SERVER-CLIENT CHATROOM USING JAVA

MINOR PROJECT REPORT

By

AKSHIT LABH (RA2211028010049)
MEDHIR ARYAN (RA221102801050)

Under the guidance of

Dr. R. Kayalvizhi

In partial fulfilment for the Course

of

21CSC203P – ADVANCED PROGRAMMING PRACTICE

in **CLOUD COMPUTING**

DEPARTMENT OF NETWORKING AND COMMUNICATIONS

FACULTY OF ENGINEERING AND TECHNOLOGY

SCHOOL OF COMPUTING

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR

NOVEMBER 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this minor project report for the course **21CSC203P ADVANCED PROGRAMMING PRACTICE** entitled in "**SERVER-CLIENT CHATROOM USING JAVA**" is the bonafide work of **Akshit Labh (RA2211028010049)** and **Medhir Aryan (RA2211028010050)** who carried out the work under my supervision.

SIGNATURE

Dr. R. Kayalvizhi
Assistant Professor
Department of Networking
and Communications
SRM Institute of Science and
Technology, Kattankulathur

SIGNATURE

Dr. Annapurani K
HOD
Department of Networking
and Communications
SRM Institute of Science and
Technology, Kattankulathur

ABSTRACT

This project presents the design and implementation of a real-time chat application that leverages Java's powerful capabilities for networking, Graphical and multi-threaded user interface. It also provides hands-on experience on how to create a chat room use the Swing framework to create a user-friendly GUI, use threads for concurrent user connections, and leverage Ability to program Java sockets for network communication.

The Swing GUI provides an intuitive and interactive user interface, allowing users to easily participate in text conversations. Multithreading is used to facilitate the simultaneous interaction of multiple clients, thereby avoiding server congestion and ensuring responsiveness. The network component of this project uses sockets to establish trusted server-client communications, allowing users to send and receive messages in real time over a local or wide area network. Throughout the project, a custom communication protocol is defined, allowing message exchange between client and server. The implementation includes exception handling, socket management, and ensuring smooth communication between server and client. In summary, this project provides a hands-on learning experience in network programming, GUI development, and threading concepts.

It serves as a practical demonstration of creating a functional chat room that can be deployed in a variety of scenarios where real-time communication is required. The use of Swing for the GUI, and the incorporation of streams and sockets make it a flexible and resource-efficient solution for building client-server chat applications in Java.

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable **Vice Chancellor Dr. C. Muthamizhchelvan**, for being the beacon in all our endeavors.

We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy,** for his encouragement.

We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V.Gopal,** for bringing out novelty in all executions.

We would like to express my heartfelt thanks to Chairperson, School of Computing **Dr. Revathi Venkataraman,** for imparting confidence to complete my course project

We wish to express my sincere thanks to **Course Audit Professors Dr. Vadivu. G , Professor, Department of Data Science and Business Systems and Dr. Sasikala. E Professor, Department of Data Science and Business Systems** and **Course Coordinators** for their constant encouragement and support.

We are highly thankful to our Course project Faculty **Dr. R.Kayalvizhi, Assistant Professor , Networking and Communications,** for her assistance, timely suggestion and guidance throughout the duration of this course project.

We extend my gratitude to **Dr. Annapurani K, Head of the Department, Department of Networking and Communications** and my Departmental faculties for their Support.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on me to complete my Course project.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Motivation

The motivation behind starting a project to develop a client-server chat room using Java while using Swing for GUI, streams and socket programming, originated from the vision the growing importance of real-time communication in the digital age. This project aims to meet the need for efficient and secure communication tools, such as chat and instant messaging applications.

Practical applications:

In today's connected world, the need for instant, real-time communication is more evident than ever. From businesses requiring seamless communication across teams to individuals connecting with friends and family around the world, chat apps have become an indispensable part of people's daily lives. By creating a Java chat room, we can develop a practical solution that meets this need.

Technical Skills:

The project provides an excellent opportunity to acquire and develop essential technical skills. By focusing on Java, we leverage a powerful, platform-independent programming language. By implementing streams and sockets, we delve deeper into the basic concepts of concurrent programming and network communication. These are highly sought after skills in the software development industry.

User-centered GUI:

The choice of Swing for GUI was driven by user-centered design. Swing provides a rich toolbox for creating user-friendly interfaces. Building GUIs with Swing not only improves our Java skills but also ensures that the chat application is visually appealing and easy to use, a key factor in the success of any tool.

Networking capacity:

Thread and socket programming is at the heart of creating a client-server chat room. Developing a deep understanding of these concepts is critical to creating powerful, efficient, and scalable networked applications. This project provides practical opportunities to master these concepts in a real-world context.

Flexibility:

Chat room is a versatile application with many applications in many different fields. It can be adapted for educational purposes, serving as a virtual classroom or for professional communication between colleagues.

By creating this chat room, we are creating a tool that can be customized to meet a variety of needs.

