Software Requirements Specification

for

TCP-Based Client-to-Client Chat Application

Version 1.0 approved

Prepared by Ankita Nath

Heritage Institute Of Technology, Kolkata

20/03/2025

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
Ankita Nath	20/03/2025	N/A	Version 1.0

1. Introduction

1.1 Purpose

Product Name: *TCP-Based Client-to-Client Chat Application*
Version: *1.0*
Purpose:

- *Provide a real-time chat system for direct client-to-client communication using TCP.*
- *Manage client connections and message forwarding via a central server.*

Scope:

- *This document describes the entire system, including both the client and server components.*
- *It is intended for local network use.*

1.2 Document Conventions

- ***Fonts and Styles***
  - ***Section Titles:*** *Bold, size 18, Times New Roman font.*
  - ***Subsection Titles:*** *Bold, size 14, Times New Roman font.*
  - ***Body Text:*** *Italicized, size 11, Arial font.*
  - ***Key Terms or Concepts:*** *Highlighted in bold for emphasis within the body text.*
- ***Requirements Prioritization***
  1. Requirements are labeled using a numbering format (e.g., **REQ-1**, **REQ-2**).
  2. Each requirement is assigned a priority using the following tags:
     - ***High:*** *Critical for the product's functionality.*
     - ***Medium:*** *Important but not essential in the initial release.*
     - ***Low:*** *Desirable features that can be deferred to later versions.*
- ***Terminology and Abbreviations:***
  - *The terms **Client**, **Server**, **Socket**, and **Message** are capitalized when referring to specific components of the chat system.*
  - *Commands like GET, SEND, and EXIT are written in uppercase to indicate user input.*

1.3 Intended Audience and Reading Suggestions

- ***Developers****: Focus on the **Functional Requirements** and **System Features** sections for implementation details.*
- ***Project Managers****: Refer to the **Scope** and **Overall Description** for project planning and progress tracking.*
- ***Testers****: Review the **Test Cases** and **Acceptance Criteria** to design and execute test plans.*
- ***Users****: Read the **User Interface Description** for an overview of application usage.*
- ***Documentation Writers****: Use the **Glossary** and **Appendices** to ensure accurate technical documentation.*

Reading Sequence:

1. ***Introduction*** *→ For an overview of the application.*
2. ***Scope and Purpose*** *→ To understand the project's boundaries.*
3. ***Functional Requirements*** *→ For detailed functionality.*
4. ***System Architecture*** *→ For implementation insights.*
5. ***Test Cases*** *→ For verification and validation understanding.*

1.4 Product Scope

- *The software is a **Client-to-Client Chat Application** using **Java**.*
- *Facilitates **real-time communication** between clients via a **TCP Server**.*
- *Designed for **simple, reliable, and efficient** messaging.*
- *Supports **multiple client connections** with direct message forwarding.*

1.5 References

None.

2. Overall Description

2.1 Product Perspective

- *The **Client-to-Client Chat Application** is a **new, self-contained product**.*
- *It is developed using **Java** and executed in **NetBeans**.*
- *The system uses a **TCP Server** to facilitate communication between clients.*
- *The product provides a **simple interface** for real-time chat.*

2.2 Product Functions

- ***Client Communication**: Enable real-time chat between connected clients.*
- ***Message Forwarding**: Relay messages from one client to the intended recipient using a central server.*
- ***Client Management**: Assign unique IDs to clients and maintain a list of active clients.*
- ***Command Support**: Provide commands like GET to view active clients and SEND to transmit messages.*
- ***Session Management**: Handle client connections, disconnections, and reconnections.*

2.3 User Classes and Characteristics

- ***Regular Users:***
  - *Use the chat application for basic communication.*
  - *Limited to sending and receiving messages.*
  - *Can view the list of connected clients using commands.*
- ***Administrator:***
  - *Manages the server-side operations.*
  - *Monitors client connections and server status.*
  - *Can disconnect users if necessary.*

2.4 Operating Environment

***Operating System**: Windows 10 or higher.*
***Development Environment**: NetBeans IDE.*
***Java Version**: JDK 8 or higher.*

2.5 Design and Implementation Constraints

- ***Development Environment**: The application will be developed and executed using **NetBeans**.*
- ***Programming Language**: The software will be implemented in **Java**.*
- ***Communication Protocol**: Uses **TCP** for client-to-client communication via a server.*
- ***Operating System**: Compatible with any OS supporting **Java Runtime Environment (JRE).***

2.6 User Documentation

- ***User Manual:***
  - *Step-by-step instructions for installing and running the chat application.*
  - *Detailed explanation of commands (GET, SEND, EXIT).*
  - *Troubleshooting guide for common issues.*
- ***Tutorial:***

- *A short, interactive tutorial guiding users through connecting to the server, sending messages, and disconnecting.*

Delivery Formats: *PDF for user manuals and tutorials.*

2.7 Assumptions and Dependencies

- ***Java Environment****: Assumes* ***Java 8*** *or later is installed for development and execution.*
- ***NetBeans IDE****: The project is developed and tested using* ***NetBeans****.*
- ***TCP/IP Network****: A stable* ***TCP/IP network*** *connection is available for communication.*
- ***Operating System****: Compatible with* ***Windows, macOS, or Linux****.*

3. External Interface Requirements

3.1 User Interfaces

- ***Interface Type****: Console-based, text-only interface using* ***NetBeans*** *for running the program.*
- ***Screen Layout****: No graphical user interface (GUI); interactions occur through a command-line terminal.*
- ***Input/Output:***
  - *User Input: Text-based commands (e.g., SEND <client_id> <message>, GET).*
  - *Output: Displayed messages from other clients and server responses.*

3.2 Hardware Interfaces

- *The* ***Client-to-Client Chat Application*** *using* ***Java*** *is executed on* ***standard desktop or laptop computers****.*
- *Developed and run using* ***NetBeans IDE****.*
- *Requires a system with a* ***TCP/IP network interface*** *for communication.*
- *Supports any hardware with* ***Windows, Linux, or macOS*** *operating systems.*

3.3 Software Interfaces

- ***Operating System****: Compatible with* ***Windows*** *or* ***Linux****.*
- ***Development Environment****: Developed and executed using* ***NetBeans IDE****.*
- ***Network Communication****: Uses* ***TCP Sockets*** *for client-server communication.*
- ***Message Handling****: Messages are transmitted using* ***TCP/IP protocols****.*

3.4 Communications Interfaces

- ***Protocol****: The chat application uses* ***TCP/IP*** *for client-to-client communication via a server.*
- ***Socket Communication****: Clients establish a connection using* ***Java Sockets*** *on port* ***5000****.*
- ***Synchronization****: Managed using server-side message queuing and client socket handling.*

4. System Use Cases

4.1 Client Registration

4.1.1 Description and Priority

- ***Description:*** *Clients register with the server and receive a unique Client ID.*
- ***Priority:*** *High*

4.1.2 Stimulus/Response

S: *Client sends a connection request.*
R: *Server assigns a Client ID and sends it back.*

