University of Dhaka

Department of Computer Science and Engineering

CSE-3111: Computer Networking Lab

Design of a Chat Application Using Multi-threaded Socket Programming

Group Members:

Jannatul Ferdousi (08)
Anirban Roy Sourav (32)

Submitted To:

1. Mr. Palash Roy, Lecturer, Dept. of CSE, DU
2. Mr. Jargis Ahmed, Lecturer, Dept. of CSE, DU
3. Dr. Ismat Rahman, Associate Professor, Dept. of CSE, DU

Submission Date: April 25, 2025

Contents

1 Introduction

Socket programming is a fundamental concept in computer networks that allows communication between processes running on different computers over a network. It provides an Application Programming Interface (API) that enables programs to create network connections and exchange data over these connections. Sockets act as endpoints for communication, allowing data to be sent from one application to another across a network.

In a socket-based communication model, two processes communicate with each other through sockets established at their respective ends. The communication can be likened to a telephone call where the caller initiates the connection, and once established, both parties can exchange information.

Multi-threaded socket programming extends this concept by allowing multiple communication channels to be handled simultaneously. In a multi-threaded environment, a new thread is created for each client connection, enabling the server to handle multiple clients concurrently without being blocked by any single client's operations. Each thread operates independently, managing its own client connection while sharing resources with other threads.

For chat applications, multi-threaded socket programming is essential for several reasons:

- **Concurrent Communication:** In a chat environment, multiple users need to communicate simultaneously. Multi-threading allows the server to handle messages from all connected clients without delays.

- **Scalability:** As the number of users increases, the server can create additional threads to handle new connections, making the application more scalable.

- **Real-time Interaction:** Chat applications require real-time message exchange. Multi-threading ensures that messages are processed and delivered promptly, providing a smooth user experience.

- **Resource Efficiency:** While one client might be idle, others could be actively sending messages. Multi-threading allows the server to efficiently utilize system resources by focusing on active connections.

By implementing a chat application using multi-threaded socket programming, we can create a robust communication platform that can handle multiple users' interactions efficiently, providing a responsive and reliable chatting experience.

2 Objectives

The primary objectives of designing a chat application using multi-threaded socket programming are:

1. **To develop a robust client-server architecture** that enables real-time communication between multiple users through a network connection.

2. **To implement multi-threading techniques** that allow the server to handle multiple client connections concurrently, ensuring efficient message distribution and system resource utilization.

3. **To create a user-friendly terminal interface** that provides intuitive messaging platform for users.

3 Design Details

The chat application is designed using a client-server architecture with multi-threading capabilities. The following steps outline the implementation process:

3.1 Server Design

1. **Server Initialization:** The server initializes on a specified port and waits for client connections.

2. **Thread Creation:** For each new client connection, the server creates a separate thread (ClientHandler) to manage communication with that client.

3. **Client List Management:** The server maintains a list of all currently connected clients to facilitate message broadcasting.

4. **Server Monitor Thread:** A dedicated server-side monitor thread listens for commands like server shutdown or sending messages to clients.

5. **Message Handling:** The server processes incoming messages from clients and sends appropriate responses.

6. **Disconnection:** When a client disconnects, the server removes them from the connected clients list and cleans up the associated resources.

3.2 Client Design

1. **Server Connection:** The client connects to the server using the server's IP address and designated port number.

2. **Message Listener Thread:** A separate thread is created to continuously listen for and handle incoming messages from the server.

3. **Terminal Interface:** The client provides a terminal interface that allows users to input and send messages to the server.

4. **Disconnection Handling:** The client detects server disconnection or shutdown and exits properly.

