
**TRIBHUVAN UNIVERSITY**

**INSTITUTE OF ENGINEERING**

**THAPATHALI CAMPUS**

**A Lab Report**

**On**

# Implementation of RMI in Java

**(Distributed System Lab:2)**

**Submitted By:**

Sajjan Acharya

(THA076BCT038)

**Submitted To:**

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

June, 2023

## Theory

Remote Method Invocation (RMI) is a Java-based technology that enables communication between distributed objects in a networked environment. RMI allows a Java object running on one machine to invoke methods on a remote Java object residing on a different machine, as if it were a local method call.

The fundamental concept behind RMI is the idea of object serialization, which is the process of converting an object into a byte stream that can be transmitted over the network. RMI uses Java's built-in object serialization mechanism to achieve this. When a remote method is invoked, the parameters and the return value (if any) are automatically serialized and transmitted to the remote object.

To establish a connection between the client and the server, RMI relies on the Java Remote Object Activation Service (Java RMI Registry). The registry acts as a lookup service, allowing clients to locate remote objects by their registered names.

## Background

The importance of RMI lies in its ability to simplify the development of distributed systems. By providing a seamless way to interact with remote objects, RMI enables developers to build distributed applications in Java with ease. RMI abstracts the complexities of network communication, allowing developers to focus on the logic of their applications rather than low-level networking details. It promotes code reusability and modularity by enabling objects to be distributed across multiple machines while maintaining a consistent programming model.

Moreover, RMI fosters the concept of distributed computing, allowing for the creation of scalable and flexible systems. It supports the development of client-server architectures, where clients can remotely invoke methods on server-side objects, enabling distributed processing and resource sharing. RMI plays a vital role in distributed systems such as enterprise applications, where components may be distributed across multiple servers, providing seamless communication and collaboration between these components.

Remote Interface: The remote interface defines the methods that can be invoked remotely. It acts as a contract between the client and server, specifying the methods that the client can call on the remote object. Both the client and server must have access to the remote interface. It serves as a contract defining the methods that can be invoked remotely, ensuring that both the client and server agree on the available methods and their signatures.

Remote object: The remote object is the implementation of the remote interface. It resides on the server and provides the actual implementation of the remote methods. The remote object must extend the '*java.rmi.server.UnicastRemoteObject'* class to enable remote method invocation. It is the actual implementation of the remote interface residing on the server, responsible for executing the logic behind the remote method calls and providing the desired functionality to the clients.

RMI Registry: The RMI Registry is a lookup service that allows clients to locate remote objects by their registered names. It acts as a centralized registry where the server binds its remote objects, making them available for clients to find and invoke. It acts as a centralized directory where the server binds its remote objects, allowing clients to locate and access the remote objects by their registered names, simplifying the process of object discovery and invocation.

Stub and Skeleton: The stub and skeleton are automatically generated by the RMI compiler (RMIC) based on the remote interface and the remote object. The stub acts as a client-side proxy, intercepting the method calls and forwarding them to the server. The skeleton resides on the server and receives the method calls from the stub, invoking the corresponding methods on the remote object.

Object Serialization: Object serialization is the process of converting an object into a byte stream that can be transmitted over the network. RMI uses Java's built-in object serialization mechanism to serialize method parameters and return values when invoking remote methods. It is a fundamental aspect of RMI, allowing Java objects to be converted into byte streams for transmission over the network. By leveraging Java's object serialization, RMI enables the transparent transmission of method parameters

and return values, ensuring the integrity and consistency of data across distributed objects.

**Algorithm**

1. Define a remote interface.
2. Implement the remote interface.
3. Compile the interface and implementation.
4. Start the RMI Registry.
5. Instantiate the remote object on the server.
6. Bind the remote object to a name in the RMI Registry.
7. Obtain a reference to the remote object from the RMI Registry on the client.
8. Cast the remote object reference to the remote interface type.
9. Invoke remote methods on the client-side interface.
10. Handle remote invocation exceptions as needed.

**Code/Implementation**

In this lab, a simple function of multiplication of two numbers was invoked from another object to demonstrate the flexibility and functionality of RMI that is considered a useful boon in the concept of distributed systems. Multiplication of '12' and '3' were carried out and it was displayed to another object when it was called.

Code for remoteinterface.java:

```
1  import java.rmi.Remote;
2  import java.rmi.RemoteException;
3
4  public interface remoteinterface extends Remote{
5      // remote methods that can be invoked by clients
6       public String sayHello() throws RemoteException;
7      public int mul(int x, int y)throws RemoteException;
8  }
9
10
11
```

Code for remoteimplement.java:

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

public class remoteimplement extends UnicastRemoteObject implements remoteinterface{
    public remoteimplement() throws RemoteException{

    }
     public String sayHello() throws RemoteException {
        return "Remote server is now on!!!!!";
    }
public int mul(int x, int y) throws RemoteException{
    return x*y;
}
}
```

Code for RMIServer.java:

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

public class rmiserver {
public static  void main(String[] args) {
    try{
        remoteinterface remoteInterface = new remoteimplement();

        Registry registry = LocateRegistry.createRegistry(1099);

        registry.rebind("MulService", remoteInterface);

       System.out.println("Server is now on, the requests from client shall be received!");
    }
    catch (Exception e){
        e.printStackTrace();
    }
}
}
```

Code for RMIClient.java:

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class rmiclient {
    public static  void main(String[] args) {
    try{

        Registry registry = LocateRegistry.getRegistry("localhost", 1099);
        remoteinterface remoteInterface = (remoteinterface) registry.lookup("MulService");
         String message = remoteInterface.sayHello();
            System.out.println("Server response: " + message);
         int result = remoteInterface.mul(12, 3);
        registry.rebind("MulService", remoteInterface);
       System.out.println("Result is :" + result);
    }
    catch (Exception e){
        e.printStackTrace();
    }

}
}
```

**Results**

Client Side:

```
Server response: Remote server is now on!!!!!
Result is :36
```

In the server:

```
Server is now on, the requests from client shall be received!
```

**Discussion**

Concept of RMI was demonstrated in the lab session in the Java programming language. During the implementation, it was noticed that RMI is tightly coupled with the Java language which can limit the interoperability of it with other systems that may be in other languages. Also, further dependency on the serialization of Java might induce further issues with compatibility during the use of serialized objects of different versions. Despite them, the implementation provided an effective means of distributed object communication. The methods were invoked from another objects remotely which was the major objective of the experiment.