WEB LABORATORY MANUAL

1. Using InetAddress and Socket Programming in JAVA

The **package java**.**net** contains classes and interfaces that provide a powerful infrastructure for networking in **Java**.

Some of the classes in java.net package are

- **InetAddress class** - represent internet addresses (IP addresses)
- **ServerSocket class** - used on the server-side to wait for client connection requests.
- **Socket class** - for connecting to particular ports on specific Internet hosts and reading and writing data using streams.
- **DatagramSocket class** - used for sending and receiving datagrams
- **DatagramPacket class** - datagram packet – in addition to data also contains IP address and port information
- **MulticastSocket class** - can be used for sending and receiving packets to/from multiple users

Socket function calls

1. socket (): Create a socket
2. bind(): bind a socket to a local IP address and port #
3. listen(): passively waiting for connections
4. connect(): initiating a connection to another socket
5. accept(): accept a new connection
6. Write(): write data to a socket
7. Read(): read data from a socket
8. sendto(): send a datagram to another UDP socket
9. recvfrom(): read a datagram from a UDP socket
10. close(): close a socket (tear down the connection)

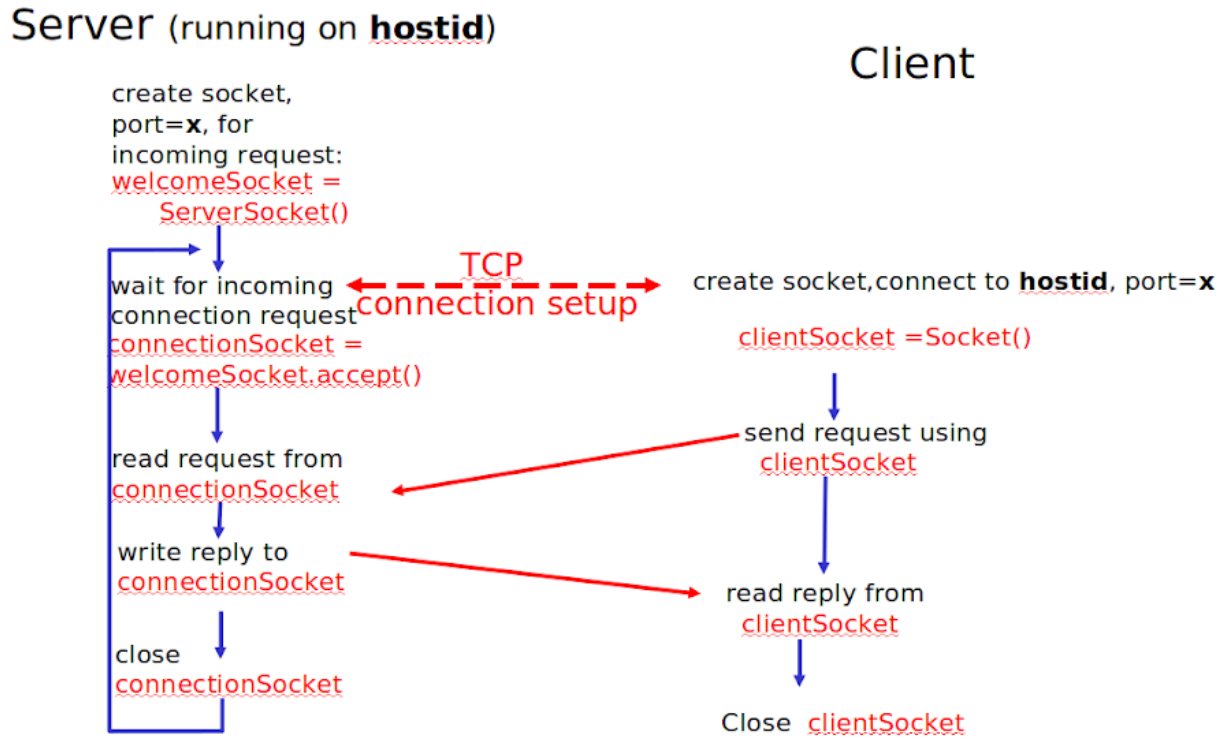
Steps involved in TCP:

Server-side:

1. Create a server socket and wait for a connection request from the client
2. Once the request from the client is available, accept the request.
3. Get the input from the client using InputStream object
4. Perform the required operation
5. Send the output using OutputStream object

Client-side:

1. Create a socket and establish a connection.
2. Get input from the user
3. Send the data to the server
4. Get output from the server
5. Output to the user
6. Close the socket.