3.3 Flow Chart

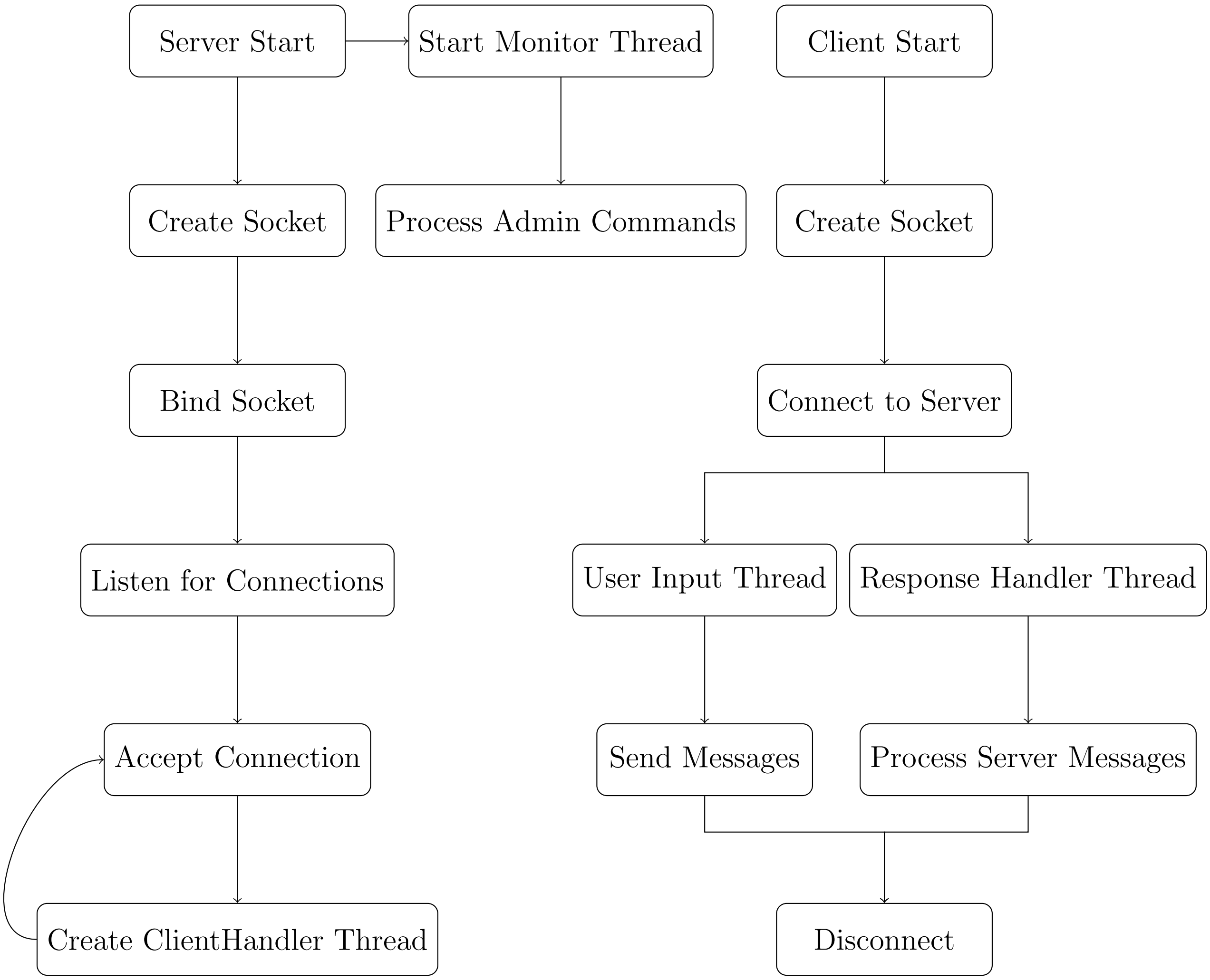


Figure 1: Flow Chart of Multi-threaded Chat Application

4 Implementation

The implementation of the chat application consists of two Java classes: `MultiClientServer.java` for the server and `EnhancedClient.java` for the client.

4.1 Server Implementation

The server implementation includes several key components:

4.1.1 Server Initialization

Listing 1: Server Initialization

```
private static final int PORT = 22222;
private static ServerSocket serverSocket;

public static void main(String[] args) {
    try {
        serverSocket = new ServerSocket(PORT);
        System.out.println("Server Started on port " + PORT);
        // ... rest of the code
    } catch (IOException e) {
        System.out.println("Server error: " + e.getMessage());
        e.printStackTrace();
    } finally {
        shutdown();
    }
}
```

This part initializes the server socket on port 22222. It creates a server socket that listens for client connections.

4.1.2 Client Connection Handling

Listing 2: Client Connection Handling

```
while (serverRunning) {
    try {
        Socket clientSocket = serverSocket.accept();
        System.out.println("New client connected: " +
           clientSocket.getPort());
        System.out.println("Current client count: " +
           (clients.size() + 1));

        ClientHandler clientHandler = new
           ClientHandler(clientSocket);
        clients.add(clientHandler);
        clientHandler.start();
    } catch (IOException e) {
        if (!serverRunning) {
            break;
        }
        e.printStackTrace();
    }
}
```

This code handles new client connections. For each new connection, it creates a new `ClientHandler` thread, adds it to the list of clients, and starts the thread.

