

Distributed Systems and Cloud Computing

Assignment 1

Program 1

1. Aim: Write a program for one-way client and server communication using Java Socket where the client sends a message to the server, then the server reads the message and prints it. Write a program and execute it.

2. Theory:

Socket programming is a crucial part of network communication, enabling applications to send and receive data over a network. By creating sockets, applications can interact with each other, sharing information and resources.

TCP vs. UDP

- TCP (Transmission Control Protocol): This protocol ensures reliable and ordered delivery of data. It's perfect for applications where data integrity is essential, such as web browsing (HTTP). TCP makes sure that all data packets arrive in the correct order and without errors.
- UDP (User Datagram Protocol): UDP is faster but doesn't guarantee delivery or order of data packets. It's ideal for real-time applications like video streaming or online gaming, where speed is more critical than perfect accuracy.

Socket Class in Java In Java, the Socket class handles client-side network connections. This class provides methods to:

- Connect to a server.
- Send and receive data.
- Close the connection once communication is complete.

ServerSocket Class in Java On the server side, the ServerSocket class is used to listen for and accept incoming client connections. This class includes methods to:

- Bind the server to a specific port.
- Accept client connections.
- Close the server socket when it's no longer needed.

By using these classes, Java applications can establish network communications, enabling a wide range of functionalities, from simple chat applications to complex client-server architectures.

3. Program

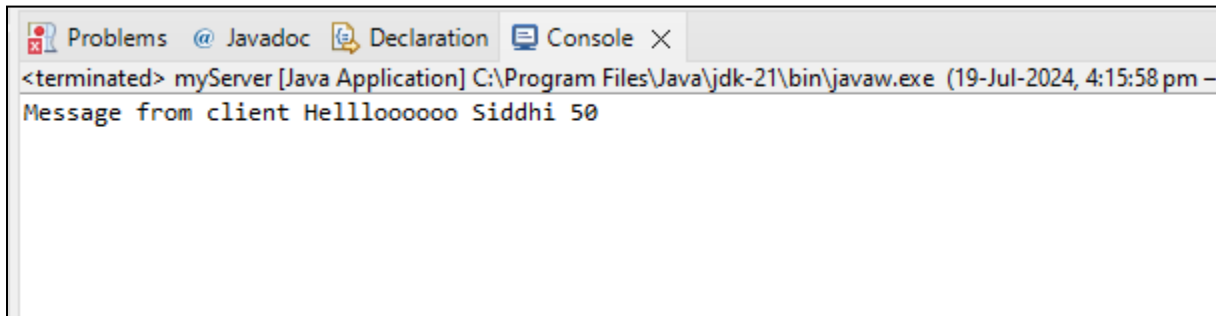
myServer.java

```
package newpractical;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
public class myServer {
    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub
        ServerSocket ss=new ServerSocket(4000);
        Socket s1 =ss.accept();
        DataInputStream dis=new DataInputStream(s1.getInputStream());
        String str=dis.readUTF( );
        System.out.println("Message from client"+str);
        //s1.close();
    }
}
```

myClient.java

```
package newpractical;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;
public class myClient {
    public static void main(String[] args) throws IOException, Exception {
        // TODO Auto-generated method stub
        Socket s= new Socket("localhost",4000);
        DataOutputStream dos=new DataOutputStream(s.getOutputStream());
        DataInputStream dis=new DataInputStream(s.getInputStream());
        dos.writeUTF(" Helllloooooo Siddhi 50");
        s.close();
    }
}
```

4. Output



The screenshot shows a Java IDE window with a tab labeled 'Console'. The console output displays the termination of a Java application named 'myServer'. The output text is: '<terminated> myServer [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (19-Jul-2024, 4:15:58 pm - Message from client Hellloooooo Siddhi 50'.

5. Conclusion

In conclusion, socket programming forms the backbone of network communication, allowing applications to interact over the internet. TCP and UDP cater to different needs, with TCP offering reliable data transfer and UDP prioritizing speed. Java's `Socket` and `ServerSocket` classes facilitate client-server interactions by providing essential methods for connecting, data exchange, and managing connections. Mastering these concepts is crucial for developing robust and efficient networked applications.

Program 2:

1. Aim: Write a program for client server chat(Two-way communication) using java socket.
Write a program and execute it.

2. Theory:

Socket Programming

Socket programming involves using sockets to enable network communication between applications, allowing them to send and receive data over a network.

