber.java`)

This interface will declare the remote method `getGreatest` that the client will call to get the greatest number between two integers.

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

public interface GreatestNumber extends Remote {
    // Remote method to find the greatest of two numbers
    int getGreatest(int a, int b) throws RemoteException;
}
```

## 2. Implement the Remote Interface on the Server (`GreatestNumberImpl.java`)

This class implements the `GreatestNumber` interface and defines the logic to return the greater of two numbers.

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

public class GreatestNumberImpl extends UnicastRemoteObject implements
GreatestNumber {

    // Constructor
    protected GreatestNumberImpl() throws RemoteException {
        super();
    }

    // Implement the remote method
    @Override
    public int getGreatest(int a, int b) throws RemoteException {
        return (a > b) ? a : b;
    }
}
```

### 3. Create the RMI Server ( `RMIServer.java` )

This server will register the `GreatestNumber` service in the RMI registry so that the client can lookup and invoke the service.

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

public class RMIServer {
    public static void main(String[] args) {
        try {
            // Create an instance of the remote object
            GreatestNumber greatestNumber = new GreatestNumberImpl();

            // Bind the remote object in the RMI registry
            Registry registry = LocateRegistry.createRegistry(1099); // Start RMI re
gistry on port 1099
```

```
            registry.rebind("GreatestNumberService", greatestNumber);

            System.out.println("RMI Server is running...");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

## 4. Create the RMI Client ( `RMIClient.java` )

This client connects to the RMI server, sends two numbers, and receives the greatest of the two.

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

public class RMIClient {
    public static void main(String[] args) {
        try {
            // Get the registry from the server
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);

            // Look up the remote object
            GreatestNumber greatestNumber = (GreatestNumber) registry.lookup
("GreatestNumberService");

            // Send two numbers and receive the greatest number
            int a = 10;
            int b = 20;
            int result = greatestNumber.getGreatest(a, b);

            // Display the result
            System.out.println("The greatest number between " + a + " and " + b +
" is: " + result);
        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }
}
```

---

## How to Run the Program:

1. **Compile the code**:
   Compile the code using
   `javac`.

   ```
   javac GreatestNumber.java GreatestNumberImpl.java RMIServer.java RMIClient.java
   ```

2. **Start the RMI Server**:
   Run the
   `RMIServer` to register the service in the RMI registry.

   ```
   java RMIServer
   ```

3. **Run the Client**:
   Start the
   `RMIClient` to send the two numbers and receive the result.

   ```
   java RMIClient
   ```

## Explanation:

- **`GreatestNumber.java`**: This defines the remote interface for the RMI service, which declares the method `getGreatest(int a, int b)`.
- **`GreatestNumberImpl.java`**: This is the implementation of the `GreatestNumber` interface. The `getGreatest` method is implemented to return the greater of the two integers.
- **`RMIServer.java`**: This class creates an instance of the `GreatestNumberImpl` and binds it to the RMI registry under the name `"GreatestNumberService"`. The RMI server runs

on port `1099`.

- `RMIClient.java`: This class connects to the RMI server, looks up the `"GreatestNumberService"`, sends two numbers, and receives the result.

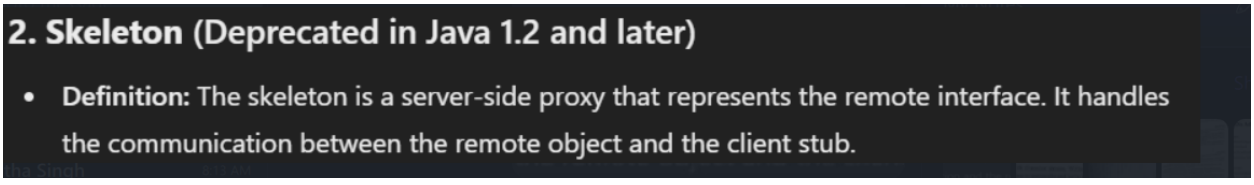
This example demonstrates how RMI can be used to implement a simple client-server communication where the server performs a task (returning the greatest number) and the client receives the result.

## Difference between Marshalling and Unmarshalling

| Aspect | Marshalling | Unmarshalling |
|---|---|---|
| **Definition** | Process of converting an object into a format suitable for storage or transmission (e.g., byte stream) | Process of converting the transmitted or stored data back into an object |
| **Direction** | Object to byte stream (serialization) | Byte stream to object (deserialization) |
| **Purpose** | Prepare data for transfer or storage | Reconstruct data after transfer or retrieval |
| **Usage in RMI** | Used when sending objects to a remote system | Used when receiving objects from a remote system |
| **Focus** | Packaging data for transmission | Extracting data from transmission |

The **this** keyword in Java refers to the current instance of the class. It is used to:

- Refer to current class instance variables
- Pass the current instance as a parameter
- Return the current class instance
- Call another constructor of the same class



## 1. Stub

- **Definition:** The stub is a client-side proxy that represents the remote object. It acts as an intermediary between the client application and the remote object on the server.