4.1.3 Functional Requirements:

REQ-1: *Assign Client ID upon connection.*
REQ-2: *Notify the client of their Client ID.*

4.2 Sending Messages

4.2.1 Description and Priority

- ***Description:*** *Clients can send messages to other clients using a specific format.*
- ***Priority:*** *High*

4.2.2 Stimulus/Response

S: *Client sends a SEND command.*
R: *Server validates and forwards the message.*

4.2.3 Functional Requirements

REQ-1: *Validate the Client ID.*
REQ-2: *Forward the message to the correct recipient.*

4.3 Client List Retrieval

4.3.1 Description and Priority

- ***Description:*** *Clients can request a list of all connected clients.*
- ***Priority:*** *Medium.*

4.3.2 Stimulus/Response

S: *Client sends a GET request.*
R: *Server responds with the list of Client IDs.*

4.3.3 Functional Requirements

REQ-1: *Provide the list of connected clients.*

4.4 Client Disconnection

4.4.1 Description and Priority

- ***Description:*** *Clients can disconnect from the chat server using the EXIT command.*
- ***Priority:*** *High.*

4.4.2 Stimulus/Response

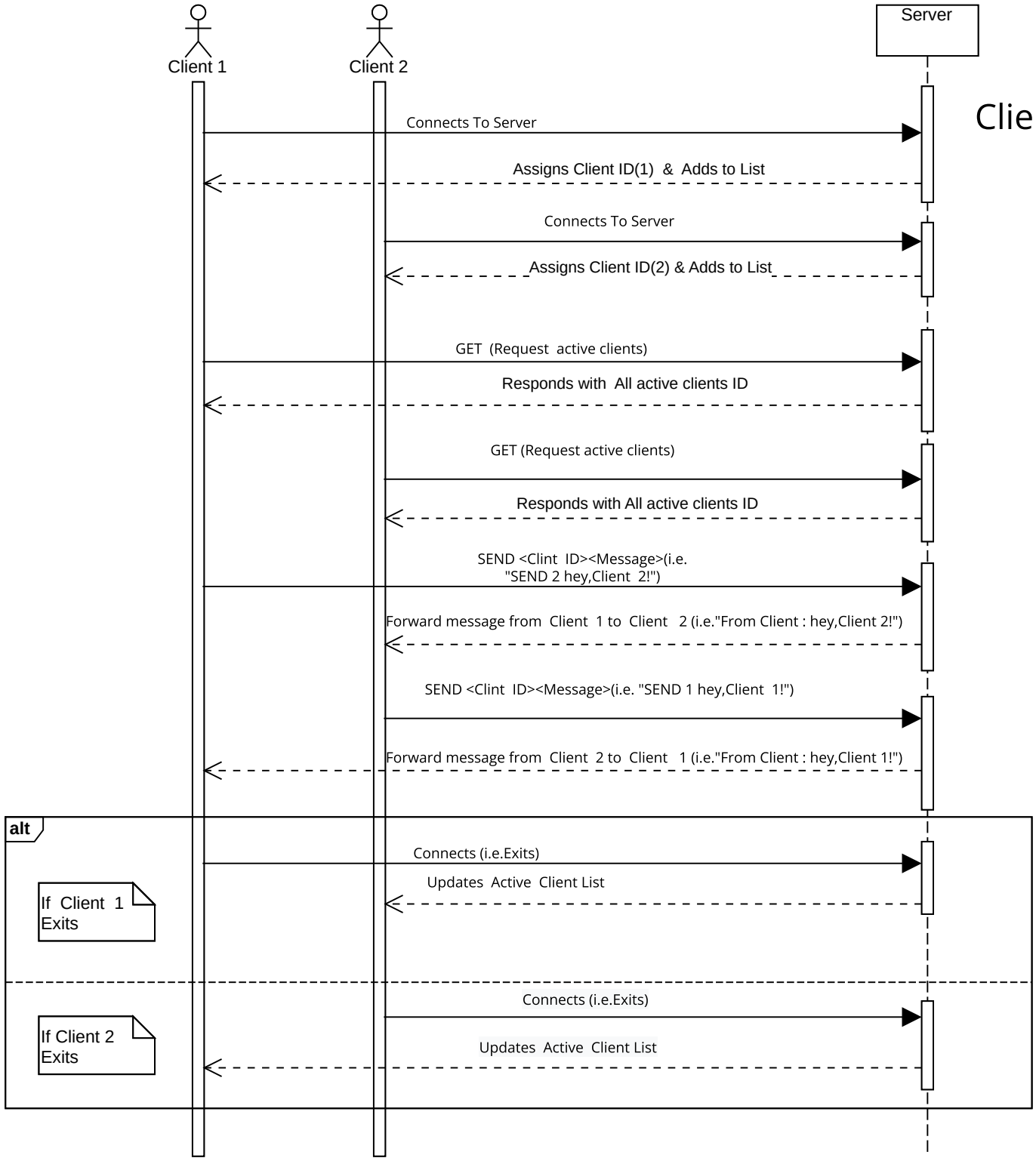
S: *Client sends an EXIT command.*
R: *Server acknowledges and disconnects the client.*

4.4.3 Functional Requirements

REQ-1: *Remove disconnected clients from the active list.*

Appendix A: Glossary

- ***REQ:*** *Requirements*
- ***S:*** *Stimulus*
- ***R:*** *Response*



Client -To- Client Chat Program (Multithreaded Way)

Software Requirements Specification

for

UDP-based Client-to-Client Chat Application

Version 1.0 approved

Prepared by Ankita Nath

Heritage Institute Of Technology, Kolkata

20/03/2025

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
Ankita Nath	20/03/2025	N/A	Version 1.0

1. Introduction

1.1 Purpose

*This document specifies the software requirements for the **UDP-based Client-to-Client Chat Application**, **version 1.0**. It **describes the entire chat system, including the server and client components**. The application facilitates **one-to-one messaging using UDP sockets**, serving as a standalone solution without integration into a larger system.*

1.2 Document Conventions

- ***Fonts and Styles***
  - ***Section Titles:** Bold, size 18, Times New Roman font.*
  - ***Subsection Titles:** Bold, size 14, Times New Roman font.*
  - ***Body Text:** Italicized, size 11, Arial font.*
  - ***Key Terms or Concepts:** Highlighted in bold for emphasis within the body text.*
- ***Requirements Prioritization***
  1. Requirements are labeled using a numbering format (e.g., **REQ-1**, **REQ-2**).
  2. Each requirement is assigned a priority using the following tags:
     - ***High:** Critical for the product's functionality.*
     - ***Medium:** Important but not essential in the initial release.*
     - ***Low:** Desirable features that can be deferred to later versions.*
- ***Terminology and Abbreviations:***
  - ***Server** refers to the central node that handles client messages.*
  - ***Client** refers to the user-side application sending and receiving messages.*
  - ***UDP** stands for **User Datagram Protocol** used for connectionless communication.*

1.3 Intended Audience and Reading Suggestions

*This document is intended for **developers**, **testers**, **project managers**, and **stakeholders** involved in the development of the UDP-based Client-to-Client Chat Application.*

- ***Developers** should focus on the **Functional Requirements** and **System Features** sections.*
- ***Testers** should review the **External Interface Requirements** and **Performance Requirements** for test case creation.*
- ***Project Managers** may refer to the **Overall Description** and **Assumptions and Dependencies** for project planning.*
- ***Stakeholders** can read the **Purpose** and **Scope** sections for a high-level overview.*

1.4 Product Scope

The chat application will:

- *Support one-to-one messaging using UDP sockets.*
- *Provide client registration and identification.*
- *Allow users to send messages using a simple command format.*
- *Enable clients to retrieve the list of connected users.*
- Provide a graceful exit mechanism for clients.