4.1.3 Server Monitor Thread

Listing 3: Server Monitor Thread

```
private static void startServerMonitor() {
    Thread monitor = new Thread(() -> {
        Scanner scanner = new Scanner(System.in);
        while (serverRunning) {
            String command = scanner.nextLine();
            if (command.equalsIgnoreCase(TERMINATION_COMMAND) || command.equalsIgnoreCase(EXIT_COMMAND)) {
                System.out.println("Server shutdown initiated...");
                serverRunning = false;
                // ... shutdown code
            } else if (command.equalsIgnoreCase(SEND_COMMAND)) {
                handleSendCommand(scanner);
            }
        }
        scanner.close();
    });
    monitor.setDaemon(true);
    monitor.start();
}
```

This thread keeps monitoring for server commands. It allows the server to shut down or send messages to the clients.

4.1.4 Sending Messages to Clients

Listing 4: Server Message Sending

```
private static void handleSendCommand(Scanner scanner) {
    // ... client selection logic

    System.out.print("Enter message to send: ");
    String message = scanner.nextLine();
    if (message.trim().isEmpty()) {
        System.out.println("Message cannot be empty. Cancelling send operation.");
        return;
    }

    String serverMessage = "[SERVER MESSAGE] " + message;

    if (clientNumber == 0) {
        // Send to all clients
        System.out.println("Sending message to all clients...");
        int successCount = 0;
        for (ClientHandler client : clients) {
            if (client.sendMessage(serverMessage)) {
                successCount++;
            }
        }
        System.out.println("Message sent to " + successCount + " out of " + clients.size() + " clients.");
    } else {
        // Send to specific client
        ClientHandler targetClient = clients.get(clientNumber - 1);
        System.out.println("Sending message to Client " + targetClient.clientAddress + "...");
        if (targetClient.sendMessage(serverMessage)) {
            System.out.println("Message sent successfully.");
        } else {
```

```java
            System.out.println("Failed to send message to client.");
        }
    }
}
```

This method handles sending messages from the server to clients. It allows sending to either all clients or a specific client.

4.1.5 ClientHandler Class

Listing 5: ClientHandler Class

```java
static class ClientHandler extends Thread {
    private Socket socket;
    private DataOutputStream out;
    private DataInputStream in;
    private boolean isRunning = true;
    private String clientAddress;

    // ... constructor and other methods

    @Override
    public void run() {
        try {
            out.writeUTF("Welcome to the chat server! Type ’EXIT’ to disconnect.");

            while (isRunning && serverRunning) {
                try {
                    String message = in.readUTF();

                    if (message.equalsIgnoreCase("EXIT")) {
                        out.writeUTF("Goodbye! Disconnecting your session.");
                        break;
                    }

                    System.out.println("From client " + clientAddress + ": " + message);

                    String response = processMessage(message);

                    out.writeUTF(response);
                } catch (IOException e) {
                    // ... error handling
                    break;
                }
            }
        } catch (IOException e) {
            // ... error handling
        } finally {
            closeConnection(null);
            removeClient(this);
        }
    }

    // ... other methods
}
```

The `ClientHandler` class manages individual client connections. It reads messages from clients, processes them, and sends responses.

4.1.6 Message Processing

Listing 6: Message Processing

```
private String processMessage(String message) {
    String[] sentences = message.split("(?<=[.!?])\\s*");
    StringBuilder responseBuilder = new StringBuilder();

    for (String sentence : sentences) {
        sentence = sentence.trim();
        if (!sentence.isEmpty()) {
            String processedSentence = sentence.toLowerCase();

            String timestamp = new
                java.text.SimpleDateFormat("HH:mm:ss").format(new
                java.util.Date());
            responseBuilder.append("[").append(timestamp).append("]
                ")
                              .append("Processed:
                                 \"").append(processedSentence)
                              .append("\"\n");
        }
    }

    return responseBuilder.toString().trim();
}
```

This method processes client messages by splitting them into sentences, converting each sentence to lowercase, and adding a timestamp.

4.2 Client Implementation

The client implementation includes several key components:

4.2.1 Client Initialization

Listing 7: Client Initialization

```
private static final String SERVER_IP = "localhost";
private static final int SERVER_PORT = 22222;
private static final String EXIT_COMMAND = "EXIT";
private static AtomicBoolean clientRunning = new
    AtomicBoolean(true);

public static void main(String[] args) {
    Socket socket = null;
    DataOutputStream out = null;
    DataInputStream in = null;

    try {
        System.out.println("Client starting...");

        socket = new Socket(SERVER_IP, SERVER_PORT);
        System.out.println("Connected to server at " + SERVER_IP + ":" + SERVER_PORT);

        out = new DataOutputStream(socket.getOutputStream());
        in = new DataInputStream(socket.getInputStream());

        // ... rest of the code
    } catch (IOException e) {
        System.out.println("Client error: " + e.getMessage());
    } finally {
        // ... cleanup code
    }
}
```

This code initializes the client and establishes a connection to the server.