TCPServer

```
import java.io.*;
import java.net.*;
class TCPServer {
	public static void main(String argv[]) throws Exception{
		String clientSentence;
		String capitalizedSentence;
		ServerSocket welcomeSocket = new ServerSocket(6789);
		while(true){
			Socket connectionSocket = welcomeSocket.accept();
			BufferedReader inFromClient = new BufferedReader(new
InputStreamReader(connectionSocket.getInputStream()));
			DataOutputStream outToClient = new
DataOutputStream(connectionSocket.getOutputStream());
	clientSentence = inFromClient.readLine();
	capitalizedSentence = clientSentence.toUpperCase() + '\n';
	outToClient.writeBytes(capitalizedSentence);
}
}
}
```

TCPClient

```
import java.io.*;
import java.net.*;
class TCPClient {
	public static void main(String argv[]) throws Exception
	{
		String sentence;
		String modifiedSentence;
		BufferedReader inFromUser = new BufferedReader(new
					InputStreamReader(System.in));
		Socket clientSocket = new Socket("127.0.0.1", 6789);
		DataOutputStream outToServer = new
			DataOutputStream(clientSocket.getOutputStream());
		BufferedReader inFromServer = new BufferedReader(new
			InputStreamReader(clientSocket.getInputStream()));
		sentence = inFromUser.readLine();
```

```
                    outToServer.writeBytes(sentence + '\n');
                    modifiedSentence = inFromServer.readLine();
                    System.out.println("FROM SERVER: " + modifiedSentence);
                    clientSocket.close();
    }
    }
```

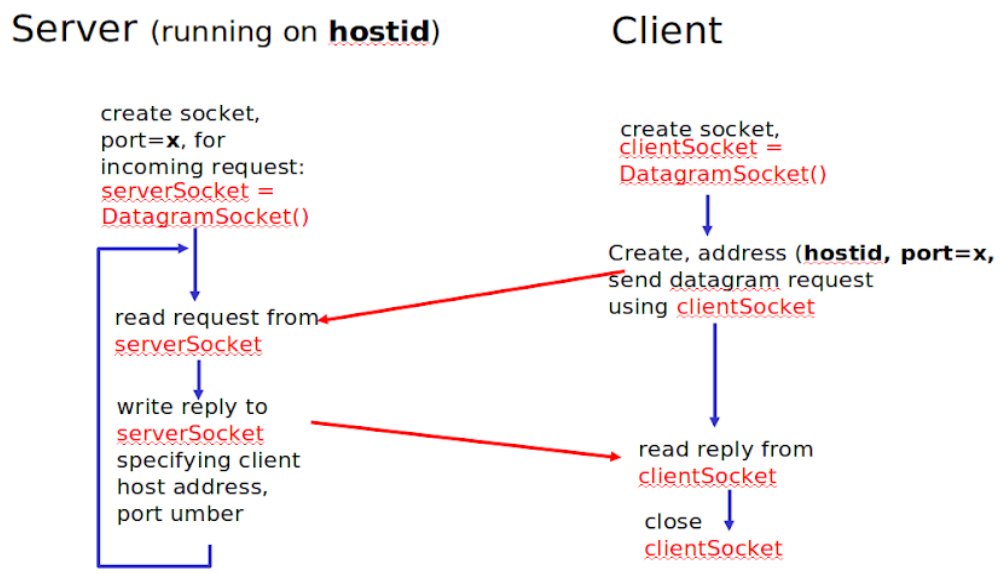
UDP

Server:

1. Create a Datagram socket and wait for datagram request request
2. Read request
3. Send a reply to the client by mentioning the host address and port number. (This is because there is no socket created for each client)

Client:

1. Create datagram socket
2. Send Datagram request
3. Receive reply from Server
4. Close Client Socket



UDPClient

```
import java.io.*;
import java.net.*;
class UDPClient {
        public static void main(String args[]) throws Exception
        {
                BufferedReader inFromUser = new BufferedReader(new
InputStreamReader(System.in));
                DatagramSocket clientSocket = new DatagramSocket();
                InetAddress IPAddress = InetAddress.getByName("hostname");
                byte[] sendData = new byte[1024];
                byte[] receiveData = new byte[1024];
                String sentence = inFromUser.readLine();
                sendData = sentence.getBytes();
                DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length, IPAddress, 9876);
                clientSocket.send(sendPacket);
                DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);
                clientSocket.receive(receivePacket);
                String modifiedSentence = new String(receivePacket.getData());
                System.out.println("FROM SERVER:" + modifiedSentence);
                clientSocket.close();
                }
}
```