Communication Protocols: TCP and UDP

- TCP (Transmission Control Protocol): Provides reliable, ordered, and error-checked data delivery by establishing a connection before transmission. Ideal for applications needing data integrity.
- UDP (User Datagram Protocol): Offers faster, connectionless communication without guarantees for delivery or order, suitable for real-time applications where speed is more critical.

Socket Class

The Socket class in Java allows for client-side network connections. It provides methods to connect to servers, send and receive data, and close the connection.

ServerSocket Class

The ServerSocket class in Java enables server-side network communication by listening for client connections, accepting them, and managing the communication.

3. Program

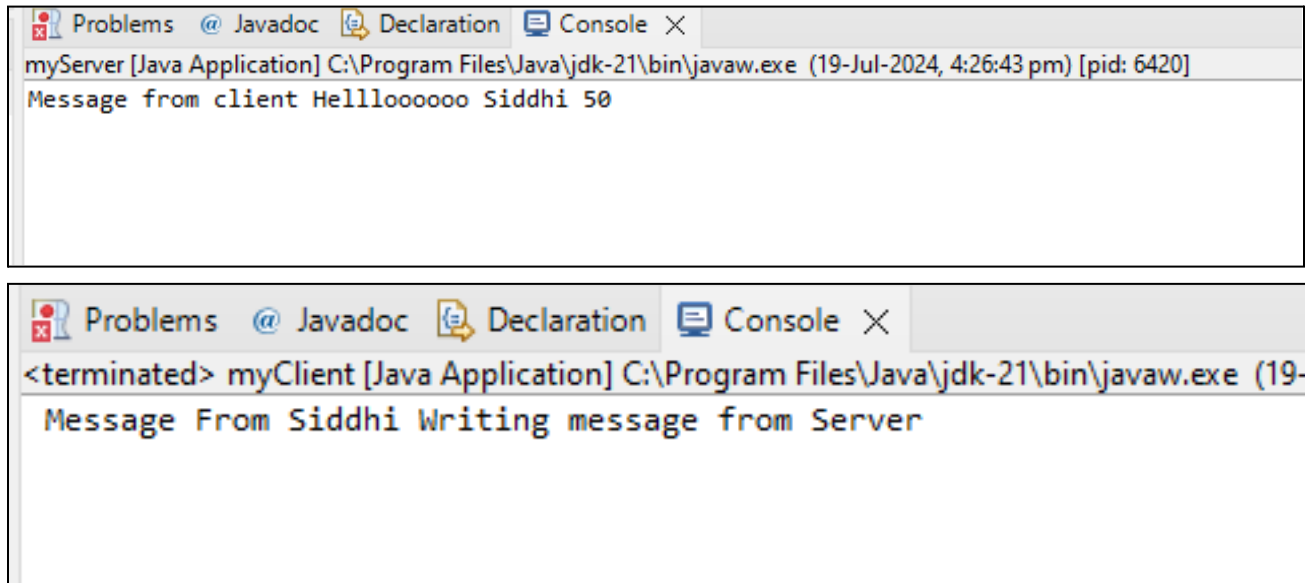
myServer.java

```
package newpractical;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
public class myServer {
    public static void main(String[] args) throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        ServerSocket ss=new ServerSocket(4000);
        Socket s1 =ss.accept();
        DataInputStream dis=new DataInputStream(s1.getInputStream());
        DataOutputStream dos=new DataOutputStream(s1.getOutputStream());
        String str=dis.readUTF( );
        System.out.println("Message from client"+str);
        Thread.sleep(3000);
        dos.writeUTF("Writing message from Server");
        //s1.close();
    }
}
```

myClient.java

```
package newpractical;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;
public class myClient {
    public static void main(String[] args) throws IOException, Exception {
        // TODO Auto-generated method stub
        Socket s= new Socket("localhost",4000);
        DataOutputStream dos=new DataOutputStream(s.getOutputStream());
        DataInputStream dis=new DataInputStream(s.getInputStream());
        dos.writeUTF(" Helllloooooo Siddhi 50");
        String str1 =dis.readUTF( );
        System.out.println(" Message From Siddhi "+str1);
        s.close();
    }
}
```

4. Output



5. Conclusion

Implementing a client-server chat application using Java sockets demonstrates fundamental network programming concepts, including socket creation, data transmission, and bi-directional communication. The provided example illustrates how to establish a connection between a client and a server, enabling them to exchange messages in real-time. This hands-on approach reinforces understanding of TCP/IP protocols and socket classes in Java, showcasing their practical applications in building interactive networked applications.