1.2 Objective

The main objective of this project is to design and implement a client-server chat room application in Java, with emphasis on the following objectives:

Real-time communication:

The main goal is to provide a platform that allows users to engage in instant text conversations. This functionality is essential for many applications, from team collaboration and customer support to personal communication. The importance of real-time communication cannot be overstated, as it promotes quick decision-making, immediate problem resolution, and meaningful human connections.

User-centered interface:

Another important goal was to design a user-centered graphical user interface (GUI) for the chat room. The GUI is the gateway to the chat room experience and has a direct impact on how users perceive and interact with the application. Using Swing, a standard library for creating Java desktop applications, ensures that the interface is intuitive and user-friendly. This goal is important to make chat rooms accessible to more users, regardless of their technical skills. The main goal is to provide a platform that allows users to engage in instant text conversations. This functionality is essential for many applications, from team collaboration and customer support to personal communication. The importance of real-time communication cannot be overstated, as it promotes quick decision-making, immediate problem resolution, and meaningful human connections.

Scalability:

The ability to handle multiple simultaneous connections is a key feature of an effective chat room. Scalability is essential to ensure that chat rooms remain functional even as the number of users increases. Achieving this goal requires implementing threading concepts, allowing parallel execution of tasks. Thanks to effective stream management, chat rooms can accommodate many users without the risk of congestion or server slowdown.

Custom communication protocols:

Successful operation of a chat room depends on the development of a custom communication protocol. This protocol serves as the language used by clients and servers to exchange messages. By designing a suitable protocol, the project aims to ensure that message exchange is efficient, reliable and secure. It also addresses data integrity and message flow issues, which are fundamental to a chat room running smoothly.

1.2 Problem Statement

The problem addressed by this project is the lack of a lightweight, database free, Java-based client-server chat room equipped with a Swing GUI capable of efficiently handling real-time communication.

The challenge is to provide a solution that is lightweight, efficient, and easy to deploy. Such a solution would be appealing to developers and organizations seeking to build communication tools without the overhead of a database system. This project aims to streamline the development process and reduce potential points of failure, thus ensuring a more robust and user-friendly communication platform.

In essence, the problem statement revolves around the need for a simplified yet highly functional chat application. Emphasizing efficiency and user-friendliness, this project seeks to provide a practical solution for real-time communication that can be employed by developers and organizations seeking a streamlined alternative.

1.2 Challenges

Developing a client-server chat room involves a variety of complex challenges, each of which needs to be considered thoughtfully and resolved effectively:

Concurrency management:

One of the foremost challenges is to manage multiple client connections simultaneously.

Concurrency management is essential to ensure that clients can communicate with each other without interfering with server operations.

The concept of threading is important to achieve this as it allows multiple tasks to be executed in parallel, thus avoiding server blocking.

Custom Protocol Design:

Developing a custom communication protocol is a complex task.

This protocol defines how clients and servers exchange messages, requiring careful consideration of data structure and format.

The challenge lies in creating a protocol that ensures efficient message exchange, data integrity, and message flow while ensuring security.

User authentication:

User authentication is a fundamental security component in any chat application.

The challenge was to implement a secure user authentication system. User credentials must be securely stored and verified, and authentication mechanisms must be robust to ensure that only authorized users have access to the chat room.

GUI Design:

Creating an attractive, responsive, and user-friendly graphical user interface (GUI) is a challenge that goes beyond the technical aspects of the project.

The GUI is the face of the chat room and it must be designed with the user experience in mind.

This not only involves choosing the right Swing components, but also ensuring that the interface is pleasing to the eye and easy to navigate.

Error handling:

The world of real-time communication is full of potential problems.

Managing network issues, client disconnections, and potential communications problems poses a significant challenge.

Error handling is essential for maintaining chat room reliability and stability in a variety of situations.

2. LITERATURE SURVEY

A client-server chat room is a popular and convenient network programming application in which multiple clients can connect to a central server to exchange messages in real time. This literature review explores the basic concepts and previous work involved in developing client-server chat rooms using Java.

In this context, the chat room application uses Swing for its graphical user interface (GUI), uses threads for concurrency, and relies on sockets for network communications.

1. Java Networking:

Java's comprehensive networking libraries, especially the java.net package, provide a solid foundation for developing client-server applications.

The use of sockets, an integral part of Java's networking API, allows establishing network connections. Sockets are an essential component for communication between client and server in a chat room application.

A study by Liang (2018) in the book "Introduction to Java Programming" covers the fundamentals of Java networking, emphasizing socket programming and creating a simple chat application.

2. GUI Development with Swing:

Swing, part of the Java Foundation Layer (JFC), is a complete and flexible toolkit for creating graphical user interfaces. Swing provides many components such as **JFrame**, **JPanel**, **JTextArea**, **JTextField** and **JButton**, which play an instrumental role in creating a user-friendly chat room GUI. The reference is the book “Swing” by Eckstein et al. (2003), provides insights into Swing's architecture and how to use it in creating rich Java applications.