4.2.2 Server Response Handler Thread

Listing 8: Response Handler Thread

```
Thread responseHandler = new Thread(() -> {
    try {
        while (clientRunning.get()) {
            try {
                String response = finalIn.readUTF();
                System.out.println("\nServer response: \n" + response);

                if ("SERVER_SHUTDOWN".equals(response)) {
                    System.out.println("Server has shut down. Press Enter to exit.");
                    clientRunning.set(false);
                    break;
                }

                if (clientRunning.get() && !isFirstMessage[0]) {
                    System.out.print("\nEnter message (or '" + EXIT_COMMAND + "' to quit): ");
                }

                isFirstMessage[0] = false;
            } catch (SocketException e) {
                // ... error handling
            }
        }
    } catch (IOException e) {
        // ... error handling
    }
});
```

```
responseHandler.setDaemon(true);
responseHandler.start();
```

This thread continuously listens for messages from the server and displays them to the user.

4.2.3 User Input Handler

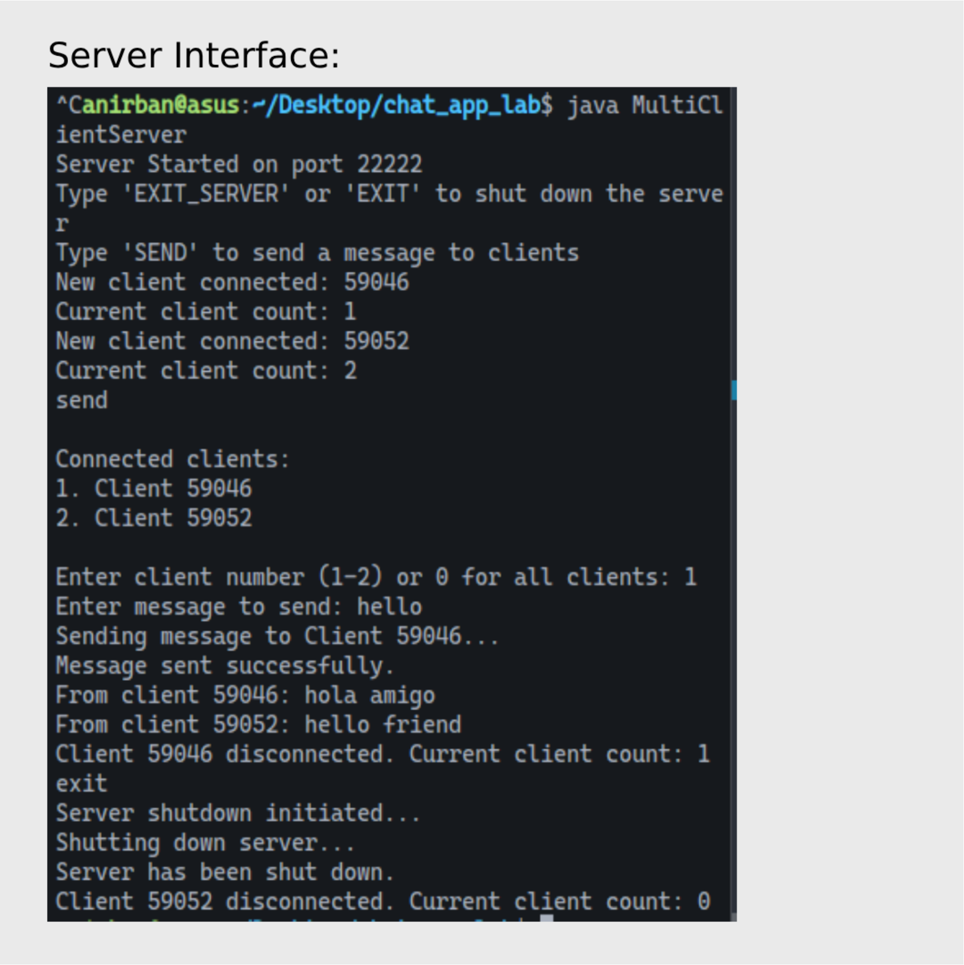
Listing 9: User Input Handler

```
Scanner scanner = new Scanner(System.in);

System.out.print("\nEnter message (or ’" + EXIT_COMMAND + "’ to quit): ");

while (clientRunning.get()) {
    String message = scanner.nextLine();

    if (!clientRunning.get()) {
        break;
    }

    if (message.equalsIgnoreCase(EXIT_COMMAND)) {
        try {
            out.writeUTF(EXIT_COMMAND);
        } catch (IOException e) {
            System.out.println("Could not send exit command, connection already closed.");
        }
        clientRunning.set(false);
        break;
    }

    if (!message.trim().isEmpty()) {
        try {
            out.writeUTF(message);
        } catch (IOException e) {
            // ... error handling
        }
    }
}
```