Program 3

1. Aim: A program for client server GUI chat using java Socket.

Write a program and execute it. Submit program documentation containing following points:

2. Theory:

Creating a GUI-based client-server chat application in Java involves using Java Swing for the graphical interface and Java Sockets for network communication. The key components include:

1. **Java Sockets:** Provide the fundamental mechanism for network communication. The server listens for incoming connections, while the client connects to the server.
2. **Swing GUI:** Provides graphical user interface elements, such as text fields, buttons, and message areas, to interact with users. JFrame is used for the main window, JTextArea for displaying messages, and JTextField for user input.
3. **Threads:** Both the client and server use separate threads to handle incoming and outgoing messages concurrently, ensuring that the GUI remains responsive.

3. Program

Server.java

```
package prac1b;
import java.awt.FlowLayout;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

public class Server extends JFrame implements ActionListener, Runnable {
    JButton b;
    JTextField tf;
    JTextArea ta;
    ServerSocket ss;
    Socket s;
    PrintWriter pw;
    BufferedReader br;
    Thread th;

    public Server() {
        b = new JButton("Send");
        b.addActionListener((ActionListener) this);
        tf = new JTextField(20);
        ta = new JTextArea(20, 30);
        add(ta);
        add(tf);
        add(b);
        try {
            ss = new ServerSocket(4000);
            s = ss.accept();
            br = new BufferedReader(new InputStreamReader(s.getInputStream()));
            pw = new PrintWriter(s.getOutputStream(), true);
```

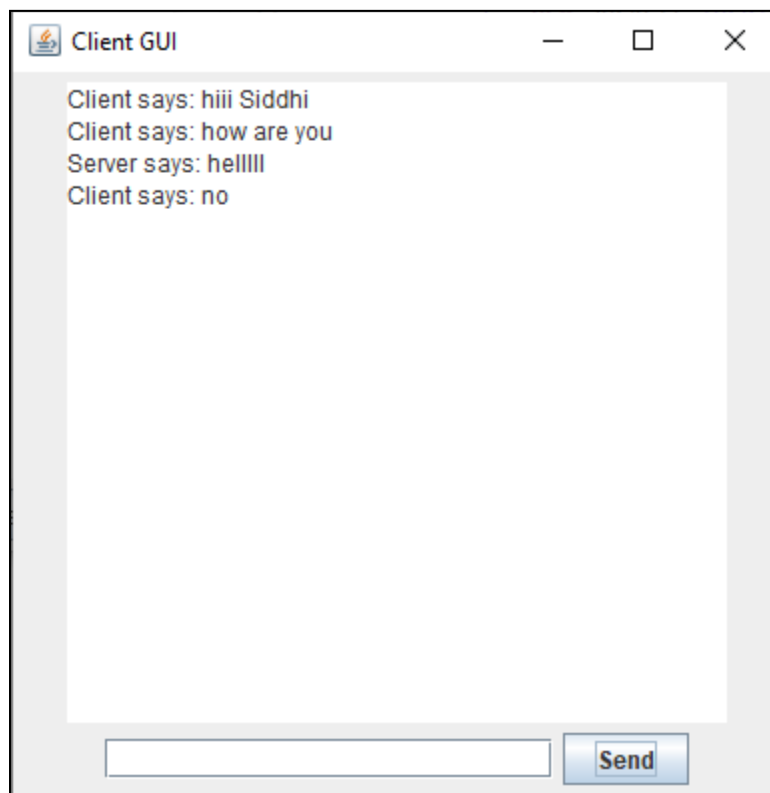
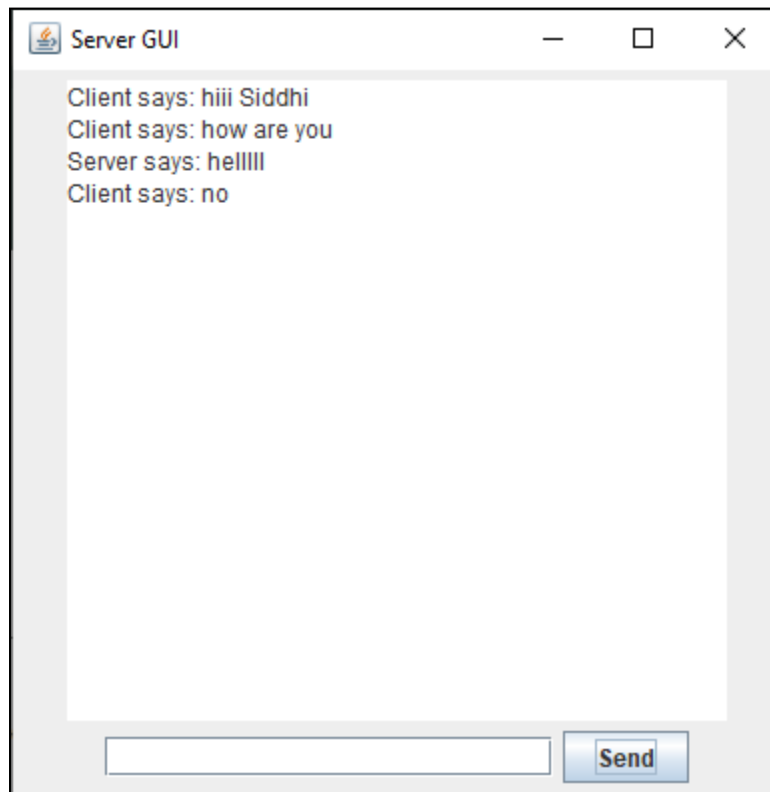
```
    }
    catch (Exception e) {
        System.out.println(e);
    }
    th = new Thread(this);
    th.start();
}
public void run() {
    while(true) {
        try {
            ta.append("Client says: " + br.readLine() + "\n");
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
public void actionPerformed(ActionEvent ae) {
    pw.println(tf.getText());
    ta.append("Server says: " + tf.getText() + "\n");
    tf.setText("");
}
public static void main(String[] args) {
    Server server = new Server();
    server.setLayout(new FlowLayout());
    server.setSize(400, 400);
    server.setTitle("Server GUI");
    server.setVisible(true);
}
}
```