3. Threading in Java:

Threading is a basic concept for concurrent programming in Java. In the context of a client-server chat room, multi-threading is essential to manage multiple client connections simultaneously without blocking the server. A seminal work by Goetz et al. (2006), “Java Concurrency in Practice” covers thread management and the complexity of concurrent programming in Java. The application of threading in client-server chat rooms is discussed in the article "Research on Multithreaded Servers for Network Services" by Schonfeld et al. (1993), explored the benefits of multithreading in server applications.

4. Client-Server Architecture:

Understanding client-server architecture is important for designing chat rooms.

A study by Tanenbaum and Van Steen (2017) in the book “Distributed Systems: Principles and Models” discusses various client-server models, including the centralized model used in chat room applications. The client-server architecture model ensures that the client communicates with a central server, which then routes and manages messages.

5. Custom Communication Protocol:

To establish effective communication between client and server, a custom communication protocol is often designed. This protocol defines the structure of messages, how they are transmitted, and how they are processed. Designing custom protocols is a common practice in chat room applications. “TCP/IP Sockets in C: A Practical Guide for Programmers” by Donahoo and Calvert (2009) covers the development of custom protocols for networked applications.

6. Error and exception handling:

In networked applications, error and exception handling is essential to ensure robust and reliable communications. Bloch's "Effective Java” (2017) provides an overview of best practices for exception handling in Java. Marciniak's (2009) work in "Software Engineering: Software Development Based on the Architecture" discusses the importance of error handling in distributed systems and network applications.

7. Security Considerations:

Although not part of the project requirements, security is an important aspect in real-world chat applications. If security is an issue, you should explore encryption and authentication mechanisms. "Java Network Programming" by Harold and Harding (2007) provides an overview of network security in Java applications.

3. REQUIREMENTS

3.1 Requirement Analysis

Developing a client-server chat room in Java is a comprehensive project that includes many different hardware and software requirements. This section describes these requirements in detail, reviewing project specifications, excluding databases, using Swing for the graphical user interface (GUI), leveraging threading concepts and use the socket to connect to the network.

3.2 Hardware Requirement

3.2.1 Server hardware:

Computer or server:

The server will require a computer or server with sufficient processing power, memory and storage capacity to manage multiple client connections simultaneously. A multi-core processor, enough RAM, and enough storage is recommended.

Network Interface:

A reliable network interface card (NIC) is essential for stable network communications.

Internet Connection:

A stable and high-speed Internet connection is required to host the server and ensure smooth communication with customers.

3.2.2 Client hardware:

Computer or device:

The customer will need access to a computer or device with basic hardware specifications capable of running Java applications. A wide range of devices, from desktops to laptops and even mobile devices, need to be supported.

Network Interface:

Customers must have a working network card to connect to the network.

Internet Connection:

The client must have an active Internet connection to access the chat room server.

3.3 Software Requirements:

3.3.1 Server-side software:

Operating system:

The server must run a Java-compatible operating system. Popular choices include Windows, Linux or macOS.

Java Development Kit (JDK):

You must install the JDK to compile and run Java programs.

The project may require Java SE (Standard Edition) or Java EE (Enterprise Edition) depending on the specific features you are implementing.

Java IDE (Integrated Development Environment): An IDE like Eclipse, IntelliJ IDEA, or NetBeans can make development, debugging, and testing much easier.

Java Socket Programming Library:

Java provides built-in libraries for socket programming that you will use to establish and manage network connections.

Swing Library:

Since you are using Swing for GUI, you need to integrate the Java Swing library into your development environment.

3.3.2 Client-side software:

Operating systems:

Clients can use different operating systems, so your Java application must be platform independent. It is essential to ensure that your Java code is cross-platform compatible.

Java Runtime Environment (JRE):

Customers need a JRE to run Java applications.

Most modern operating systems come with a JRE pre-installed, but it is important to check this compatibility.

Graphical User Interface (GUI):

Because you are using Swing, the clients must have Swing library components available. Swing is part of the standard Java library and is widely supported by the JRE.

3.3 Java Threading Concept

Threading is a basic requirement for this project. The server must handle multiple client connections simultaneously, and each client must run in its own thread to ensure the GUI remains responsive when listening for messages. Java provides the java.lang.Thread class and related APIs for thread management, and you must implement synchronization and thread management.

3.4 Socket Programming

The project involves socket programming to establish communication between a client and a server. Java provides the java.net package for socket-related classes and methods. Servers and clients must use socket programming to connect, send, and receive messages. The server must bind to a specific IP address and port, and the client must connect to the server's IP address and port.

4 ARCHITECTURE AND DESIGN

The Client-Server Chat Room project is designed to create a real-time chat application in Java. This chat room does not involve the use of a database but relies on Java's built-in networking capabilities, especially sockets, and uses the Swing framework for the graphical user interface. The concept of threads is important to this project because they allow multiple clients to connect and communicate with the server at the same time.