This part handles user input, sending messages to the server and handling the exit command.

5 Result Analysis

5.0.1 Server Interface

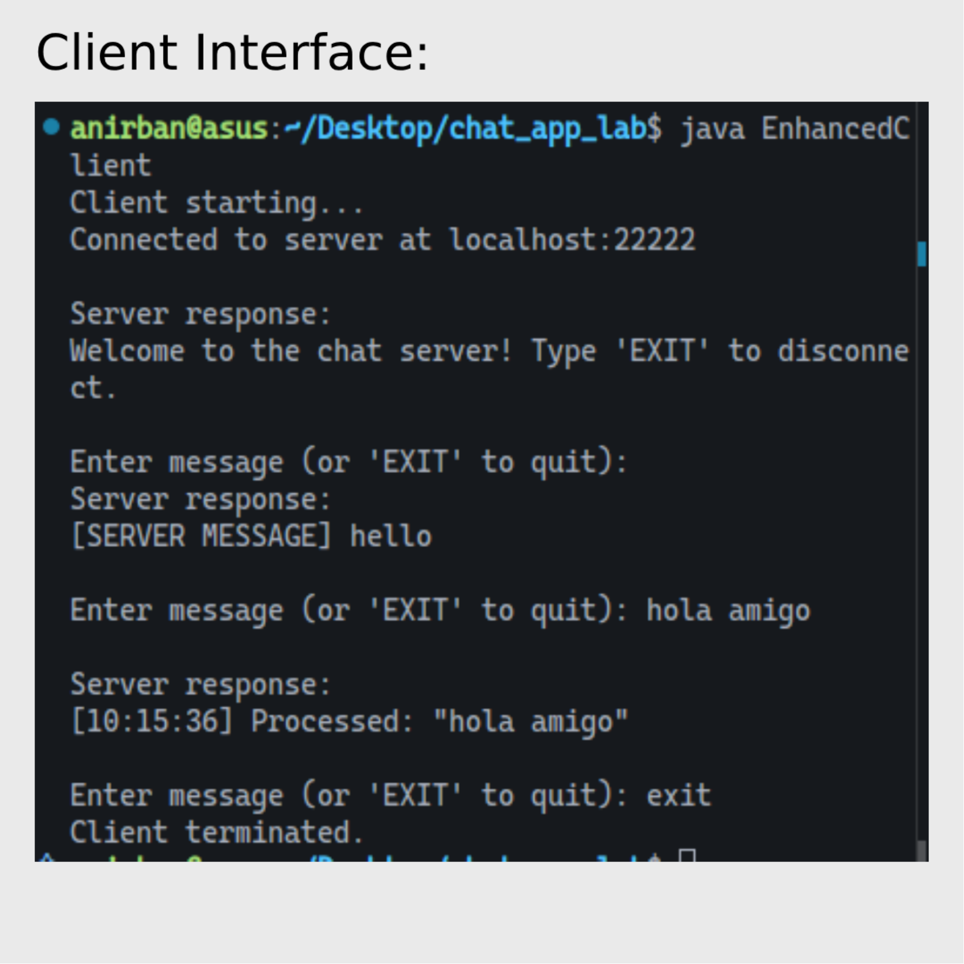


Server Interface:

```
^Canirban@asus:~/Desktop/chat_app_lab$ java MultiCl
ientServer
Server Started on port 22222
Type 'EXIT_SERVER' or 'EXIT' to shut down the serve
r
Type 'SEND' to send a message to clients
New client connected: 59046
Current client count: 1
New client connected: 59052
Current client count: 2
send

Connected clients:
1. Client 59046
2. Client 59052

Enter client number (1-2) or 0 for all clients: 1
Enter message to send: hello
Sending message to Client 59046...
Message sent successfully.
From client 59046: hola amigo
From client 59052: hello friend
Client 59046 disconnected. Current client count: 1
exit
Server shutdown initiated...
Shutting down server...
Server has been shut down.
Client 59052 disconnected. Current client count: 0
```