Client.java

```
package prac1b;
import java.awt.FlowLayout;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;
public class Client extends JFrame implements ActionListener, Runnable {
    JButton b;
    JTextField tf;
    JTextArea ta;
    Socket s;
    PrintWriter pw;
```

```
        BufferedReader br;
        Thread th;
        public Client() {
            b = new JButton("Send");
            b.addActionListener((ActionListener) this);
            tf = new JTextField(20);
            ta = new JTextArea(20, 30);
            add(ta);
            add(tf);
            add(b);
            try {
                s = new Socket("localhost", 4000);
                br = new BufferedReader(new InputStreamReader(s.getInputStream()));
                pw = new PrintWriter(s.getOutputStream(), true);
            }
            catch (Exception e) {
                System.out.println(e);
            }
            th = new Thread(this);
            th.start();
        }
        public void run() {
            while(true) {
                try {
                    ta.append("Server says: " + br.readLine() + "\n");
                }
                catch(Exception e) {
                    System.out.println(e);
                }
            }
        }
        public void actionPerformed(ActionEvent ae) {
            pw.println(tf.getText());
            ta.append("Client says: " + tf.getText() + "\n");
            tf.setText("");
        }
        public static void main(String[] args) {
            Client client = new Client();
            client.setLayout(new FlowLayout());
            client.setSize(400, 400);
            client.setTitle("Client GUI");
            client.setVisible(true);
        }
    }
```


4. Output



5. Conclusion

Creating a GUI-based client-server chat application using Java sockets and Swing provides a practical example of network programming combined with user interface design. This approach highlights the integration of network communication with interactive graphical elements, demonstrating the ability to manage real-time text exchange through a visually appealing interface. By using threads for concurrent communication and Swing for the GUI, developers can build responsive and user-friendly chat applications.

Program No. 4

1. **Aim:** Implement a Program for multi-client chat server.

Write a program and execute it. Submit program documentation containing following points:

2. Theory:

A multi-client chat server enables multiple clients to connect and communicate simultaneously. This setup requires handling several client connections concurrently, typically through threading or asynchronous I/O. The core concepts include:

1. **Socket Communication:** Each client and server use sockets to establish connections and exchange messages.
2. **Concurrency:** The server must manage multiple client connections at once. This is usually achieved through threading, where each client connection is handled by a separate thread, or using non-blocking I/O for scalability.
3. **Broadcasting Messages:** The server needs to broadcast messages from one client to all other connected clients to ensure real-time communication.

