RMI

Mahboob Ali

What is RMI?

Definition: Java Remote Method Invocation (RMI) system allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine.

- RMI provides a framework for building *distributed Java Systems*.

Distributed Systems: is a program or a set of programs that runs on more than one computing resources.

RMI Basics

- RMI is a higher-level API built on top of sockets.
- Socket-level programming allows you to pass data through sockets among computers.
- RMI enables you not only to pass data among objects on different systems, but also to invoke methods in a remote object.

Difference between RPC and RMI

- RMI is similar to Remote Procedure Calls (RPC) in the sense that both RMI and RPC enable you to invoke methods, but with some main differences.
- With RPC, you call a standalone procedure.
- With RMI, you invoke a method within a specific object. RMI can be viewed as object-oriented RPC.

RMI Application

- Often comprise of two separate programs:
  - Server program.
  - Client program.

Server program

- A typical server program
  - Creates remote objects
  - Makes references to these objects accessible
  - Waits for clients to invoke methods on these objects.

Client program

- A typical client program
  - obtains a remote reference to one or more remote objects on a server.
  - Invokes methods on them.

Distributed Object Application

- RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

Distributed object application steps

Step 1

- **Locate remote objects:** Applications can use various mechanisms to obtain references to remote objects.

For example, an application can register its remote objects with RMI's simple naming facility, the **RMI registry**. Alternatively, an application can pass and return remote object references as part of other remote invocations.

Step 2

- **Communicate with remote objects:** Details of communication between remote objects are handled by RMI.

To the programmer, remote communication looks similar to regular Java method invocations.