1.5 References

- *Java Documentation for UDP Socket Programming.*

2. Overall Description

2.1 Product Perspective

The application consists of two main components:

- **Client**: Provides an interface for users to send and receive messages.
- **Server**: Manages client registrations and relays messages between connected clients.

2.2 Product Functions

- *Client registration using UDP sockets.*
- *Display of connected clients using the GET command.*
- *Sending messages using the SEND <ClientID> <Message> command.*
- *Disconnecting using the EXIT command.*

2.3 User Classes and Characteristics

- ***Regular User (Client):***
  - *Uses the chat application to send and receive messages.*
  - *Can request a list of connected clients.*
  - *Can initiate and terminate chat sessions.*
- ***Administrator (Server Manager):***
  - *Manages server operations and monitors active clients.*
  - *Can start, stop, and configure the server.*

2.4 Operating Environment

Operating System Support:

- *Windows 10 and 11 (64-bit)*
- *Linux (Ubuntu 20.04 and above)*

Development Environment:

- *Java Development Kit (JDK) 11 or higher*
- *NetBeans, Eclipse, or IntelliJ IDEA*

Network Requirements:

- *Reliable UDP communication with ports available (default port: 5000)*

Additional Software Components:

- *Java Runtime Environment (JRE) 11 or higher*
- *Standard networking libraries in Java (java.net package)*

2.5 Design and Implementation Constraints

- ***UDP Protocol****: The application uses **UDP** for communication, which means no guaranteed delivery, ordering, or error correction.*
- ***Client-Server Model****: A central server is required to relay messages between clients. Direct client-to-client communication is not supported.*
- ***Error Handling****: Limited error detection since UDP does not provide built-in acknowledgment mechanisms.*
- ***Programming Language****: The application is developed using **Java** with **DatagramSocket** and **DatagramPacket** classes.*
- ***Network Dependency****: Requires stable network connections for efficient message delivery.*

2.6 User Documentation

*The following user documentation components will be provided with the **UDP-based Client-to-Client Chat Application**:*

- ***User Manual:***
  - *Step-by-step instructions for installing and running the chat application.*
  - *Detailed explanation of commands (GET, SEND, EXIT).*
  - *Troubleshooting guide for common issues.*
- ***Tutorial:***

- *A short, interactive tutorial guiding users through connecting to the server, sending messages, and disconnecting.*

Delivery Formats: *PDF for user manuals and tutorials.*

2.7 Assumptions and Dependencies

- *Java JDK 8 or above is installed.*
- Network connectivity is stable for UDP communication.

3. External Interface Requirements

3.1 User Interfaces

- ***Development Environment****:*
  - *The chat application will be executed in* ***NetBeans IDE*** *for development, testing, and debugging.*
  - *A simple* ***console-based interface*** *will be used for client interactions.*
- ***Screen Layout****:*
  - ***Main Area****: Displays received messages and system notifications.*
  - ***Input Section****: Allows users to enter text messages or commands (e.g., GET, SEND).*
  - ***Status Messages****: Displays connection status, errors, or acknowledgments.*

3.2 Hardware Interfaces

None.

3.3 Software Interfaces

- ***Operating System:*** *The application runs on Windows, Linux, or macOS.*
- ***Programming Language:*** *Developed using Java.*
- ***IDE:*** *Implemented and executed using NetBeans IDE.*
- ***Networking Protocol:*** *Utilizes UDP (User Datagram Protocol) for message communication.*
- ***External Libraries:*** *Uses standard Java Networking libraries (java.net and java.io) for socket communication.*
- ***Data Management:*** *The server maintains temporary client session data using in-memory structures (e.g., HashMaps).*
- ***Communication:***
  - *Clients send messages to the server using UDP packets.*
  - *The server forwards messages to the appropriate client using UDP packets.*

3.4 Communications Interfaces

- ***UDP Protocol:*** *The chat system uses UDP (User Datagram Protocol) for communication between clients and the server.*
- ***Message Format:***

  Messages follow a simple text format using predefined commands-
  - *GET to retrieve the client list.*
  - *SEND <client_id> <message> to send a message to a specific client.*
  - *EXIT to disconnect from the server.*
- ***Network Communication:*** *The system uses localhost (127.0.0.1) and port 5000 for both client and server operations in a local environment.*
- ***Development Environment:*** *The program is developed and executed in NetBeans IDE using Java.*

4. System Use Cases

4.1 Client Registration

4.1.1 Description and Priority

- ***Description:*** *This feature allows clients to register with the server and receive a unique Client ID for communication.*
- ***Priority:*** *High*

4.1.2 Stimulus/Response Sequences

S: *Client sends a registration request to the server.*
R: *Server assigns a Client ID and confirms the registration.*

4.1.3 Functional Requirements:

REQ-1: *The system shall assign a unique Client ID to each connected client.*
REQ-2: *The server shall send a confirmation message with the Client ID to the client.*
REQ-3: *If the server reaches the client limit, it shall send an error message to the client.*

4.2 Sending Messages

4.2.1 Description and Priority

- ***Description:*** *This feature allows clients to send messages to other connected clients using the server as an intermediary.*
- ***Priority:*** *High*

4.2.2 Stimulus/Response Sequences

S: *Client sends a message using the format SEND <client_id> <message>.*
R: *The server forwards the message to the target client.*

4.2.3 Functional Requirements:

REQ-1: *The system shall validate the Client ID of the recipient.*
REQ-2: *The server shall forward the message to the specified client.*
REQ-3: *If the recipient ID is invalid, the server shall send an error message to the sender.*
REQ-4: *The server shall maintain message delivery logs for debugging purposes.*

4.3 Client List Retrieval

4.3.1 Description and Priority

- ***Description:*** *This feature allows clients to retrieve a list of all currently connected clients.*
- ***Priority:*** *Medium*

4.3.2 Stimulus/Response Sequences

S: *Client sends a GET request.*
R: *The server sends a list of connected client IDs.*

4.3.3 Functional Requirements:

REQ-1: *The system shall return a list of all active Client IDs upon receiving a GET request.*
REQ-2: *If no other clients are connected, the server shall notify the client.*
REQ-3: *The system shall ensure the client list is accurate and updated in real-time.*