4.1 System Architecture

The project follows the classic client-server architecture. It consists of two main components:

Server: The server component is responsible for managing client connections, routing messages and maintaining the state of the chat room.

Client: The client component is responsible for connecting to the server and supporting user interaction through the GUI.

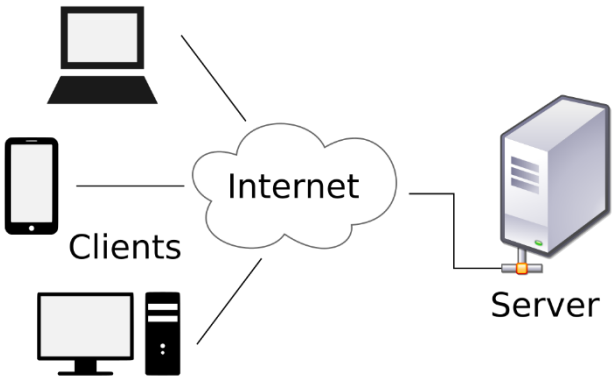

Figure 4.3.1 Server-Client architecture

4.2 Server design:

Socket programming:

The server listens on a specific port, waiting for incoming client connections. Once connected, a new thread will be spawned to handle each client. Java's ServerSocket and Socket classes are used for socket programming.

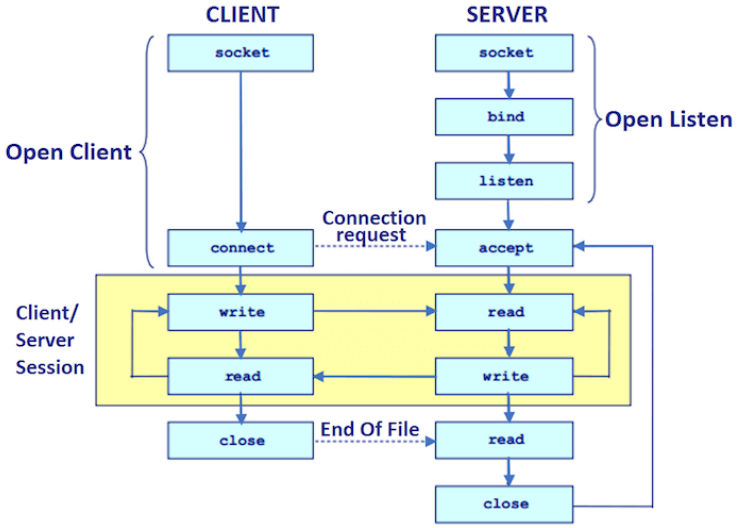

Figure 4.4.1 Socket Programming Concept

Communication Protocol:

The server uses a custom communication protocol in which messages are sent as plain text, with messages separated by a special character, such as a newline ("\n"). This design allows for client-side message analysis.

User Management:

The server maintains a list of connected clients and their respective output streams for sending messages. User information, such as the username, is stored in memory throughout the session. Disconnected clients are removed from the user list.

Error Handling:

Comprehensive error handling is built in to handle network problems, disconnections, and other potential problems that may arise during communications. The server is designed to handle these errors gracefully to ensure robustness.

Multithreading:

Multithreading is a central concept in server design, allowing multiple clients to connect simultaneously without blocking the server. Each customer is managed within its flow, ensuring that one customer's activity does not impact other customers.

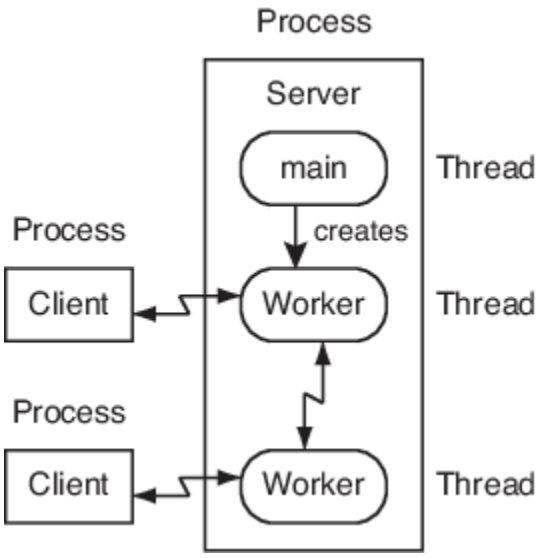

Figure 4.4.2 Threading Concept

4.3 Client Design

GUI with Swing:

Client-side GUI designed with Swing, providing a user-friendly interface. Swing's main components include JFrame, JPanel, JTextArea, JTextField, and JButton. GUI components allow users to view incoming messages, type and send their messages, and interact seamlessly with the chat room.

Threading for response:

To maintain GUI responsiveness when handling incoming messages and user interactions, a separate thread is used to receive messages from the server. This ensures that users can continue typing and sending messages without interruption.