Figure 2: Server console interface

5.0.2 Client Interface



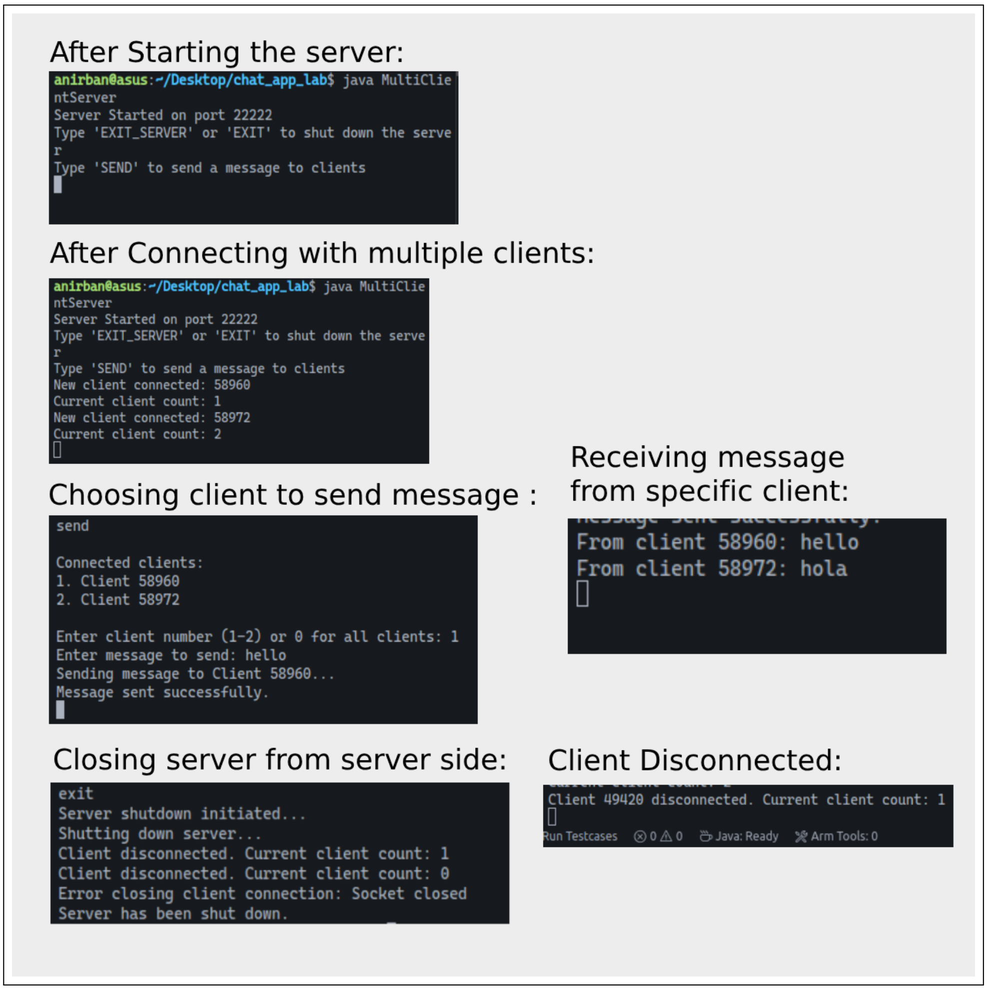
Client Interface:

```
• anirban@asus:~/Desktop/chat_app_lab$ java EnhancedC
  lient
  Client starting...
  Connected to server at localhost:22222

  Server response:
  Welcome to the chat server! Type 'EXIT' to disconne
  ct.

  Enter message (or 'EXIT' to quit):
  Server response:
  [SERVER MESSAGE] hello

  Enter message (or 'EXIT' to quit): hola amigo

  Server response:
  [10:15:36] Processed: "hola amigo"

  Enter message (or 'EXIT' to quit): exit
  Client terminated.
```

Figure 3: Client console interface

5.1 Server Side Output

When the server is launched, it displays initialization messages and starts listening for client connections on port 22222. The server console output provides real-time feedback on client activities:



After Starting the server:

```
anirban@asus:~/Desktop/chat_app_lab$ java MultiClie
ntServer
Server Started on port 22222
Type 'EXIT_SERVER' or 'EXIT' to shut down the serve
r
Type 'SEND' to send a message to clients
```

