Week 9: Remote Method Invocation (RMI)

Introduction to RMI

Remote Method Invocation (RMI) is a powerful feature in Java that enables communication between objects on different machines over a network. Simply put, it allows a Java program running on one computer (the **client**) to invoke methods on a Java object located on another computer (the **server**), as if the method were part of the local program. This makes it possible to build **distributed applications**, where different parts of an application can run on different computers, yet work together seamlessly as one unit.

Why Use RMI?

RMI simplifies the process of building distributed systems in Java by handling the underlying complexities of network communication. This allows developers to focus more on the business logic and less on managing data transfer, networking protocols, or low-level socket programming. For example, you can create an RMI application where clients request information from a server or perform complex computations remotely, all while Java abstracts away the complexities of network connections.

Key Concepts of RMI

- **Remote Objects**: An object that resides on a remote server and whose methods can be called by a client application. These objects are created on the server side and registered with a special service called the **RMI registry**, which makes them available for clients to find and invoke.
- **Stub and Skeleton** (Prior to Java 5): The **stub** is a client-side proxy object that represents the remote object. When a client calls a method on the stub, the stub forwards the request to the remote object on the server. The **skeleton** was a server-side entity that handled the request before sending it to the remote object, but this has been replaced with more modern approaches since Java 5.
- **RMI Registry**: A simple service that allows clients to look up remote objects by their names. Think of it like a telephone directory for remote objects, where clients can find and connect to the services they need.
- **Communication Flow in RMI**:
    1. **Server Side**
        - The server creates an instance of the remote object and registers it with the RMI registry, making it available for client calls.
    2. **Client Side**
        - The client looks up the remote object using the RMI registry and obtains a reference to the object (in the form of a stub).

- The client invokes methods on the stub, which forwards the request to the actual remote object on the server.

How RMI Works - A Simple Example

To better understand how RMI works, let's walk through a simple example. Suppose you have a remote service that provides greetings based on the name you provide. Here's how it works:

1. **Create a Remote Interface**: Define the methods that can be called remotely.
2. **Implement the Remote Object**: Create the actual object on the server that provides the logic for the methods defined in the interface.
3. **Register the Remote Object**: The server makes the remote object available to clients by registering it with the RMI registry.
4. **Client Side**: The client connects to the server using the RMI registry and invokes methods on the remote object.

Benefits of Using RMI

- **Platform Independence**: Since Java is platform-independent, RMI-based distributed systems can communicate across different operating systems without any modifications.
- **Simplicity**: RMI abstracts away the complexities of network communication, such as socket programming, making it easier for developers to build distributed applications.
- **Object-Oriented Approach**: Unlike lower-level networking solutions, RMI allows developers to work with objects, keeping the development process consistent with Java's object-oriented paradigm.

Real-World Applications of RMI

- **Distributed Systems**: RMI is used to create systems that are spread across different machines, such as a distributed banking application where different branches communicate with a central server.
- **Remote Services**: RMI enables the creation of remote services, such as a calculator service that clients can use to perform calculations without having the logic on their local machines.
- **Data Sharing**: Multiple clients can access shared data or services from a central server using RMI.

Step-by-Step Example: Implementing RMI

Goal: Create a simple RMI application where a client sends a request to a server to get a greeting message.

1. Create the Remote Interface: The interface defines the methods that can be called remotely.

Code (Greeting.java):

```
import java.rmi.Remote;
import java.rmi.RemoteException;
// Remote interface
public interface Greeting extends Remote {
   String sayHello(String name) throws RemoteException;
}
```

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

// Remote interface
public interface Greeting extends Remote {
    String sayHello(String name) throws RemoteException;
}
```

Explanation:

- The Greeting interface extends Remote.
- The sayHello method throws a RemoteException to handle communication issues.

2. Create the Remote Object Implementation: This class implements the remote interface and contains the logic for the remote method.

Code (GreetingImpl.java):

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

// Remote object implementation
public class GreetingImpl extends UnicastRemoteObject implements Greeting {
   // Constructor
   protected GreetingImpl() throws RemoteException {
      super();
   }

   // Implement the sayHello method
   @Override
   public String sayHello(String name) throws RemoteException {
      return "Hello, " + name + "!";
   }
```

}

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

// Remote object implementation
public class GreetingImpl extends UnicastRemoteObject implements Greeting {
    // Constructor
    protected GreetingImpl() throws RemoteException {
        super();
    }

    // Implement the sayHello method
    @Override
    public String sayHello(String name) throws RemoteException {
        return "Hello, " + name + "!";
    }
}
```

3. Create the Server: The server binds the remote object to the RMI registry.

Code (Server.java):

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

public class Server {
   public static void main(String[] args) {
      try {
         // Start the RMI registry
         LocateRegistry.createRegistry(1099);
         // Create and bind the remote object
         Greeting greeting = new GreetingImpl();
         Naming.rebind("rmi://localhost/GreetingService", greeting);
         System.out.println("Server is running...");
      } catch (Exception e) {
         System.out.println("Server exception: " + e.getMessage());
         e.printStackTrace();
      }
   }
}
```