3. Program

MulClient.java

```
package practical;
```

```
import java.awt.Frame;  
import java.awt.TextArea;  
import java.awt.*;  
import java.io.DataInputStream;  
import java.io.DataOutputStream;  
import java.io.IOException;  
import java.net.Socket;  
import java.util.logging.Level;  
import java.util.logging.Logger;
```

```
public class MulClient extends Frame implements Runnable
```

```
{  
    TextArea ta;  
    TextField tf;  
    Button btnSend, btnClose;  
  
    Socket s;  
    DataInputStream dis;  
    DataOutputStream dos;  
  
    String LoginName;  
    String sendTo;
```

```
Thread th=null;

public MulClient(String LoginName,String chatswith) throws IOException{
    this.LoginName=LoginName;
    this.sendTo=chatswith;

    ta=new TextArea(50,50);
    tf=new TextField(50);
    btnSend=new Button("SEND");
    btnClose=new Button("CLOSE");

    s=new Socket("localhost", 1223);
    dis=new DataInputStream(s.getInputStream());
    dos=new DataOutputStream(s.getOutputStream());
    dos.writeUTF(LoginName);
    th=new Thread(this);
    th.start();
}

void setup(){
    setSize(600, 400);
    setLayout(new GridLayout(2, 1));
    add(ta);

    Panel p=new Panel();
    p.add(tf);
    p.add(btnSend);
    p.add(btnClose);
    add(p);
    setTitle(LoginName);
    setVisible(true);
}

public boolean action(Event e,Object o){
    if(e.arg.equals("SEND")){

        try {
            //another client DATA MSg
            dos.writeUTF(sendTo+" DATA "+tf.getText().toString());
            ta.append("\n"+LoginName+" says:"+tf.getText().toString());
            tf.setText("");
        } catch (IOException ex) {
            Logger.getLogger(MulClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

```
    }

    else if(e.arg.equals("CLOSE")){
        try {
            //LoginName LOGOUT
            dos.writeUTF(LoginName+" LOGOUT");
            System.exit(1);

        } catch (IOException ex) {
            Logger.getLogger(MulClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    return super.action(e, o);
}

@Override
public void run() {
    while(true){
        try {

            ta.append("\n"+sendTo+" says:"+dis.readUTF());

        } catch (IOException ex) {
            Logger.getLogger(MulClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