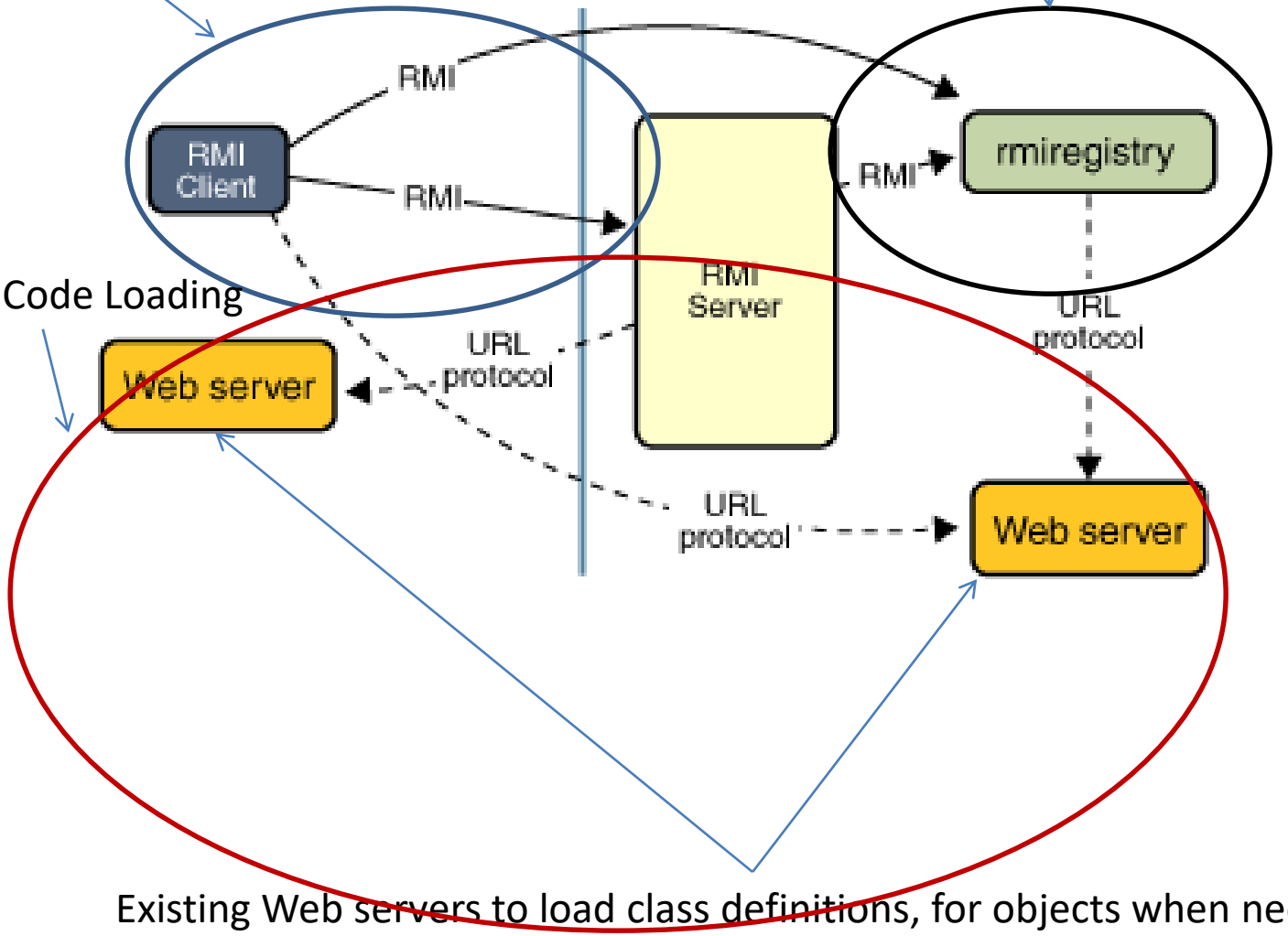
Step 3

- **Load class definitions for objects that are passed around:**

Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

Client looks up the remote object by its name in the server's registry and then invokes a method on it.

Server calls the registry to associate (or bind) a name with a remote object



Dynamic Code Loading

Existing Web servers to load class definitions, for objects when needed

Dynamic code loading

- Previously:
  - All of the types and behavior of an object, was available only in a single Java virtual machine.
- One of the central and unique features of RMI is:
  - its ability to download the definition of an object's class if the class is not defined in the receiver's Java virtual machine.

- RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another Java virtual machine.
- This capability enables *new types and behaviors* to be introduced into a remote Java virtual machine, thus dynamically extending the behavior of an application.
- The *compute engine* example in this trail uses this capability to introduce new behavior to a distributed program.

Remote Interfaces, Objects and Methods

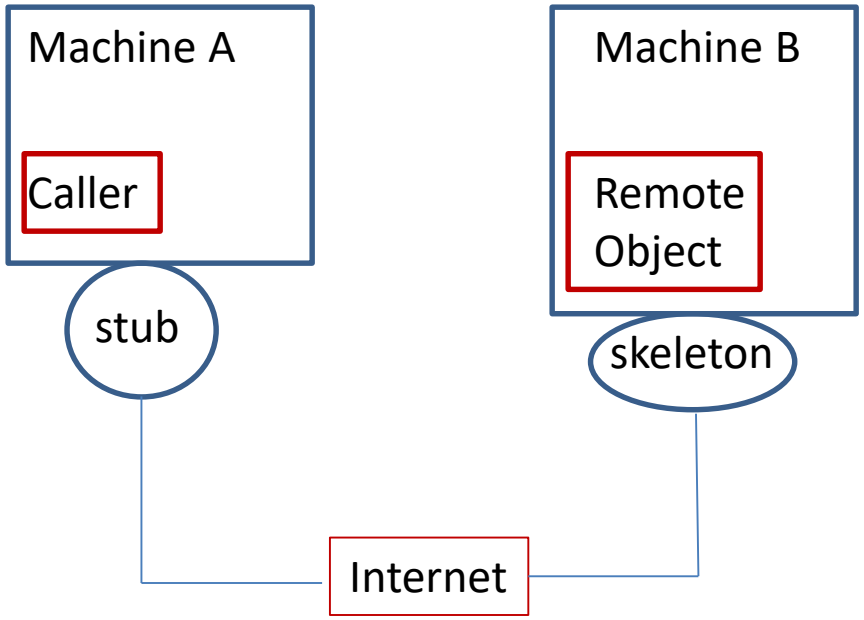
- The interfaces declare methods.
- The classes implement the methods declared in the interfaces and can also declare additional methods as well.
- In a distributed application, some implementations might reside in some Java virtual machines but not others.

Remote Interface

- Remote Interface
  - An object becomes remote by implementing a *remote interface.*
- Characteristics
  - A remote interface **extends** the interface *java.rmi.Remote*.
  - **Each method** of the interface declares *java.rmi.RemoteException*

How does the communication happens?