4.4 Client Disconnection

4.4.1 Description and Priority

- ***Description:*** *This feature allows clients to disconnect from the server safely using the EXIT command.*
- ***Priority:*** *High*

4.4.2 Stimulus/Response Sequences

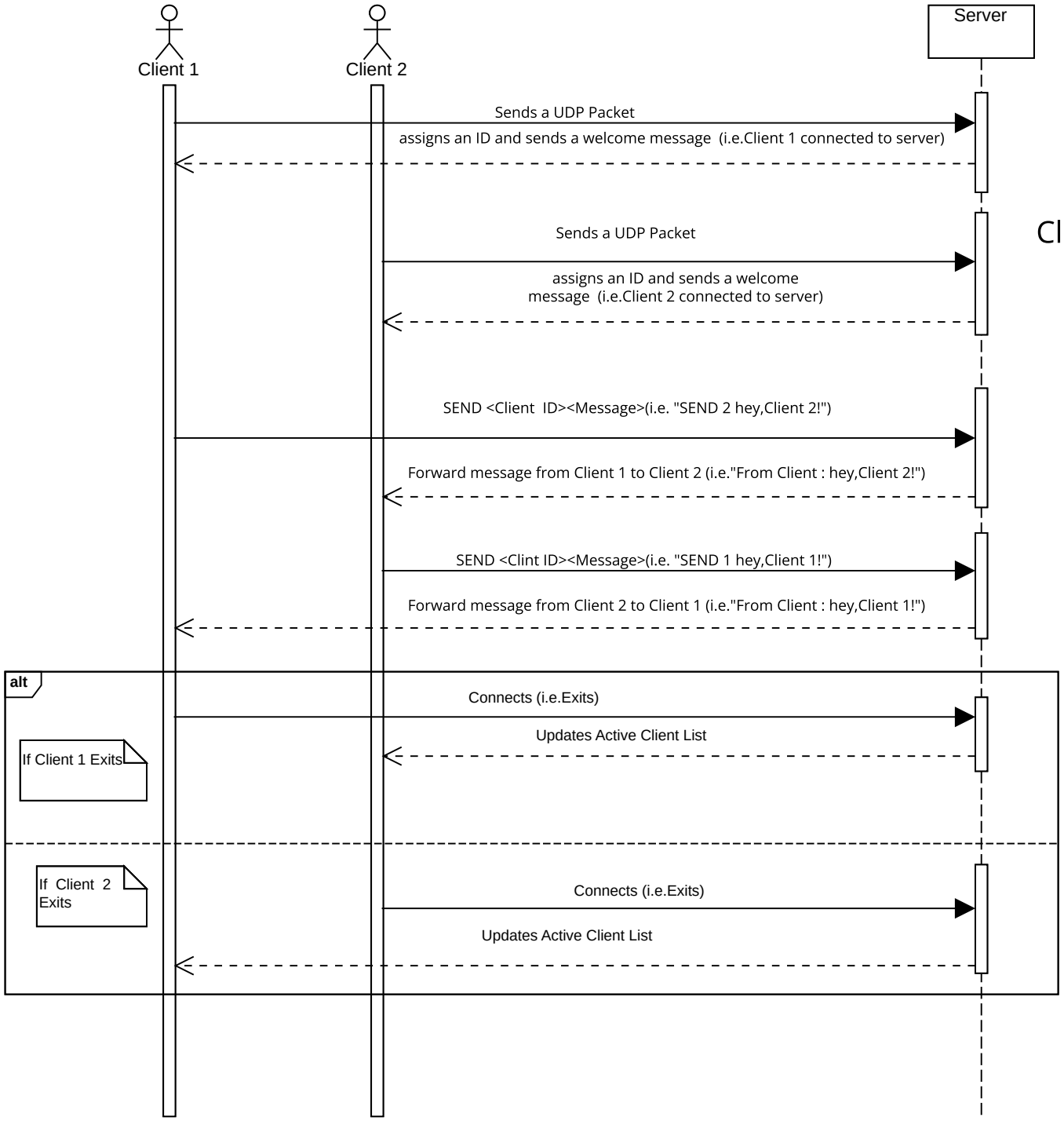
S: *Client sends an EXIT command.*
R: *The server acknowledges the disconnection and removes the client from the connected client list.*

4.4.3 Functional Requirements

REQ-1: *The system shall recognize and process the EXIT command from clients.*
REQ-2: *The server shall remove the client's details from the connected client list.*
REQ-3: *The system shall send a confirmation message to the client upon successful disconnection.*

Appendix A: Glossary

- ***REQ:****Requirements*
- ***S:****Stimulus*
- ***R:****Response*



Client -To- Client Chat Program (UDP Socket Based)

The design, code and SRS of the client-to-client chat program using both versions:

- The multithreaded way
- The UDP socket based

Also add graphical interface to enter data to be passed or setting up any other parameters like IP address or port number etc.

CODE:

TCPServer1GUI.java

```
package tcpserver1gui;

import javax.swing.*;

import java.awt.*;

import java.io.*;

import java.net.*;

import java.util.*;


public class TCPServer1GUI{

   private static int clientIdCounter = 1;

   private static final Map<Integer, ClientHandler> clients = new HashMap<>();

   private static JTextArea textArea;

   private static volatile boolean running = true;  // Flag to control server execution


   public static void main(String[] args) {

      JFrame frame = new JFrame("TCP Chat Server");

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.setSize(400, 300);


      textArea = new JTextArea();

      textArea.setEditable(false);

      frame.add(new JScrollPane(textArea), BorderLayout.CENTER);

      frame.setVisible(true);


      startServer();
```

```
}

private static void startServer() {
    new Thread(() -> {
        try (ServerSocket serverSocket = new ServerSocket(5000)) {
            log("Server started on port 5000...");

            while (running) {
                Socket clientSocket = serverSocket.accept();
                ClientHandler handler = new ClientHandler(clientSocket, clientIdCounter);
                synchronized (clients) {
                    clients.put(clientIdCounter, handler);
                }
                new Thread(handler).start();
                log("Client " + clientIdCounter + " connected.");
                clientIdCounter++;
            }
        } catch (IOException e) {
            log("Error: " + e.getMessage());
        }
    }).start();
}

private static void log(String message) {
    SwingUtilities.invokeLater(() -> textArea.append(message + "\n"));
}

static class ClientHandler implements Runnable {
    private Socket socket;
    private int clientId;
    private PrintWriter out;
```

```
private BufferedReader in;
private Thread readThread, writeThread;
private volatile boolean clientRunning = true;  // Flag to control client threads

public ClientHandler(Socket socket, int clientId) {
    this.socket = socket;
    this.clientId = clientId;
}

@Override
public void run() {
    try {
        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);
        out.println("Welcome! Your Client Id: " + clientId);

        startReadThread();
        startWriteThread();

        // Wait for threads to finish
        readThread.join();
        writeThread.join();
    } catch (IOException | InterruptedException e) {
        log("Client " + clientId + " disconnected.");
    } finally {
        disconnectClient();
    }
}

private void startReadThread() {
    readThread = new Thread(() -> {
```

```
        try {
            String message;
            while (clientRunning && (message = in.readLine()) != null) {
                if (message.equalsIgnoreCase("GET")) {
                    sendClientList();
                } else if (message.startsWith("SEND")) {
                    sendMessageToClient(message);
                } else if (message.equalsIgnoreCase("exit")) {
                    disconnectClient();
                    break;
                } else {
                    out.println("Invalid Command! Use 'GET' or 'SEND <client_id> <message>'");
                }
            }
        } catch (IOException e) {
            log("Client " + clientId + " disconnected.");
        }
    });
    readThread.start();
}

private void startWriteThread() {
    writeThread = new Thread(() -> {
        try {
            while (clientRunning) {
                Thread.sleep(100); // Prevents CPU overuse
            }
        } catch (InterruptedException e) {
            log("Write thread interrupted.");
        }
    });
```

```
    writeThread.start();
}

private void sendClientList() {
    StringBuilder clientList = new StringBuilder("Connected Clients: ");
    synchronized (clients) {
        for (Integer id : clients.keySet()) {
            clientList.append(id).append(" ");
        }
    }
    out.println(clientList.toString().trim());
}

private void sendMessageToClient(String message) {
    String[] parts = message.split(" ", 3);
    if (parts.length < 3) {
        out.println("Invalid SEND format. Use: SEND <client_id> <message>");
        return;
    }
    try {
        int recipientId = Integer.parseInt(parts[1]);
        String msg = parts[2];
        synchronized (clients) {
            if (clients.containsKey(recipientId)) {
                clients.get(recipientId).out.println("From Client " + clientId + ": " + msg);
                log("Client " + clientId + " sent to Client " + recipientId + ": " + msg);
            } else {
                out.println("Client " + recipientId + " not found.");
            }
        }
    } catch (NumberFormatException e) {
```

```
            out.println("Invalid Client ID");
        }
    }

    private void disconnectClient() {
        clientRunning = false;
        try {
            socket.close();
            synchronized (clients) {
                clients.remove(clientId);
            }
            log("Client " + clientId + " disconnected.");
        } catch (IOException e) {
            log("Error closing connection: " + e.getMessage());
        }
    }
  }
}
```

TCPClient1GUI.java

```
package tcpclient1gui;
import javax.swing.*;
import java.awt.*;
import java.io.*;
import java.net.*;

public class TCPClient1GUI {
   private JTextArea chatArea;
   private JTextField messageField;
   private PrintWriter out;
   private BufferedReader in;
   private Socket socket;
```

```
private Thread readThread, writeThread;
private volatile boolean running = true;  // Flag to control thread execution

public TCPClient1GUI() {
    JFrame frame = new JFrame("TCP Chat Client");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(400, 400);

    JPanel panel = new JPanel(new BorderLayout());
    chatArea = new JTextArea();
    chatArea.setEditable(false);
    panel.add(new JScrollPane(chatArea), BorderLayout.CENTER);

    JPanel inputPanel = new JPanel(new BorderLayout());
    messageField = new JTextField();
    JButton sendButton = new JButton("Send");
    inputPanel.add(messageField, BorderLayout.CENTER);
    inputPanel.add(sendButton, BorderLayout.EAST);

    panel.add(inputPanel, BorderLayout.SOUTH);
    frame.add(panel);
    frame.setVisible(true);

    setupConnection();

    sendButton.addActionListener(e -> sendMessage());
    messageField.addActionListener(e -> sendMessage());
}

private void setupConnection() {
    String ip = JOptionPane.showInputDialog("Enter Server IP:", "localhost");
```

```
    String portString = JOptionPane.showInputDialog("Enter Server Port:", "5000");

    try {
        int port = Integer.parseInt(portString);
        socket = new Socket(ip, port);
        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);
        chatArea.append("Connected to server\n");

        startReadThread();
        startWriteThread();
    } catch (IOException | NumberFormatException e) {
        JOptionPane.showMessageDialog(null, "Connection failed: " + e.getMessage());
    }
}

private void startReadThread() {
    readThread = new Thread(() -> {
        try {
            String serverMessage;
            while (running && (serverMessage = in.readLine()) != null) {
                chatArea.append(serverMessage + "\n");
            }
        } catch (IOException e) {
            chatArea.append("Disconnected from server\n");
        }
    });
    readThread.start();
}

private void startWriteThread() {
```

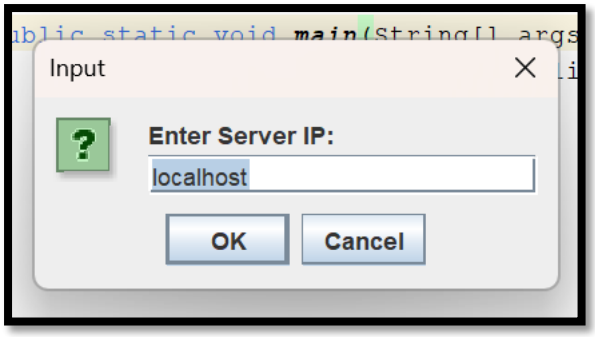
```
    writeThread = new Thread(() -> {
        try {
            while (running) {
                synchronized (messageField) {
                    messageField.wait(); // Wait until a message is entered
                }
                String message = messageField.getText().trim();
                if (!message.isEmpty()) {
                    out.println(message);
                    if (message.equalsIgnoreCase("exit")) {
                        disconnect();
                        break;
                    }
                    messageField.setText("");
                }
            }
        } catch (InterruptedException e) {
            chatArea.append("Write thread interrupted.\n");
        }
    });
    writeThread.start();
}

private void sendMessage() {
    synchronized (messageField) {
        messageField.notify(); // Wake up the Write Thread to send the message
    }
}

private void disconnect() {
    running = false;
```

```
        try {
            socket.close();
        } catch (IOException e) {
            chatArea.append("Error closing socket\n");
        }
        System.exit(0);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(TCPClient1GUI::new);
    }
}
```

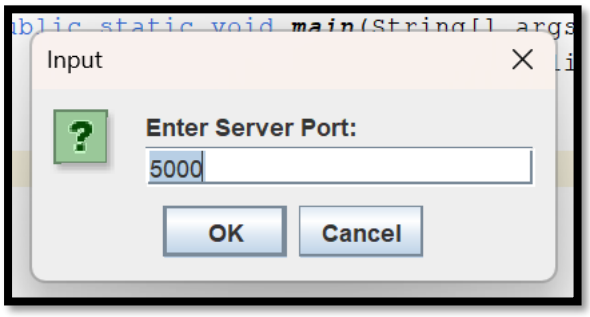
INPUT:

Client 1:

Click “OK” to take Server IP as input

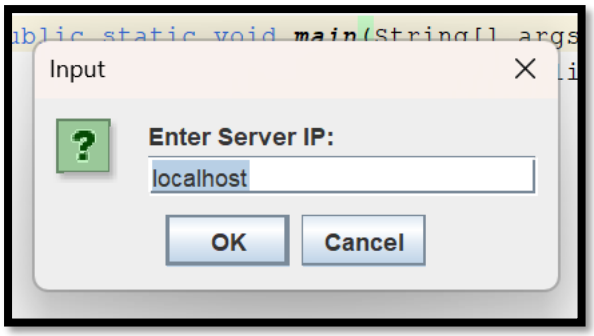


Click “OK” to take Server Port as input



Client 2:

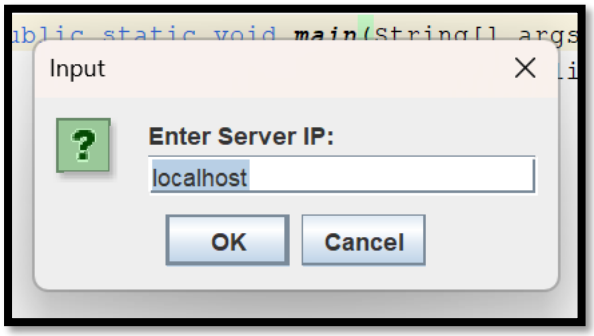
Click “OK” to take Server IP as input



Click “OK” to take Server Port as input

Client 3:

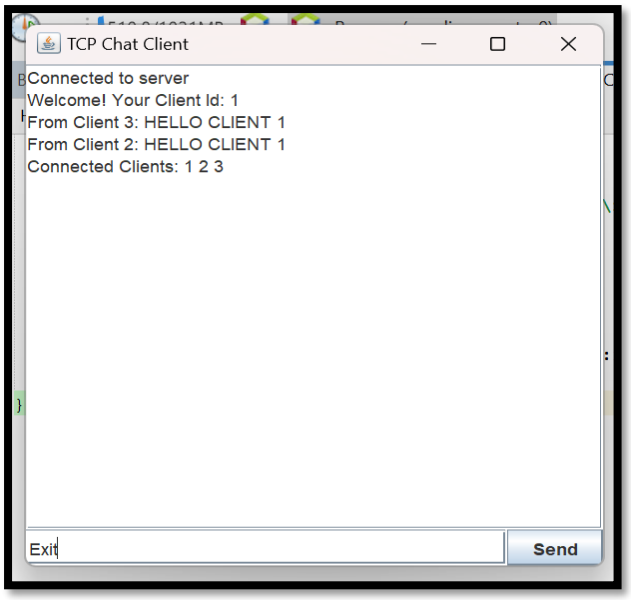
Click “OK” to take Server IP as input



Click “OK” to take Server Port as input

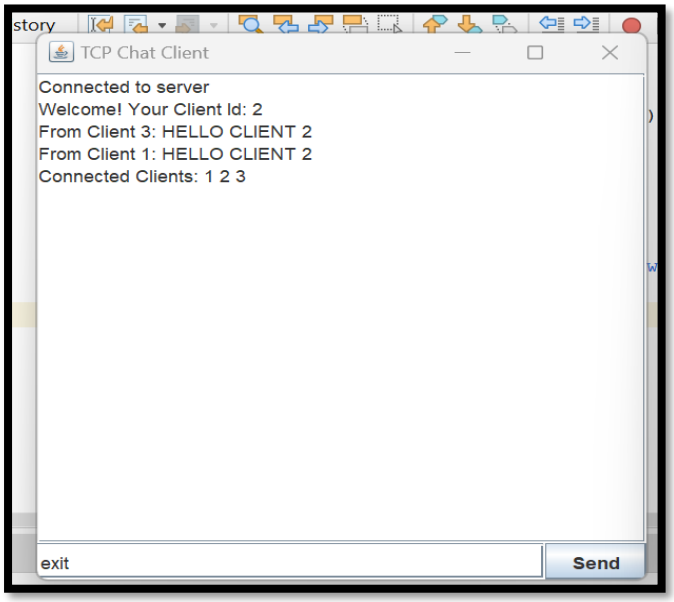
OUTPUT:

Client 1:



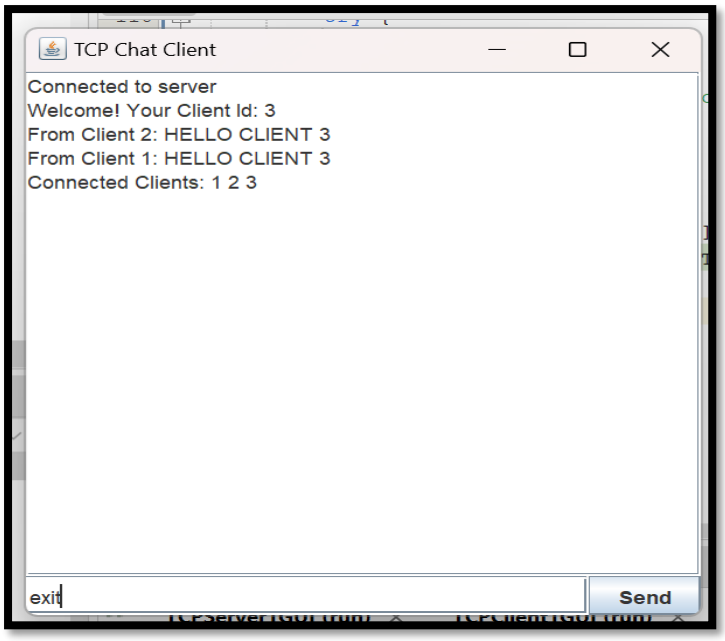
When “Send” is clicked both read and write threads stop, and the client 1 disconnects.

Client 2:



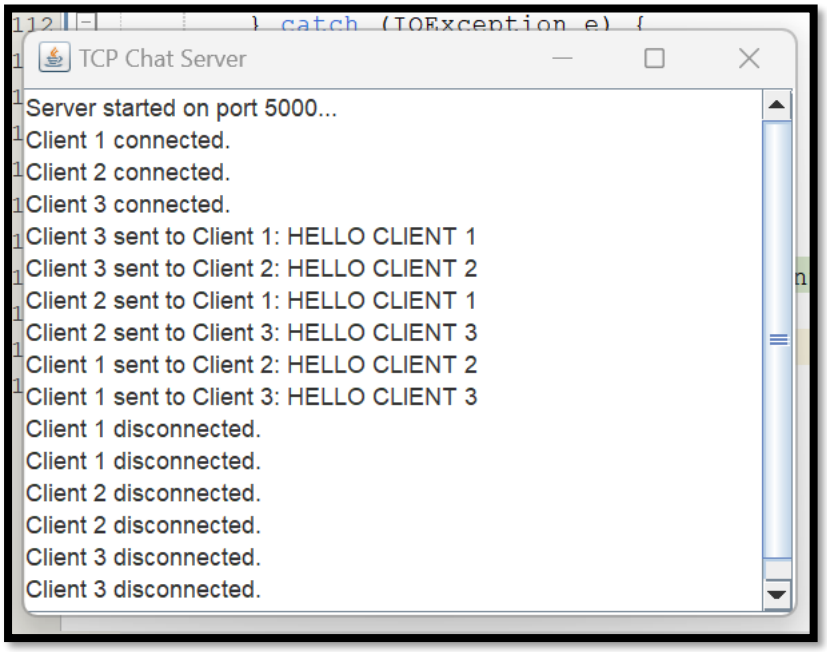
When “Send” is clicked both read and write threads stop, and the client 2 disconnects.