public static void main(String args[]) throws IOException{
    MulClient mc=new MulClient("Ria", "rita");
    mc.setup();
}
}
```

MulServer.java

package practical;

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.*;
import java.util.StringTokenizer;
import java.util.Vector;
public class MulServer
{
    static Vector ClientSockets;
    static Vector LoginNames;

    public MulServer() throws IOException, Exception{
        ServerSocket ss=new ServerSocket(1223);
        ClientSockets=new Vector();
        LoginNames=new Vector();

        while(true){
            Socket CSoc=ss.accept();
            AcceptClient objClient=new AcceptClient(CSoc);

        }
    }
    class AcceptClient extends Thread{

        Socket ClientSocket;
        DataInputStream dis;
        DataOutputStream dos;
        public AcceptClient(Socket CSoc) throws Exception
        {
            ClientSocket=CSoc;
            dis=new DataInputStream(ClientSocket.getInputStream());
            dos=new DataOutputStream(ClientSocket.getOutputStream());
            String LoginName=dis.readUTF();

            System.out.println("User logged In:"+LoginName);

            LoginNames.add(LoginName);
            ClientSockets.add(ClientSocket);
            start(); }
        public void run(){
```

```
while(true){

    try{
        String msgFromClient=dis.readUTF();
        StringTokenizer st=new StringTokenizer(msgFromClient);

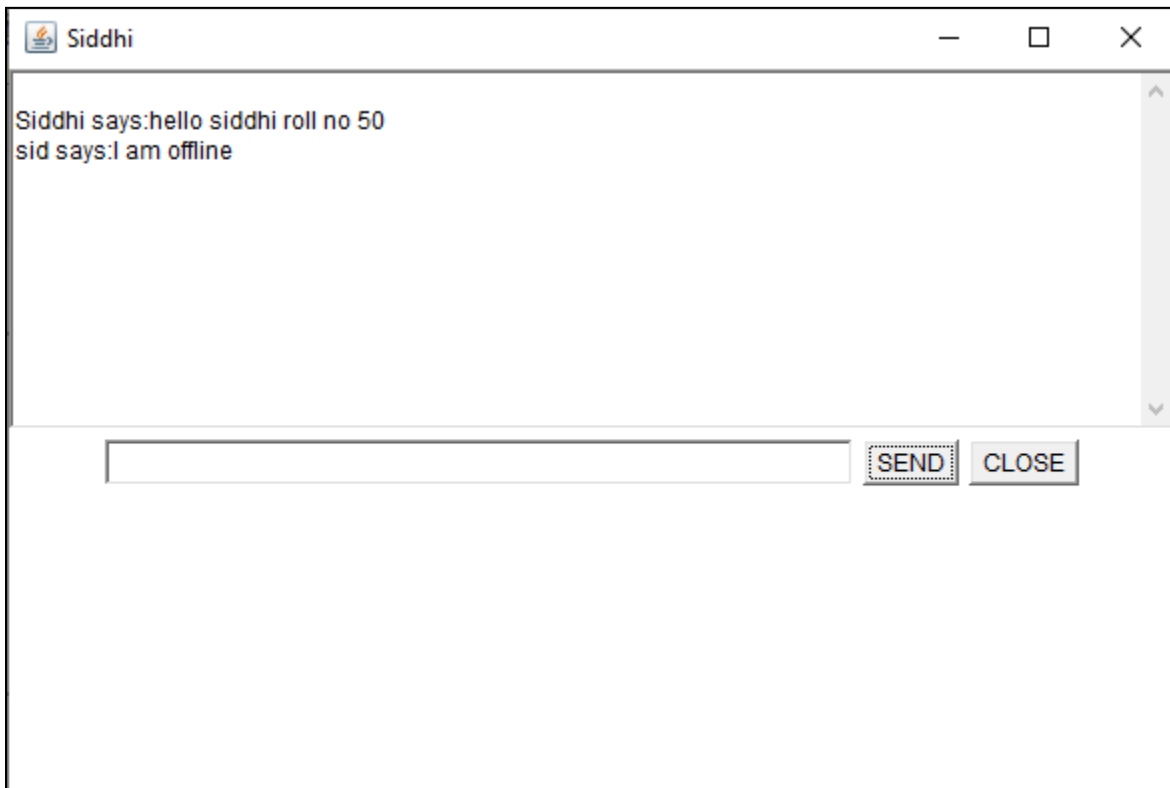
        //sendto msgType msg //loginName Logout
        String sendTo=st.nextToken();
        String MsgType=st.nextToken();
        int iCount=0;

        if(MsgType.equals("LOGOUT")){
            for(iCount=0;iCount<LoginNames.size();iCount++){
                {
                    if(LoginNames.elementAt(iCount).equals(sendTo)){
                        LoginNames.removeElementAt(iCount);
                        ClientSockets.removeElementAt(iCount);
                        System.out.println("User "+sendTo+" has Logged out...");
                        break;
                    }
                }
            }
        }
    }
    else{

        //Hello How are you?
        String msg="";
        while(st.hasMoreTokens()){
            msg+=st.nextToken();
        }
        for(iCount=0;iCount<LoginNames.size();iCount++){
            if(LoginNames.elementAt(iCount).equals(sendTo)){
                Socket dSoc=(Socket)ClientSockets.elementAt(iCount);
                DataOutputStream ddos=new DataOutputStream(dSoc.getOutputStream());
                ddos.writeUTF(msg);
                break;
            }
        }
        if(iCount==LoginNames.size()){
            dos.writeUTF("I am offline");
        }
    }
}
```

```
        }catch(Exception e){  
        }  
    }  
}  
  
public static void main(String[] args) throws Exception {  
    MulServer ms=new MulServer();  
}  
}
```

4. Output



5. Conclusion

Implementing a multi-client chat server in Java demonstrates how to manage multiple simultaneous connections and facilitate real-time communication among clients. This approach utilizes threads to handle each client connection separately and broadcasts messages to all connected clients, ensuring efficient message delivery. The client-side GUI provides an interactive interface for users, while the server manages connections and message distribution. This setup is fundamental for developing scalable chat applications and understanding concurrent network programming.