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

public class Server {
    public static void main(String[] args) {
        try {
            // Start the RMI registry
            LocateRegistry.createRegistry(1099);
            // Create and bind the remote object
            Greeting greeting = new GreetingImpl();
            Naming.rebind("rmi://localhost/GreetingService", greeting);
            System.out.println("Server is running...");
        } catch (Exception e) {
            System.out.println("Server exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

4. Create the Client: The client looks up the remote object in the RMI registry and calls its methods.

Code (Client.java):

```
import java.rmi.Naming;
public class Client {
  public static void main(String[] args) {
    try {
      // Look up the remote object
      Greeting greeting = (Greeting) Naming.lookup("rmi://localhost/GreetingService");
      // Call the remote method
      String response = greeting.sayHello("Alice");
      System.out.println("Response from server: " + response);
    } catch (Exception e) {
      System.out.println("Client exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Lab Activity: Create an RMI-Based Distributed Application

1. **Create a Remote Interface**:
    - Define methods to perform simple arithmetic operations (e.g., addition, subtraction).

2. **Implement the Remote Object**:
    - Create a class that implements the interface and defines the logic.
3. **Create the Server**:
    - Bind the remote object to the RMI registry.
4. **Create the Client**:
    - Look up the remote object and invoke methods.

Week 10: Implementing RMI on Remote & Local Hosts

Deploying RMI Applications

In this week, we will take our understanding of **Remote Method Invocation (RMI)** further by focusing on deploying RMI applications on both **local** and **remote** hosts. This means you will learn to connect components that may be running on different machines across a network. The goal is to gain practical knowledge on how distributed systems work, leveraging RMI to build scalable applications that function seamlessly over networks.

Deploying RMI Applications on Local and Remote Hosts

1. Setting Up the Server on a Remote Host

To deploy an RMI application on a remote host, you need to ensure that the server-side setup is properly configured to allow remote access. Here are the steps and considerations to make:

- **Modify the Server Code**: In your server code, bind the remote object to the IP address of the remote host. This makes the remote object accessible to clients from other machines on the network.
    - **Example**:

```
// Server code for binding a remote object
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class RemoteServer {
   public static void main(String[] args) {
      try {
         // Start the RMI registry on port 1099
         LocateRegistry.createRegistry(1099);

         // Create and bind the remote object to the server's IP address
         MyRemoteService service = new MyRemoteServiceImpl();
         Naming.rebind("rmi://<remote-server-ip>/MyService", service);
         System.out.println("Remote server is running...");
      } catch (Exception e) {
         System.out.println("Server exception: " + e.getMessage());
         e.printStackTrace();
      }
   }
}
```

```java
// Server code for binding a remote object
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;

public class RemoteServer {
    public static void main(String[] args) {
        try {
            // Start the RMI registry on port 1099
            LocateRegistry.createRegistry(1099);

            // Create and bind the remote object to the server's IP address
            MyRemoteService service = new MyRemoteServiceImpl();
            Naming.rebind("rmi://<remote-server-ip>/MyService", service);
            System.out.println("Remote server is running...");
        } catch (Exception e) {
            System.out.println("Server exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

    - Replace <remote-server-ip> with the actual IP address of the server hosting the remote object.

- **Open Ports for RMI Communication**: Ensure that the remote host's firewall allows incoming traffic on the port used for RMI (default is **1099**). If the port is blocked, clients will be unable to connect to the server.

2. Connecting the Client to the Remote Server

On the client side, you need to connect to the remote server and access the exposed RMI service. This is done using the server's IP address.

- **Example**:

```
import java.rmi.Naming;

public class RemoteClient {
   public static void main(String[] args) {
      try {
         // Connect to the remote service
         MyRemoteService service = (MyRemoteService) Naming.lookup("rmi://<remote-server-ip>/MyService");
         // Call a method on the remote service
         String response = service.sayHello("Alice");
         System.out.println("Response from server: " + response);
      } catch (Exception e) {
         System.out.println("Client exception: " + e.getMessage());
         e.printStackTrace();
      }
```