- RMI provides communication between the applications using two objects ***stub*** and ***skeleton.***



stub

- **Stub:** is an object, acts as a gateway for the client side.
  - All the outgoing requests are routed through it.
  - It resides at the client side and represents the remote object.
  - When the caller invokes method on the stub object, it does the following tasks:
    - It initiates a connection with remote Virtual Machine (JVM).
    - It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM).
    - It waits for the result.
    - It reads (unmarshals) the return value or exception.
    - Finally, returns the value to the caller.

skeleton

- **skeleton** is an object, acts as a gateway for the server side object.
  - All the incoming requests are routed through it.
  - When the skeleton receives the incoming request, it does the following tasks:
    - It reads the parameter for the remote method.
    - It invokes the method on the actual remote object.
    - It writes and transmits (marshals) the result to the caller.

Passing Parameters

- When a client invokes a remote method with parameters, passing parameters are handled under the cover by the stub and the skeleton. Let us consider three types of parameters:
  - Primitive data types.
  - Local object type.
  - Remote object type.

Primitive data type

- Primitive data type. A parameter of primitive type such as char, int, double, and boolean is passed by value like a local call.

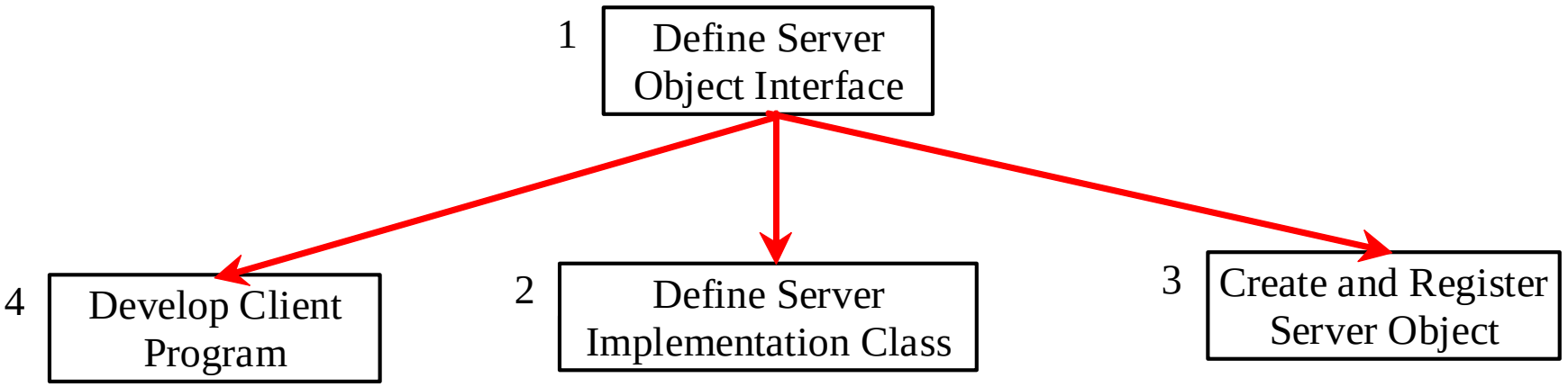
Local object type

- A parameter of local object type such as java.lang.String is also passed by value.
- Which is completely different from passing object parameter in a local call. In a local call, an object parameter is passed by reference, which corresponds to the memory address of the object.
- In a remote call, there is no way to pass the object reference because the address on one machine is meaningless to a different Java VM.
- Any object can be used as a parameter in a remote call as long as the object is serializable (marshals). The stub serializes the object parameter and sends it in a stream across the network. The skeleton deserializes (unmarshals) stream into an object.

Remote object type.

- Remote objects are passed differently from the local objects. When a client invokes a remote method with a parameter of some remote object type, the stub of the remote object is passed. The server receives the stub and manipulates the parameter through the stub.
- Passing a remote object by reference means that any changes made to the state of the object by remote method invocations are reflected in the original remote object.