Client 3:



When “Send” is clicked both read and write threads stop, and the client 3 disconnects.

Server:



CODE:

UDPServerGUI.java

```
package udpservergui;

import javax.swing.*;

import java.awt.*;

import java.net.*;
```

```
import java.util.HashMap;
import java.util.Map;

public class UDPServerGUI {
    private static final int SERVER_PORT = 5000;
    private static Map<InetSocketAddress, Integer> clients = new HashMap<>();
    private static int clientIdCounter = 1;
    private static JTextArea logArea;

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> createGUI());
        startServer();
    }

    private static void createGUI() {
        JFrame frame = new JFrame("UDP Chat Server");
        frame.setSize(500, 400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());

        logArea = new JTextArea();
        logArea.setEditable(false);
        frame.add(new JScrollPane(logArea), BorderLayout.CENTER);

        frame.setVisible(true);
    }

    private static void startServer() {
        new Thread(() -> {
            try (DatagramSocket serverSocket = new DatagramSocket(SERVER_PORT)) {
                log("Server started on port " + SERVER_PORT);
```

```
            byte[] receiveBuffer = new byte[1024];
            while (true) {
                DatagramPacket receivePacket = new DatagramPacket(receiveBuffer,
receiveBuffer.length);
                serverSocket.receive(receivePacket);
                String message = new String(receivePacket.getData(), 0, receivePacket.getLength());

                InetSocketAddress clientAddress = new InetSocketAddress(
                        receivePacket.getAddress(), receivePacket.getPort());

                if (!clients.containsKey(clientAddress)) {
                    clients.put(clientAddress, clientIdCounter++);
                    sendMessage(serverSocket, "Your Client ID: " + clients.get(clientAddress),
clientAddress);
                    log("New client connected. Assigned ID: " + clients.get(clientAddress));
                }

                handleClientMessage(serverSocket, message, clients.get(clientAddress), clientAddress);
            }
        } catch (Exception e) {
            log("Server error: " + e.getMessage());
        }
    }).start();
  }

  private static void handleClientMessage(DatagramSocket serverSocket, String message, int clientId,
InetSocketAddress clientAddress) {
    try {
      if (message.equalsIgnoreCase("GET")) {
        sendClientList(serverSocket, clientAddress);
      } else if (message.startsWith("SEND")) {
```

```
            sendMessageToClient(serverSocket, message, clientId);
        } else if (message.equalsIgnoreCase("exit")) {
            clients.remove(clientAddress);
            log("Client " + clientId + " disconnected.");
        } else {
            sendMessage(serverSocket, "Invalid Command! Use 'GET' or 'SEND <client_id> <message>'",
clientAddress);
        }
      } catch (Exception e) {
        log("Error handling message: " + e.getMessage());
      }
    }

    private static void sendClientList(DatagramSocket serverSocket, InetSocketAddress clientAddress) {
      StringBuilder clientList = new StringBuilder("Connected Clients: ");
      for (Integer id : clients.values()) {
        clientList.append(id).append(" ");
      }
      sendMessage(serverSocket, clientList.toString().trim(), clientAddress);
    }

    private static void sendMessageToClient(DatagramSocket serverSocket, String message, int
senderId) {
      String[] parts = message.split(" ", 3);
      if (parts.length < 3) return;

      try {
        int recipientId = Integer.parseInt(parts[1]);
        String msg = parts[2];
        for (Map.Entry<InetSocketAddress, Integer> entry : clients.entrySet()) {
          if (entry.getValue() == recipientId) {
            sendMessage(serverSocket, "From Client " + senderId + ": " + msg, entry.getKey());
```

```
                return;
            }
        }
    } catch (NumberFormatException ignored) {}
}

private static void sendMessage(DatagramSocket serverSocket, String message, InetSocketAddress clientAddress) {
    try {
        byte[] sendData = message.getBytes();
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                clientAddress.getAddress(), clientAddress.getPort());
        serverSocket.send(sendPacket);
    } catch (Exception e) {
        log("Error sending message.");
    }
}

private static void log(String message) {
    SwingUtilities.invokeLater(() -> logArea.append(message + "\n"));
}
}
```

UDPClientGUI.java

```
package udpclientgui;
import javax.swing.*;
import java.awt.*;
import java.net.*;

public class UDPClientGUI {
    private DatagramSocket socket;
    private InetAddress serverAddress;
```

```
private int serverPort;
private volatile boolean running = true;
private JTextArea chatArea;
private JTextField messageField;
private int clientId = -1; // Client ID assigned by server
private JFrame frame;

public UDPClientGUI(String serverIP, int serverPort) throws Exception {
    this.serverAddress = InetAddress.getByName(serverIP);
    this.serverPort = serverPort;
    this.socket = new DatagramSocket();

    createGUI();
    startReadThread();
}

private void createGUI() {
    frame = new JFrame("UDP Chat Client");
    frame.setSize(500, 400);
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.setLayout(new BorderLayout());

    chatArea = new JTextArea();
    chatArea.setEditable(false);
    frame.add(new JScrollPane(chatArea), BorderLayout.CENTER);

    JPanel bottomPanel = new JPanel(new BorderLayout());
    messageField = new JTextField();
    JButton sendButton = new JButton("Send");
    sendButton.addActionListener(e -> sendMessage());
```

```
        bottomPanel.add(messageField, BorderLayout.CENTER);
        bottomPanel.add(sendButton, BorderLayout.EAST);
        frame.add(bottomPanel, BorderLayout.SOUTH);

        frame.setVisible(true);
    }

    private void startReadThread() {
        new Thread(() -> {
            try {
                byte[] receiveBuffer = new byte[1024];
                while (running) {
                    DatagramPacket receivePacket = new DatagramPacket(receiveBuffer,
receiveBuffer.length);
                    socket.receive(receivePacket);
                    String message = new String(receivePacket.getData(), 0, receivePacket.getLength());

                    if (message.startsWith("Your Client ID:")) {
                        clientId = Integer.parseInt(message.split(":")[1].trim());
                        chatArea.append("Assigned Client ID: " + clientId + "\n");
                    } else {
                        chatArea.append(message + "\n");
                    }
                }
            } catch (Exception e) {
                chatArea.append("Disconnected from server.\n");
            } finally {
                closeClient();
            }
        }).start();
    }
```