After Connecting with multiple clients:

```
anirban@asus:~/Desktop/chat_app_lab$ java MultiClie
ntServer
Server Started on port 22222
Type 'EXIT_SERVER' or 'EXIT' to shut down the serve
r
Type 'SEND' to send a message to clients
New client connected: 58960
Current client count: 1
New client connected: 58972
Current client count: 2
```

Choosing client to send message :

```
send

Connected clients:
1. Client 58960
2. Client 58972

Enter client number (1-2) or 0 for all clients: 1
Enter message to send: hello
Sending message to Client 58960...
Message sent successfully.
```

Receiving message from specific client:

```
Message sent successfully.
From client 58960: hello
From client 58972: hola
```

Closing server from server side:

```
exit
Server shutdown initiated...
Shutting down server...
Client disconnected. Current client count: 1
Client disconnected. Current client count: 0
Error closing client connection: Socket closed
Server has been shut down.
```

Client Disconnected:

```
Current client count: 2
Client 49420 disconnected. Current client count: 1
```

Figure 4: Server Console Output

The server console displays the following information:

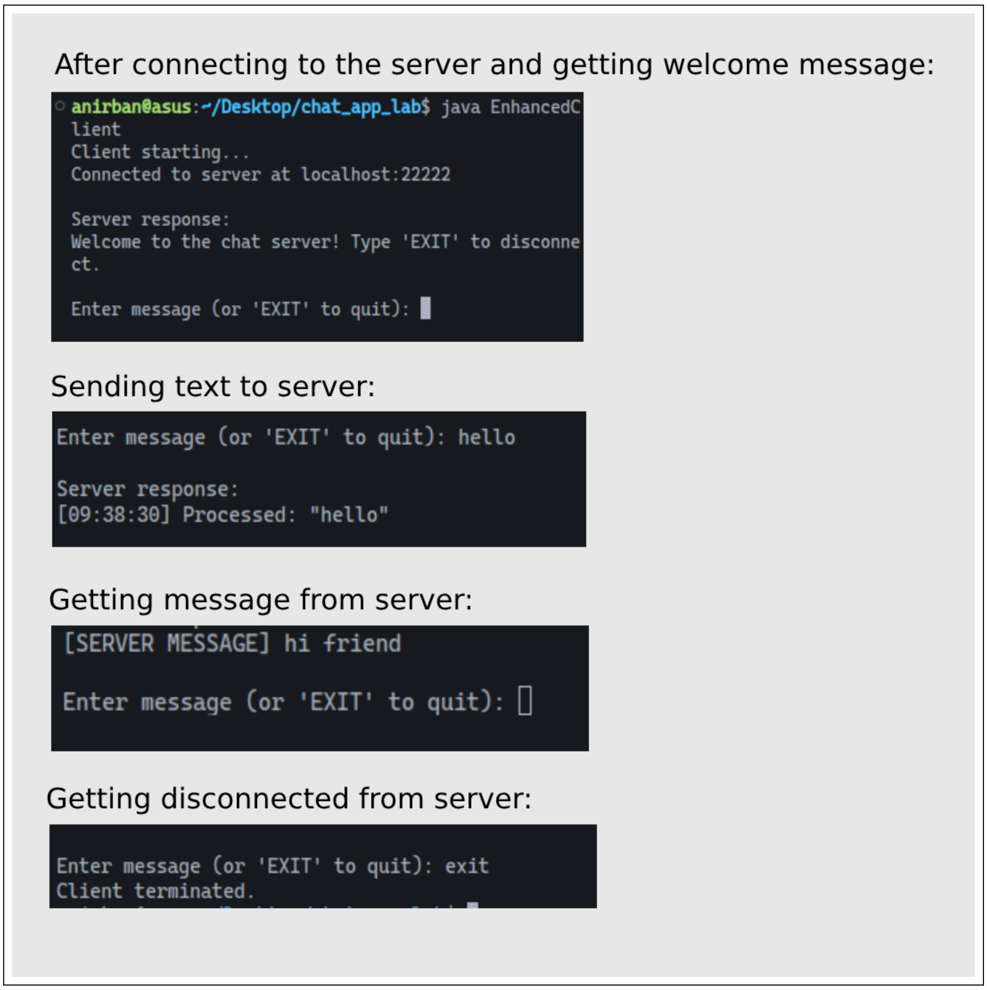
- Server initialization: "Server Started on port 22222"
- Client connection notifications: "New client connected: [PORT]" with incremental client count
- Message logging: "From client [ADDRESS]: [MESSAGE]"
- Server commands processing: Response to administrator commands like shutdown or message sending
- Client disconnection tracking: Removal of clients from active list and resource cleanup