UDPServer:

```
import java.io.*;
import java.net.*;
class UDPServer {
public static void main(String args[]) throws Exception
{
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData  = new byte[1024];
        while(true){
```

```
        DatagramPacket receivePacket = new DatagramPacket(receiveData,
        receiveData.length);
        serverSocket.receive(receivePacket);
        String sentence = new String(receivePacket.getData());
        InetAddress IPAddress = receivePacket.getAddress();
        int port = receivePacket.getPort();
        String capitalizedSentence = sentence.toUpperCase();
        sendData = capitalizedSentence.getBytes();
        DatagramPacket sendPacket =new DatagramPacket(sendData,
        sendData.length, IPAddress, port);
        serverSocket.send(sendPacket);
}
}
}
```

2.JAVA RMI

RMI allows a program to hold a reference to an object on a remote system and to call that object's methods.

Client-Server Architecture

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

Working of RMI

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called invoke() of the object remoteRef. It passes the request to the RRL on the server-side.

- The RRL on the server-side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

Components of RMI application

- **Remote interface**
- **Implementation class**
- **Server**
- **Client**

1) Remote interface

- Provides a description of all methods of a particular remote object

Steps:

I. Create an **interface** that extends the **predefined interface Remote** which belongs to the package.
II. Declare all the business methods that can be invoked by the client in this interface.

//Code

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

public interface Reminterface extends Remote {
//declaring methods
  void printMsg() throws RemoteException;
  void add(int a) throws RemoteException;
 }
```

2) Implementation Class

Steps:

I. **Implement** the interface created in the previous step.
II. Provide **implementation to all the abstract methods** of the remote interface.

//code

```
public class Impl implements Reminterface
{
    public static int total = 10;
// Implementation to abstract methods
  public void printMsg() {
    System.out.println("This is an example RMI program");
  }
  public void add(int a)
  {
          total=total+a;
  }
}
```

3) Server

Steps:

I. Create a class that **extends the implementation class** implemented in the previous step. (or implement the remote interface)
II. **Create a remote object** by instantiating the implementation class as shown below.
III. **Export the remote object** using the method exportObject() of the class named UnicastRemoteObject which belongs to the package java.rmi.server.
IV. **Get the RMI registry using the getRegistry()** method of the LocateRegistry class which belongs to the package java.rmi.registry.
V. **Bind the remote object** created to the registry using the bind() method of the class named Registry. To this method, pass a string representing the bind name and the object exported, as parameters.

//code

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class Serv extends Impl{
  public Serv() {}
  public static void main(String args[]) {
```

```
      try {
    // Instantiating the implementation class
        Impl obj = new Impl();

        // Exporting the object of implementation class
        Reminterface stub = (Reminterface) UnicastRemoteObject.exportObject(obj, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.bind("Reminterface", stub);
        System.err.println("Server ready");
       } catch (Exception e) {
        System.err.println("Server exception: " + e.toString());
        e.printStackTrace();
       }
   }
}
```

Client

I. Create a **client class** from where you want invoke the remote object.
II. **Get the RMI registry** using the getRegistry() method of the LocateRegistry class which belongs to the package java.rmi.registry.
III. **Fetch the object** from the registry using the method **lookup()** of the class Registry which belongs to the package java.rmi.registry. To this method you need to pass a string value representing the bind name as a parameter. This will return you the remote object downcast it.
IV. The lookup() returns an object of type remote,
V. invoke the required method using the obtained remote object.

```
public class Cli {
   private Cli() {}
  public static int a,op;
   public static void main(String[] args) {
      try {

        Registry registry = LocateRegistry.getRegistry(null);
         Reminterface stub = (Reminterface) registry.lookup(.l"Reminterface");
```

```
        // Calling the remote method using the obtained object
        stub.printMsg();

        while( true ){
                Scanner s= new Scanner(System.in);
                op=s.nextInt();

                switch(op){
                        case 1:
                                System.out.println("add invoked");
                                System.out.println("ENTER NUMBER TO BE ADDED");
                                a=s.nextInt();
                                stub.add(a);
                                break;
                        default:
                                break;

                        }
                    }
            }
            catch (Exception e) {
                    System.err.println("Client exception: " + e.toString());
                    e.printStackTrace();
        }

    }
}
```