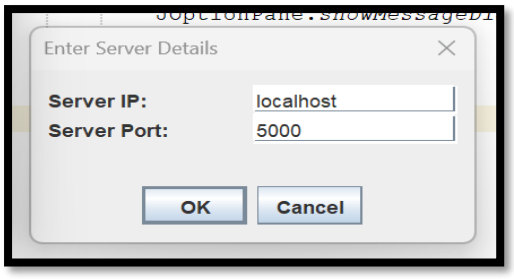
```
    private void sendMessage() {
        try {
            String message = messageField.getText();
            if (message.isEmpty()) return;
            messageField.setText("");
            byte[] sendData = message.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
serverAddress, serverPort);
            socket.send(sendPacket);


            if (message.equalsIgnoreCase("exit")) {
                running = false;
                socket.close();
                closeClient();
            }
        } catch (Exception e) {
            chatArea.append("Error sending message.\n");
        }
    }


    private void closeClient() {
        running = false;
        if (socket != null && !socket.isClosed()) {
            socket.close();
        }
        SwingUtilities.invokeLater(() -> {
            frame.dispose(); // Close the GUI window when disconnected
        });
    }
```

```
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            try {
                JPanel inputPanel = new JPanel(new GridLayout(3, 2));
                JTextField ipField = new JTextField("localhost");
                JTextField portField = new JTextField("5000");

                inputPanel.add(new JLabel("Server IP:"));
                inputPanel.add(ipField);
                inputPanel.add(new JLabel("Server Port:"));
                inputPanel.add(portField);

                int result = JOptionPane.showConfirmDialog(null, inputPanel, "Enter Server Details",
                        JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE);

                if (result == JOptionPane.OK_OPTION) {
                    String serverIP = ipField.getText();
                    int serverPort = Integer.parseInt(portField.getText());
                    new UDPClientGUI(serverIP, serverPort);
                }
            } catch (Exception e) {
                JOptionPane.showMessageDialog(null, "Client error: " + e.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
            }
        });
    }
}
```

INPUT:

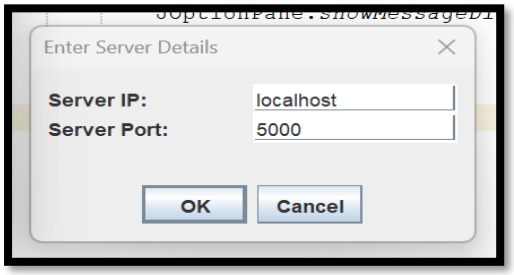
Client 1:

Click “OK” to take Server IP and Server Port as input



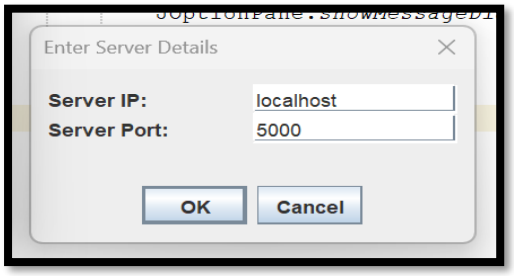
Client 2:

Click “OK” to take Server IP and Server Port as input



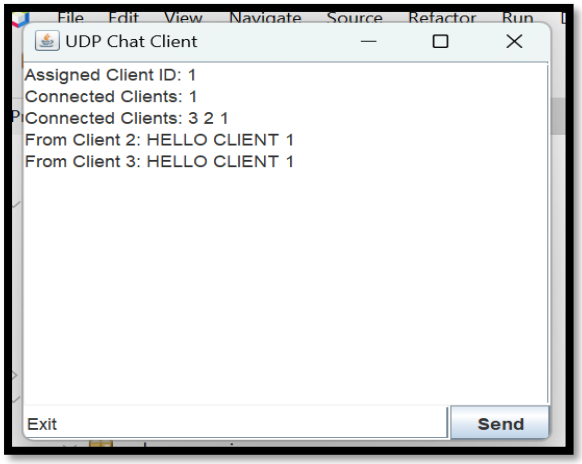
Client 3:

Click “OK” to take Server IP and Server Port as input



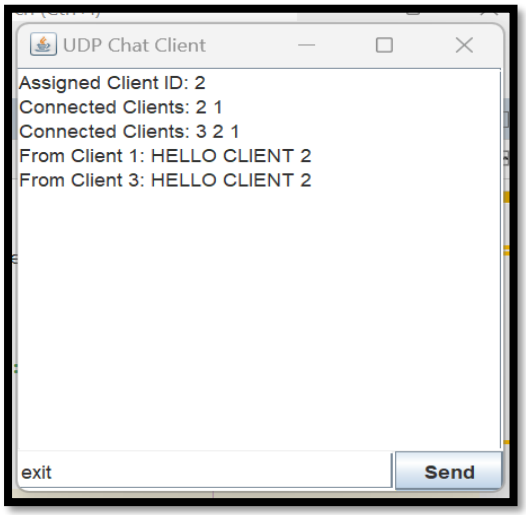
OUTPUT:

Client 1:



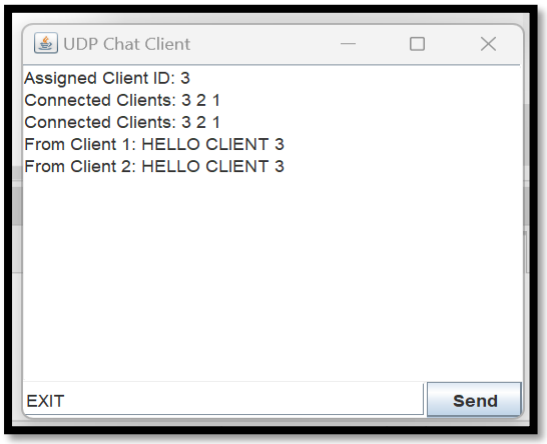
When “Send” is clicked both read(main) and write(separate) threads stop, and the client 1 disconnects.

Client 2:



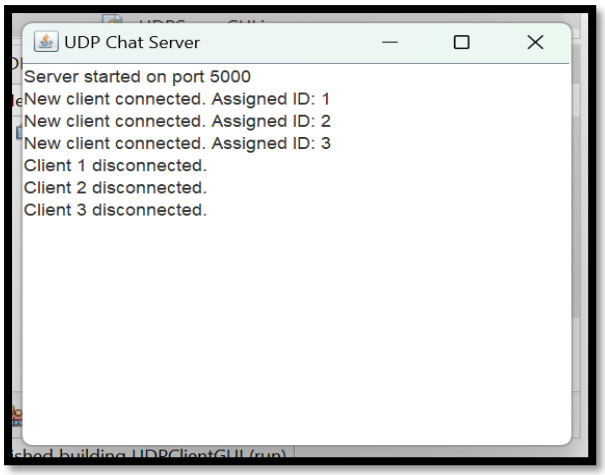
When “Send” is clicked both read(main) and write(separate) threads stop, and the client 2 disconnects.

Client 3:



When “Send” is clicked both read(main) and write(separate) threads stop, and the client 3 disconnects.

Server:


UDP Chat Server
Server started on port 5000
New client connected. Assigned ID: 1
New client connected. Assigned ID: 2
New client connected. Assigned ID: 3
Client 1 disconnected.
Client 2 disconnected.
Client 3 disconnected.