Message Handling:

The client is designed to send messages to the server and display incoming messages from other clients. Messages are displayed in the chat area, and input is collected from a text field. Send buttons trigger the message transmission.

Event Handling:

Event listeners are incorporated to manage user interactions, such as button clicks and text input. These events are processed in the Swing event dispatch thread to keep the GUI responsive.

4.4 Communication Protocol:

A custom text-based communication protocol is implemented for this chat room. Messages are sent as plain text, with each message separated by a special character, usually a newline ("\n"). This simple protocol ensures efficient analysis on both the client and server sides.

4.5 Error handling and exception handling:

The project focuses on error handling and in-depth exception handling to maintain chat room stability. This includes troubleshooting network issues, client disconnections, and potential message formatting errors. By handling errors gracefully, the application ensures that unexpected events do not cause application crashes.

4.6 Testing:

The chatroom is thoroughly tested with multiple clients to validate its functionality. Testing scenarios include message sending, receiving, client disconnections, and the handling of simultaneous connections. Extensive testing ensures that the chatroom functions as expected and provides a reliable user experience.

4.7 Future Enhancements:

Potential future enhancements to the project include user authentication, the ability to join multiple chat rooms, and multimedia messaging support as shown photos and files.

5 IMPLEMENTATION

5.1 Chat Server Algorithm

- Import necessary Java libraries for networking and data structures.
- Define a class called ChatServer responsible for the server application.
- Define constants for the server's listening port (PORT), and create data structures to manage clients, user credentials, and chat history.
- clientMap: A Map that associates PrintWriter objects with usernames to track connected clients.
- userCredentials: A Map that stores predefined user credentials (username and password pairs).
- chatHistory: A List used to store the chat history.
- The server listens for incoming client connections, authenticates users based on the provided credentials, and maintains a chat history. Messages sent by clients are broadcast to all connected clients. If a client fails to authenticate, the server informs the client and closes the connection
- In the main method:
- Populate the userCredentials map with some predefined username-password pairs.
- Create a ServerSocket on the specified port (PORT) to listen for incoming client connections.
- Start a continuous loop to accept client connections.
- For each new client connection, create a ClientHandler thread to manage that client.
- Define an inner class ClientHandler responsible for handling each connected client:
- Initialize a Socket for communication with the client.
- In the run method.
- In the run method:
- Create a BufferedReader to read data from the client and a PrintWriter to send data to the client.
- Read the username and password sent by the client.
- Call the authenticateUser method to check if the provided credentials are valid.
- If the user is authenticated, inform the client by sending "AUTHENTICATED."
- Store the username for this client in the clientMap.

- Send the chat history to the client.
- Enter a message-receiving loop:
- Receive messages from the client, append them to the chat history, and broadcast them to all connected clients.
- If authentication fails, inform the client by sending "AUTH_FAILED" and close the connection. Implement a broadcast method within the ClientHandler class to send a message to all connected clients:
- Synchronize access to the clientMap.
- Iterate through the PrintWriter objects in clientMap and send the message to each client.
- Implement an authenticateUser method within the ClientHandler class to validate user credentials:
- Check if the provided username exists in the userCredentials map and if the associated password matches the one sent by the client.
- Return true if authentication is successful; otherwise, return false.
- The server listens for incoming client connections, authenticates users based on the provided credentials, and maintains a chat history. Messages sent by clients are broadcast to all connected clients. If a client fails to authenticate, the server informs the client and closes the connection

5.2 Chat Server Code

```
import java.io.*;
import java.net.*;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class ChatServer {
private static final int PORT = 5000;
private static Map<PrintWriter, String> clientMap = new HashMap<>();
private static Map<String, String> userCredentials = new HashMap<>();
private static List<String> chatHistory = new ArrayList<>(); // List to store chat history
```

```
public static void main(String[] args) {
 userCredentials.put("Medhir", "12");
userCredentials.put("Akshit", "12");
userCredentials.put("Rahul", "12");

try (ServerSocket serverSocket = new ServerSocket(PORT)) {
        System.out.println("Chat server is running...");

        while (true) {
         Socket clientSocket = serverSocket.accept();
        System.out.println("Client connected: " + clientSocket);

        Thread clientThread = new ClientHandler(clientSocket);
        clientThread.start();
        }
} catch (IOException e) {
        e.printStackTrace();
}
}

private static class ClientHandler extends Thread {
private Socket socket;
private PrintWriter writer;
private String username;

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

public void run() {
        try {

BufferedReaderreader=newBufferedReader(newInputStreamReader(socket.getInputStrem
()));
```

```
            writer = new PrintWriter(socket.getOutputStream(), true);

            String username = reader.readLine();
            String password = reader.readLine();

            if (authenticateUser(username, password)) {
            writer.println("AUTHENTICATED");
            this.username = username;

            synchronized (clientMap) {
             clientMap.put(writer, username);
            }

            // Send chat history to the client
            for (String historyMessage : chatHistory) {
                    writer.println(historyMessage);
            }

            String message;
            while ((message = reader.readLine()) != null) {
                    String formattedMessage = username + ": " + message;
                    System.out.println("Received: " + formattedMessage);
                    chatHistory.add(formattedMessage); // Store the message in chat history
                    broadcast(formattedMessage);
            }
            }
else {
            writer.println("AUTH_FAILED");
            socket.close();
            }
            } catch (IOException e) {
            e.printStackTrace();
            } finally {
            synchronized (clientMap) {
```

```
                clientMap.remove(writer);
                }

        try {
                socket.close();
                } catch (IOException e) {
                e.printStackTrace();
                }
                }
        }

        private void broadcast(String message) {
                synchronized (clientMap) {
                for (PrintWriter writer : clientMap.keySet()) {
                writer.println(message);
                }
                }
        }

        private boolean authenticateUser(String username, String password) {
                returnuserCredentials.containsKey(username)&&
        userCredentials.get(username).equals(password);
        }
        }
}
```