}
}

```
import java.rmi.Naming;

public class RemoteClient {
    public static void main(String[] args) {
        try {
            // Connect to the remote service
            MyRemoteService service = (MyRemoteService) Naming.lookup("rmi://<remote-s
            // Call a method on the remote service
            String response = service.sayHello("Alice");
            System.out.println("Response from server: " + response);
        } catch (Exception e) {
            System.out.println("Client exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

- The client connects to the remote object using its network address (rmi://<remote-server-ip>/MyService). Replace <remote-server-ip> with the IP address of the remote host.

3. Deploying on a Local Host

For local deployment (e.g., running both the server and client on the same machine), the setup is simpler since network configurations such as IP addresses and open ports are less restrictive. You can bind the remote object to localhost.

- **Example of Local Deployment**:
  - **Server Side**:

Naming.rebind("rmi://localhost/MyService", service);

  - **Client Side**:

MyRemoteService service = (MyRemoteService) Naming.lookup("rmi://localhost/MyService");

This setup is useful for testing and development before moving to a networked environment.

Understanding Parameter Passing in RMI

In an RMI call, parameters are passed from the client to the server **by value**, not by reference. This means that when a client invokes a remote method and passes parameters, the values are serialized (converted into a byte stream) and sent over the network to the server. The server then deserializes these values to use them within its methods.

- **Example of Parameter Passing**:

// Remote interface

public interface Calculator extends Remote {

int add(int a, int b) throws RemoteException;

}

// Client-side call

Calculator calc = (Calculator) Naming.lookup("rmi://localhost/CalculatorService");

int result = calc.add(5, 3); // Parameters 5 and 3 are passed by value

System.out.println("Result: " + result);

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

// Client-side call
Calculator calc = (Calculator) Naming.lookup("rmi://localhost/CalculatorService");
int result = calc.add(5, 3); // Parameters 5 and 3 are passed by value
System.out.println("Result: " + result);
```

- **Serialization**: RMI uses **serialization** to convert objects and parameters into a format that can be sent over the network. This ensures that even complex data types can be transmitted between the client and server.

Example: Distributed Calculator Using RMI

Remote Interface (Calculator.java):

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

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

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

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

Remote Object Implementation (CalculatorImpl.java):

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

public class CalculatorImpl extends UnicastRemoteObject implements Calculator {
   protected CalculatorImpl() throws RemoteException {
      super();
   }

   @Override
   public int add(int a, int b) throws RemoteException {
      return a + b;
   }

   @Override
   public int subtract(int a, int b) throws RemoteException {
      return a - b;
   }
}
```


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

public class CalculatorImpl extends UnicastRemoteObject implements Calculator {
    protected CalculatorImpl() throws RemoteException {
        super();
    }

    @Override
    public int add(int a, int b) throws RemoteException {
        return a + b;
    }

    @Override
    public int subtract(int a, int b) throws RemoteException {
        return a - b;
    }
}
```


Server Code (CalculatorServer.java):

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

```
public class CalculatorServer {
   public static void main(String[] args) {
      try {
         LocateRegistry.createRegistry(1099);
         Calculator calculator = new CalculatorImpl();
         Naming.rebind("rmi://localhost/CalculatorService", calculator);
         System.out.println("Calculator Server is running...");
      } catch (Exception e) {
         System.out.println("Server exception: " + e.getMessage());
         e.printStackTrace();
      }
   }
}
```

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

public class CalculatorServer {
    public static void main(String[] args) {
        try {
            LocateRegistry.createRegistry(1099);
            Calculator calculator = new CalculatorImpl();
            Naming.rebind("rmi://localhost/CalculatorService", calculator);
            System.out.println("Calculator Server is running...");
        } catch (Exception e) {
            System.out.println("Server exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Client Code (CalculatorClient.java):

```
import java.rmi.Naming;

public class CalculatorClient {

   public static void main(String[] args) {

      try {

         Calculator calculator = (Calculator) Naming.lookup("rmi://localhost/CalculatorService");

         int resultAdd = calculator.add(5, 3);
```

```
            int resultSubtract = calculator.subtract(10, 4);
            System.out.println("Addition Result: " + resultAdd);
            System.out.println("Subtraction Result: " + resultSubtract);
        } catch (Exception e) {
            System.out.println("Client exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

```java
import java.rmi.Naming;

public class CalculatorClient {
    public static void main(String[] args) {
        try {
            Calculator calculator = (Calculator) Naming.lookup("rmi://localhost/Calculator
            int resultAdd = calculator.add(5, 3);
            int resultSubtract = calculator.subtract(10, 4);
            System.out.println("Addition Result: " + resultAdd);
            System.out.println("Subtraction Result: " + resultSubtract);
        } catch (Exception e) {
            System.out.println("Client exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Exercises for Practice

1. **Exercise 1: Create a Remote Interface** (15 Points)
    - Define a remote interface for performing string operations (e.g., concatenation).
    - Implement the methods in a remote object.
2. **Exercise 2: Deploy the Application on Different Hosts** (15 Points)
    - Modify your RMI application to work across different hosts (remote machines).
    - Test the client-server communication using different IP addresses.