Creating Distributed Applications by Using RMI



Designing and Implementing the Application Components

Determining your application architecture, including which components are local objects and which components are remotely accessible. This step includes:

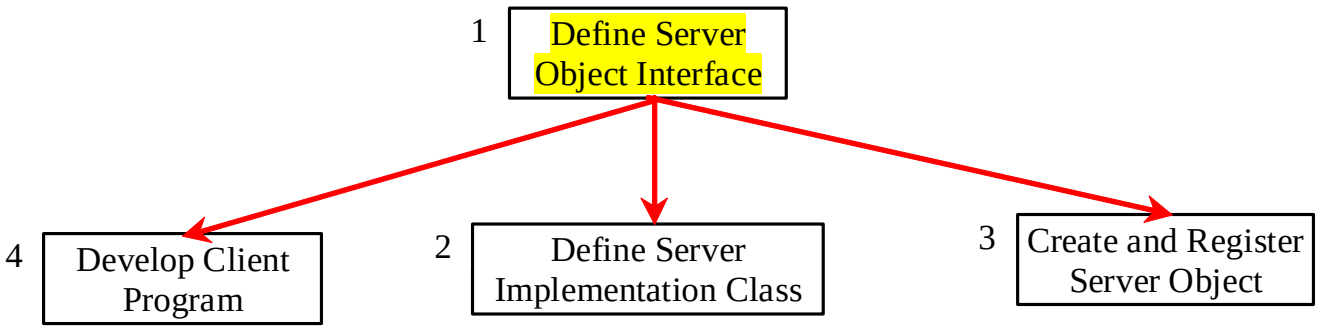
- Defining the remote interfaces (Server Object Interface).
- Implementing the remote objects (Server Implementation class).
- Implementing the clients (Client Program).
- Create and register server objects (Registry).

Define Server Object Interface

- A remote interface specifies the methods that can be invoked remotely by a client.

```
public interface ServerInterface extends Remote{
      public void service1(…) throws RemoteException;
      //other methods
}
```

- A server object interface must extend the **java.rmi.Remote** interface.

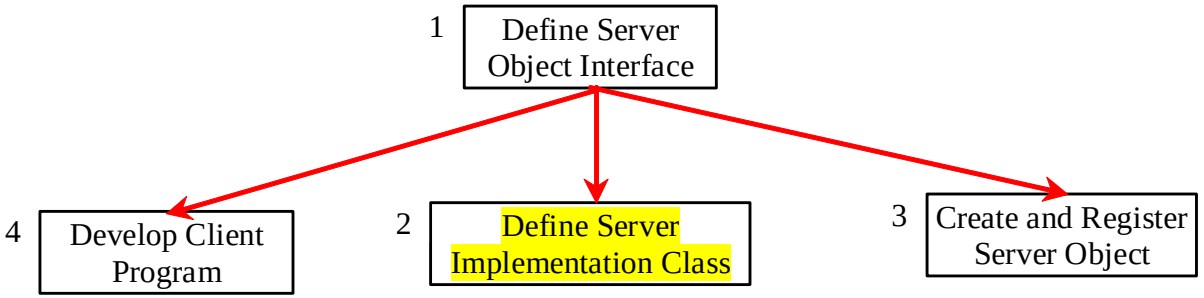


Server Implementation Class

- Define a class that implements the server object interface,

```
public class ServerInterfaceImpl extends UnicastRemoteObject
                                         implements ServerInterface{
        Public void service1(…) throws RemoteException{
        //Implement it
        }
        //implement other methods
}
```

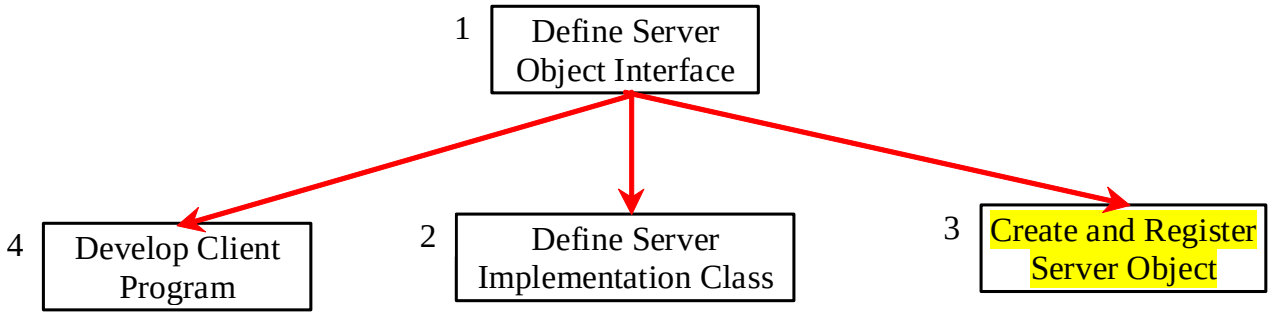
- Must extend the **java.rmi.server.UnicasstRemoteobject** class.
- **UnicasstRemoteobject** class provides support for point-to-point active object references using TCP streams.