5.3 Client GUI Algorithm

- Import necessary Java libraries for the GUI, networking, and file handling.
- Define a class called ChatClientGUI that extends JFrame, representing the main chat client GUI application.
- Define constants for the server address and port number, which are used to establish a connection with the server.
- Create instance variables for the socket and a PrintWriter to send messages to the server.
- Initialize the user interface (UI) in the initUI method:

- Set the window title and size.
- Create UI components, including a text input field, a chat area (text area), a "Send" button for text messages, and a "Send File" button for sending files.
- Add event listeners to the "Send" button and text input field to send messages when the user presses Enter or clicks the "Send" button.
- Add the components to the main frame.
- Establish a connection to the server in the connectToServer method:
- Create a BufferedReader to read from the server and a PrintWriter to send messages.
- Prompt the user to enter their name and password using JOptionPane dialogs.
- Send the name and password to the server.
- Receive a response from the server to check if authentication was successful.
- If authentication fails, allow the user three attempts before exiting the application.
- If authentication is successful, show a message dialog and continue to the message receiving loop in a separate thread.
- Start a separate thread to continuously read messages from the server and display them in the chat area.
- Implement a method sendMessage to send text messages to the server:
- Get the text from the input field and send it to the server using the PrintWriter.
- Clear the input field after sending the message.
- Implement a method sendFile to send files to the server:
- Create a JFileChooser dialog to allow the user to select a file to send.
- If the user selects a file, open an output stream to the server and send the file using ObjectOutputStream.
- Implement a method ipaddress to retrieve the local machine's IP address. This method uses the NetworkInterface class to find the first site-local address.
- In the main method:
- Create a socket connection to the server using the defined server address and port.
- Use SwingUtilities.invokeLater to run the chat client GUI in the Event Dispatch Thread (EDT), ensuring thread safety for Swing components.

5.4 Client GUI Code

```
import javax.swing.*;
import javax.swing.text.DefaultCaret;
```

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;

public class ChatClientGUI extends JFrame {
    private static final String SERVER_ADDRESS = ipaddress();
    //private static final String SERVER_ADDRESS = "10.4.242.74";
    private static final int SERVER_PORT = 5000;

    private Socket socket; // Add a Socket field
    private PrintWriter writer;

    private JTextField inputField;
    private JTextArea chatArea;
    private JButton sendButton; // New button for sending messages
    private JButton sendFileButton; // Button for sending files

    public ChatClientGUI(Socket socket) {
        this.socket = socket;
        initUI();
        connectToServer();
    }

    private void initUI() {
        setTitle("Chat Client");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(400, 300);

        inputField = new JTextField();
        chatArea = new JTextArea();
        chatArea.setEditable(false);
```

```
        JScrollPane chatScrollPane = new JScrollPane(chatArea);
        DefaultCaret caret = (DefaultCaret) chatArea.getCaret();
        caret.setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);

        sendButton = new JButton("Send"); // Initialize the send button
        sendButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                sendMessage();
            }
        });
        inputField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                sendMessage();
            }
        });

        sendFileButton = new JButton("Send File");
        sendFileButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                sendFile();
            }
        });

        JPanel bottomPanel = new JPanel(new BorderLayout());
        bottomPanel.add(inputField, BorderLayout.CENTER);
        bottomPanel.add(sendButton, BorderLayout.EAST);
        bottomPanel.add(sendFileButton, BorderLayout.WEST); // Add the Send File button

        setLayout(new BorderLayout());
        add(chatScrollPane, BorderLayout.CENTER);
        add(bottomPanel, BorderLayout.SOUTH);

        setVisible(true);
    }
```

```
    private void connectToServer() {
        int loginAttempts = 0;

        while (loginAttempts < 3) {
            try {
                BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
                writer = new PrintWriter(socket.getOutputStream(), true);