The server also includes administrator functionality through its monitor thread, allowing the server operator to:

- Send messages to all clients or a specific client
- Initiate server shutdown with the commands "exit" or "shutdown"

5.2 Client Side Output

The client interface operates in console mode, providing clear communication with the server:



After connecting to the server and getting welcome message:

```
○ anirban@asus:~/Desktop/chat_app_lab$ java EnhancedC
lient
Client starting...
Connected to server at localhost:22222

Server response:
Welcome to the chat server! Type 'EXIT' to disconne
ct.

Enter message (or 'EXIT' to quit): █
```

Sending text to server:

```
Enter message (or 'EXIT' to quit): hello

Server response:
[09:38:30] Processed: "hello"
```

Getting message from server:

```
[SERVER MESSAGE] hi friend

Enter message (or 'EXIT' to quit): ▯
```

Getting disconnected from server:

```
Enter message (or 'EXIT' to quit): exit
Client terminated.
```

Figure 5: Client Interface Example

The client console displays:

- Connection status: "Connected to server at [SERVER_IP]:[SERVER_PORT]"
- Welcome message from server: "Welcome to the chat server! Type 'EXIT' to disconnect."
- Server responses with timestamps: "[HH:mm:ss] Processed: [message in lowercase]"
- Input prompt: "Enter message (or 'EXIT' to quit):"
- Server notifications: Including server shutdown messages

The client implementation effectively separates user input handling and server response processing into two different threads:

- The main thread handles user input and sending messages to the server
- A daemon thread continuously listens for and displays server responses

This dual-threaded approach ensures that users can send messages at any time while still receiving incoming server responses without disruption.

6 Discussion

6.1 Comparison between Basic and Multi-threaded Socket Programming

Basic Socket Programming	**Multi-threaded Socket Programming**
Handles one client at a time	Handles multiple clients simultaneously
Blocks while processing each client request	Processes client requests concurrently
Simple implementation with less overhead	More complex implementation with thread management
Suitable for applications with one client	Ideal for applications with many clients
Poor scalability	Good scalability
Poor real-time response	Good real-time response

6.2 Drawbacks of Basic Socket Programming

Basic socket programming has several limitations:

1. **Sequential Processing**: It handles messages sequentially. It means client cannot send message until the server sends the response for the previous message.

2. **Blocking Nature**: The server or client blocks while the other end is sending something.

3. **Limited Scalability**: It cannot handle multiple clients.

4. **Poor Real-time Response**: Real-time applications like chat suffer from delays as the client has to wait for server response and vice varsa.

6.3 Overcoming Drawbacks with Multi-threaded Programming

The implemented multi-threaded chat application addresses these limitations by:

1. **Handling Stuff at Once**: Each client gets its own thread, so everyone's requests get handled at the same time.

2. **Server Keeps Working**: The main server part keeps accepting new clients while other parts handle existing ones.

3. **Handles Lots of Users**: The server can deal with tons of clients.

4. **Quick Responses**: Nobody waits in line. Everyone gets answers simultaneously.

5. **Server Can Start Chats**: The server can send messages to a specific client. Even it can send same message to all the clients at once.

6.4 Learning Outcomes

Through this project, we have learned:

1. How to implement multi-threaded socket programming in Java
2. Techniques for managing multiple client connections
3. Methods for handling concurrent data communication
4. Strategies for error handling and connection termination
5. Approaches for implementing server and client-side monitoring
6. Understanding shared resource management

6.5 Challenges Faced

During the implementation, we encountered several challenges:

1. **Connection Termination**: Properly handling client disconnections and server shutdown.
2. **Error Handling**: Managing error handling mechanisms for network and I/O exceptions.
3. **User Interface**: Creating an intuitive terminal interface for both server and client.
4. **Message Processing**: Implementing efficient message processing and delivery with proper formatting.

7 Conclusion

The multi-threaded chat application successfully demonstrates the advantages of multi-threaded socket programming over basic socket programming. By handling multiple clients simultaneously, the application provides a responsive and scalable solution for real-time communication.

The server component efficiently manages client connections, processes messages, and enables server-initiated communication. The client component provides a simple terminal interface for sending messages to the server and displaying responses.

This project has shown insights into network programming concepts, multi-threading. The skills and knowledge gained from this implementation can be applied to more complex networked applications in the future.