Create and register server Object

- Create server object from the server implementation class and register it with an RMI registry:

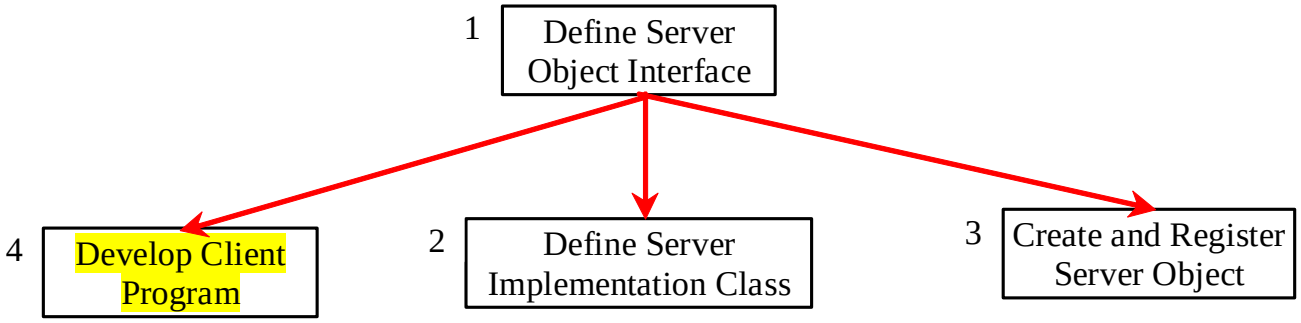
```
ServerInterface server = new ServerInterfaceImpl(…);
Registry registry = LocateRegistry.getRegistry();
Registry.rebind("RemoteObjectName", server);
```



Develop client program

- Develop a client that locates a remote object and invokes its methods like,

```
Registry registry = LocateRegistry.getRegistry(host);
ServerInterface server =
(ServerInterfaceImpl)registry.lookup("RemoteObjectName");
Server.service1(…);
```



Calculator.java (Interface)

```
pubilc interface Calculator extends java.rmi.Remote{
      public long add(long a, long b) throws
                              java.rmi.RemoteException;
      public long sub(long a, long b) throws
                              java.rmi.RemoteException;
      public long mul(long a, long b) throws
                              java.rmi.RemoteException;
      public long div(long a, long b) throws
                              java.rmi.RemoteException;
}
```

CalculatorImpl.java (Implementation Class)

```
public class CalculatorImpl extends
java.rmi.server.UnicastRemoteObject implements Calculator{
        //Implementations must have an explicit constructor
        //in order to declare the RemoteException exception

        public CalculatorImpl() throws
                                java.rmi.RemoteException{
        super();
        }
        public long add(long a, long b) throws
                                    java.rmi.RemoteException{
        return a + b;
        }
```

CalculatorImpl.java (Implementation Class)

```
    public long sub(long a, long b) throws
                                java.rmi.RemoteException{
    return a - b;
    }
    public long mul(long a, long b) throws
                                java.rmi.RemoteException{
    return a * b;
    }
    public long div(long a, long b) throws
                                java.rmi.RemoteException{
    return a / b;
    }
}
```

CalculatorServer.java

```
import java.rmi.Naming;

public class CalculatorServer{
	public CalculatorServer(){
		try{
			Calculator cal = new CalculatorImpl();
		Naming.rebind("rmi://localhost:1099/
					Calculatorservice", cal);
		}catch Exception(e){
			System.out.println("Trouble"+ e);
		}
	}
	public static void main(String[] args){
		new CalculatorServer();
	}
}
```