                String name = JOptionPane.showInputDialog("Enter your name:");
                writer.println(name);
                String password = JOptionPane.showInputDialog("Enter your password:");
                writer.println(password);

                String response = reader.readLine();
                if (response.equals("AUTHENTICATED")) {
                    JOptionPane.showMessageDialog(this, "Authenticated successfully!");
                    break;
                } else if (response.equals("AUTH_FAILED")) {
                    loginAttempts++;
                    JOptionPane.showMessageDialog(this, "Authentication failed. Please try
again.");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        if (loginAttempts >= 3) {
            JOptionPane.showMessageDialog(this, "Authentication failed 3 times. Exiting...");
            System.exit(0);
        }
```

```
        Thread outputThread = new Thread(() -> {
            try {
                BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

                String message;
                while ((message = reader.readLine()) != null) {
                    chatArea.append(message + "\n");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        });

        outputThread.start();
    }

    private void sendMessage() {
        String message = inputField.getText();
        if (!message.isEmpty()) {
            writer.println(message);
            inputField.setText("");
        }
    }

    private void sendFile() {
        JFileChooser fileChooser = new JFileChooser();
        int choice = fileChooser.showOpenDialog(this);

        if (choice == JFileChooser.APPROVE_OPTION) {
            File selectedFile = fileChooser.getSelectedFile();
            System.out.println("Sending file: " + selectedFile);

            try {
```

```
            OutputStream outputStream = socket.getOutputStream();
            BufferedOutputStream bufferedOutputStream = new
BufferedOutputStream(outputStream);
            ObjectOutputStream objectOutputStream = new
ObjectOutputStream(bufferedOutputStream);

            FileInputStream fileInputStream = new FileInputStream(selectedFile);
            byte[] buffer = new byte[1024];
            int bytesRead;

            while ((bytesRead = fileInputStream.read(buffer)) != -1) {
               objectOutputStream.write(buffer, 0, bytesRead);
            }

            objectOutputStream.flush();
            objectOutputStream.close();
            fileInputStream.close();
         } catch (IOException e) {
            e.printStackTrace();
         }
      }
   }

   private static String ipaddress() {
      String ipadd = "";
      try {
         Enumeration<NetworkInterface> networkInterfaceEnumeration =
         NetworkInterface.getNetworkInterfaces();
         while (networkInterfaceEnumeration.hasMoreElements()) {
            for (InterfaceAddress interfaceAddress :
networkInterfaceEnumeration.nextElement()
                  .getInterfaceAddresses())
               if (interfaceAddress.getAddress().isSiteLocalAddress())
                  ipadd = interfaceAddress.getAddress().getHostAddress();
```

```
            }
            return ipadd;
        } catch (SocketException e) {
            e.printStackTrace();
        }
        return ipadd;
    }

    public static void main(String[] args) {
        try {
            Socket socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
            SwingUtilities.invokeLater(() -> new ChatClientGUI(socket));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

6 EXPERIMENT RESULTS

In this section we'll be exploring the output of the code and seeing the key points implemented in this project.

6.1 Output

Figure 6.1 GUI Prompt to enter name

The figure 6.1 shows the GUI prompt to enter the name. The user has to now enter their name which would be used as their user-name while chatting on the server. Entered user-name would be unique to each user using the application and cannot be used again by another user.

Figure 6.1 GUI output to enter password

The figure 6.2 shows that the user has to now enter his password for authentication. If authentication fails the program stops. Each user-name will have its unique password which will allow users to chat on the server.

