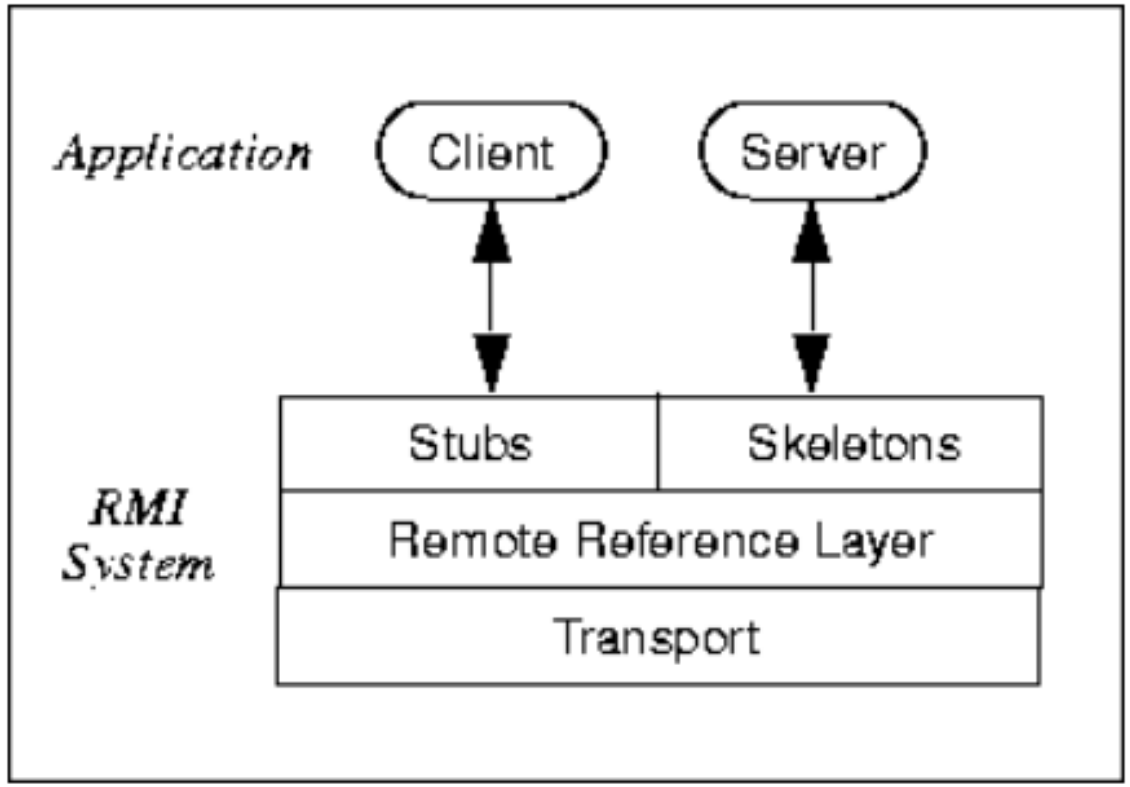
REMOTE METHOD INVOCATION(RMI)

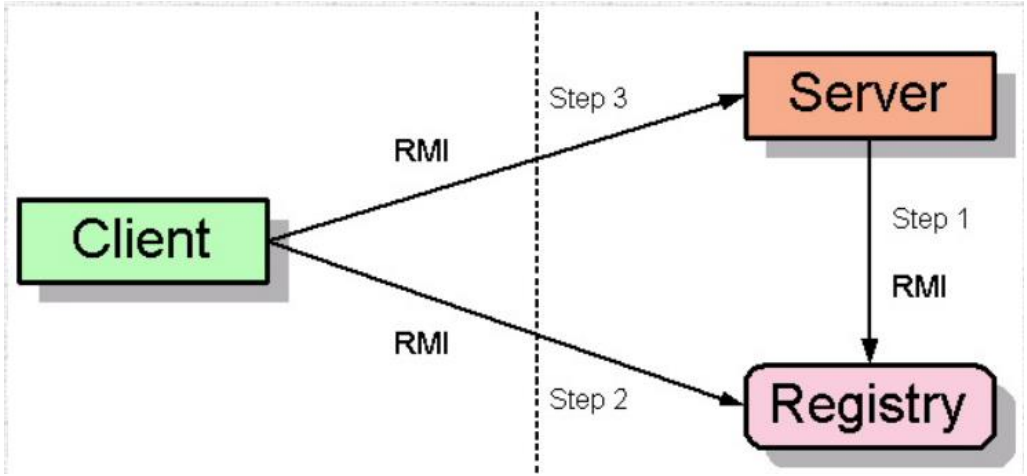
Remote Method Invocation

- RMI stands for Remote Method Invocation. It is a Java API that allows Java objects to communicate and invoke methods on remote objects **residing in different Java Virtual Machines (JVMs) over a network**. RMI provides a **mechanism for distributed computing, enabling seamless interaction** between objects running on different machines.
- **Simplified Distributed Computing**: RMI simplifies the development of distributed applications by abstracting the complexities of network communication and serialization. It allows developers to invoke methods on remote objects as if they were local objects, making **distributed computing** more accessible and intuitive.
- **Object-Oriented Communication**: RMI is based on the object-oriented paradigm, **allowing objects to be shared and accessed across different JVMs**. It enables developers to **build distributed systems by designing objects that communicate and collaborate seamlessly**.
- **Transparency**: RMI provides transparency in distributed computing. Clients can invoke methods on remote objects without worrying about the underlying network communication details. **RMI takes care of serialization**, network transmission, and method invocation, making it easier for developers to build distributed applications.
- **Dynamic Invocation**: RMI supports **dynamic method invocation**, meaning that **remote method calls can be made without the need for pre-generated stubs or proxies**. This dynamic nature of RMI allows for flexibility and adaptability in distributed systems.
- **Interoperability**: RMI is **platform-independent and can be used for communication between different systems and architectures**. It promotes interoperability by enabling Java objects to communicate with each other across different platforms, making it suitable for heterogeneous environments.
- **Java Standard**: RMI is a standard part of the **Java platform and is** well-integrated with the Java language and libraries. It provides **a robust and well-supported framework for building distributed applications in Java.**

RMI ARCHITECTURE

- It's worth noting that in modern Java versions (Java 5 and above), **the Stub and Skeleton components are generated automatically by the Java runtime** and do not need to be explicitly implemented by the developer. The stub and skeleton generation is done through the use of dynamic proxies and reflection.





REMOTE METHOD INVOCATION(RMI)

Remote Method Invocation

1.Adder.iava

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

2. AdderImp.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class AdderImp extends UnicastRemoteObject implements Adder {
    protected AdderImp() throws RemoteException {
        super();
    }
    public int add(int a, int b) throws RemoteException {
        return a + b;
    }
}
```

REMOTE METHOD INVOCATION(RMI)

Remote Method Invocation

3.RmiServer.java

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class RmiServer {
    Run | Debug
    public static void main(String[] args) {
        try {
            Adder rmiobj = new AdderImp();
            Registry registry = LocateRegistry.createRegistry(port:1099);
            registry.bind(name:"AddService", rmiobj);
            System.out.println(x:"Server started and waiting for client requests...");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

REMOTE METHOD INVOCATION(RMI)

Remote Method Invocation

4.RmiClientjava

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class RmiClient {
    Run | Debug
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry(host:"localhost", port:1099);
            Adder rs = (Adder) registry.lookup(name:"AddService");
            int num1 = 5;
            int num2 = 10;
            int sum = rs.add(num1, num2);
            System.out.println("Sum: " + sum);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```