CalculatorClient.java

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class CalculatorClient {
	public static void main(String[] args) {
		try {
			Calculator c = (Calculator)
			Naming.lookup(
		"rmi://localhost/CalculatorService");
		System.out.println( c.sub(4, 3) );
		System.out.println( c.add(4, 5) );
		System.out.println( c.mul(3, 6) );
		System.out.println( c.div(9, 3) );
		}
```

```
        catch (MalformedURLException murle) {
        System.out.println();
        System.out.println("MalformedURLException");
        System.out.println(murle);
        }
        catch (RemoteException re) {
                System.out.println();
                System.out.println(
                                "RemoteException");
                System.out.println(re);
        }
        catch (NotBoundException nbe) {
                System.out.println();
                System.out.println( "NotBoundException");
                System.out.println(nbe);
        }
        catch (java.lang.ArithmeticException ae) {
                System.out.println();
                System.out.println( "java.lang.ArithmeticException");
                System.out.println(ae);
        }
        }
}
```

Running the application

- Step 1
  - Compile all the .java files
- Step 2 (Console – 1)
  - Run the *Registry*. You must be in the directory that contains all the classes in your command prompt then write command,
    - » rmiregistry
- Step 3 (Console – 2)
  - Run the server
    - » java CalculatorServer
- Step 4 (Console – 3)
  - Run the client
    - » java CalculatorClient

Simple Chat Program
ChatInterface.java

```
import java.rmi.*;

public interface ChatInterface extends Remote{
        public String getName() throws RemoteException;
        public void send(String msg) throws
                                                RemoteException;
        public void setClient(ChatInterface c)throws
                                                RemoteException;
        public ChatInterface getClient() throws
                                                RemoteException;
}
```

Chat.java (implementation)

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

public class Chat extends UnicastRemoteObject
implements ChatInterface  {

        public String name;
        public ChatInterface client=null;

        public Chat(String n)  throws
RemoteException {
                this.name=n;
        }
        public String getName() throws
RemoteException {
                return this.name;
        }

public void setClient(ChatInterface c){
                client=c;
        }

        public ChatInterface getClient(){
                return client;
        }

        public void send(String s) throws
RemoteException{
                System.out.println(s);
        }
}
```

ChatServer.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ChatServer {
public static void main (String[] argv) {
   try {

                Scanner s=new Scanner(System.in);
                Chat server = new Chat("Ali");

                Registry registry =
                LocateRegistry.createRegistry(2020);

                registry.rebind("Chat", server);

                System.out.println("[System] Chat Remote
                                Object is ready:");

while(true){
                String msg=s.nextLine().trim();
                if (server.getClient()!=null){

                ChatInterface client=server.getClient();

                msg="["+server.getName()+"] "+msg;

                client.send(msg);
                                }
                }

                }catch (Exception e) {
                System.out.println("[System] Server failed:
" + e);

                }
                }

}
```

ChatClient.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ChatClient {
                public static void main (String[] argv) {
                   try {
                ChatInterface client = new Chat("Ali");
                Scanner s=new Scanner(System.in);
                Registry registry =
                LocateRegistry.getRegistry(2020);
                ChatInterface server =
                (ChatInterface)registry.lookup("Chat");
                String msg="["+client.getName()+"] got
                                                connected";

                server.send(msg);
System.out.println("[System] Chat Remote Object is
ready:");
server.setClient(client);

while(true){

msg=s.nextLine().trim();

msg="["+client.getName()+"] "+msg;


server.send(msg);
}
}catch (Exception e) {
  System.out.println("[System] Server failed:
" + e);

                }
                }

}
```

RMI vs Socket Programming

- Enables you to program at higher level of abstraction.
- Hides the details of socket server, socket, connection and sending receiving data.
- Even implements the multithreading under the hood.

- Programming at a low-low-level.
- Program the details of socket server, socket, connection and also sending and receiving data.
- Explicitly implements threads for handling multiple clients

- Scalable and easy to maintain.
- Change the server or even move it another server without modifying the client program.
- Directly invoke the server method.
- RMI is similar to programming like in high-level programming.

- A client operation to send data requires a server operation to read it.
- Implementation of client and server is tightly synchronized.
- Limited to passing values.
- Socket programming is similar to programming in assembly language.