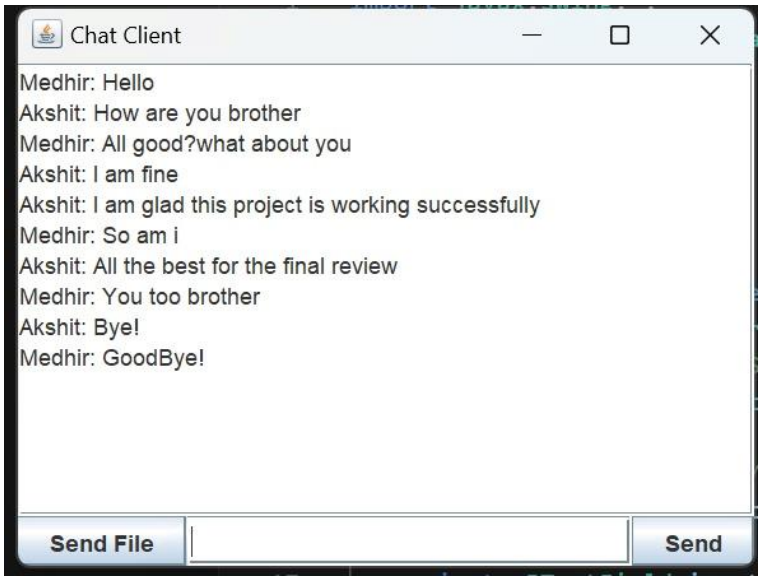

Figure 6.2 GUI Output of live chat-room

The figure 6.3 shows the output of how the chat GUI looks. While chatting users can communicate with each other in real time without any delay. Users can also transfer files via chat client to other users.

```
PS C:\Users\User\Documents\Python\Adzip>  & 'C:\Program Files\Java\jdk-20\bin\java.exe' '-agentlib:jdwp=transport=dt_socket,server=n
d=y,address=localhost:53364' '--enable-preview' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\User\AppData\Roaming\Code\
rkspaceStorage\d835de87795e40b14a54f233e826bca8\redhat.java\jdt_ws\jdt.ls-java-project\bin' 'ChatServer'
Chat server is running...
```

Figure 6.3 Server status in terminal

The figure 6.4 shows the chat server live and running successfully. The server status is required to be live in order to communicate via the network. If the server is terminated the clients won't be able to communicate between themselves.

6.2 Key topics covered

1. Master network programming:

Client-server architecture, implemented through socket programming, allows we explore the subtleties of network communication. We learned how to establish connections, exchange data between client and server, and handle network-related errors effectively.

2. Threading and concurrency:

A project's use of multithreading is important to handle multiple client connections simultaneously. By creating separate streams for each connected client, we ensure that the server can handle communication without blocking or slowing down.

3. GUI Development with Swing:

Using Swing for GUI provided a practical introduction to creating interactive desktop applications. We successfully designed and implemented a user-friendly chat interface with elements such as text boxes, input fields, and buttons.

4. Custom Communication Protocol:

Developing a custom communication protocol for our chatroom was a critical component of the project. We defined message formats and handled message parsing, demonstrating our ability to create effective and efficient data transfer mechanisms.

5. Error Handling and Exception Management:

The project exposed us to various network and application-related errors, such as connection issues, disconnections, and message format errors. We learned to handle these exceptions gracefully, ensuring a robust and reliable chatroom.

6. User Experience Design:

Focusing on user experience, we designed the GUI to be intuitive and visually appealing, allowing users to easily send and receive messages across the room chat. We considered aspects such as layout, responsiveness, and ease of use.

7. Real-World Application Development:

This project provides practical experience in creating a real-world application. This requires a combination of technical, project management and problem-solving skills as we face a variety of challenges and complexities.

8. Testing and debugging:

Rigorous testing and debugging are the foundation for this project. We learned how to test applications in different scenarios, identify and resolve issues quickly, and ensure reliability.

9. Project Organization:

Proper project planning, code organization, and documentation were pivotal in delivering a maintainable and extensible application. We gained insights into structuring code, naming conventions, and maintaining a clean codebase.

10. Collaboration and Communication:

If we engaged with others, be it project partners or online communities, it provided a chance to refine our collaboration and communication skills. Effective teamwork and communication are essential attributes in a professional development environment.

6.CONCLUSION

Client-server chat room development uses Java, with special emphasis on not incorporating a database, using Swing for the graphical user interface (GUI), implementing threading concepts and taking advantage of the sockets for network connectivity, is a valuable and educational experience. This project allowed us to gain a deeper understanding of many concepts and technologies in software development, from network programming to user interface design and multithreading.

While making this project we learnt how to implement multithreading for concurrent client connections, socket programming for client-server architectures, enabling efficient network communication and error handling. ensuring server efficiency, we also created Create user-friendly chat GUIs with Swing, including text boxes and input fields and develop a personalized communication protocol to effectively manage messages in chat rooms.

7.REFERENCES

1. https://www.javatpoint.com
2. https://www.javatpoint.com
3. https://www.javatpoint.com
4. https://www.digitalocean.com
5. “Traini, L., Cortellessa, V., Di Pompeo, D. and Tucci, M., 2023. Towards effective assessment of steady state performance in Java software: Are we there yet?. Empirical Software Engineering, 28(1), p.13.”
6. “Bala Dhandayuthapani, V., Implementation of Python Interoperability in Java through TCP Socket Communication.”
7. “Korunović, A. and Vlajić, S., 2023. An Example of Integration of Java GUI Desktop Technologies Using the Abstract Factory Pattern for Education Purposes. ETF Journal of Electrical Engineering, 29(1), pp.3-11.”
8. “Hamidi, B. and Hamidi, L., 2023. Synchronization Possibilities and Features in Java. European Journal of Formal Sciences and Engineering, 6(2), pp.124-136.”
9. “Chen, Bochuan, Xiao Guo, Yuting Chen, Xiaofeng Yu, and Lei Bu. "Constructing exception handling chains for testing Java virtual machine implementations." Journal of Software: Evolution and Process (2023): e2562.”
10. “Huang, Wanhong, and Tomoharu Ugawa. "An Object-Oriented Programming Model for Processing-in-Memory Computing in Java."”