Development of a Multi-Client Chat System Using Threads

Submitted to : Polash Roy
Team Members: Sara Faria Sundra (58)
Asuma Bhuiyan (85)
Course: Networking Lab 3
Department of Computer Science and Engineering, University of Dhaka

Submission Date: 18th September 2025

1 Overview

Network programming allows programs to communicate over a network by creating endpoints called sockets. These sockets are used to send and receive data between two nodes, commonly known as a client and a server.

Implementing threads in a server program permits handling multiple clients at once. Each client operates independently in its own thread, which prevents blocking and improves responsiveness.

This technique is crucial in chat applications, where multiple users need to exchange messages concurrently without delays.

2 Lab Goals

1. Learn the basics of network socket programming in Java.
2. Implement a threaded server that can serve several clients at the same time.
3. Build a working chat program demonstrating bidirectional communication.

3 System Design

1. Initialize a server program to listen on a defined port for incoming connections.
2. Allocate a dedicated thread for every connected client to handle its messages.
3. Maintain a thread-safe collection of all active clients for message delivery.
4. Develop a client application capable of sending messages to and receiving messages from the server.
5. Use a separate listener thread on the client side to continuously receive messages from the server.

4 Source Code

4.1 Server Implementation

```java
import java.io.*;
import java.net.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.Map;

public class MessageServer {

    private static Map<Integer, PrintWriter> connectedClients = new ConcurrentHashMap<>();

    public static void main(String[] args) throws IOException {
        int listenPort = 12345;
        System.out.println("Server started on port " + listenPort);
        System.out.println("Enter 'exit' to terminate the server.");

        new Thread(new ConsoleHandler()).start();

        ServerSocket serverSock = new ServerSocket(listenPort);
        while (true) {
            Socket clientSock = serverSock.accept();
            new Thread(new ClientProcessor(clientSock)).start();
        }
    }

    private static class ClientProcessor implements Runnable {
        private Socket socket;
        private int portNumber;
        private PrintWriter writer;

        public ClientProcessor(Socket sock) {
            this.socket = sock;
            this.portNumber = sock.getPort();
        }

        public void run() {
            try {
                BufferedReader reader = new BufferedReader(
                        new InputStreamReader(socket.getInputStream())
                );
                this.writer = new PrintWriter(socket.getOutputStream(), true);

                connectedClients.put(portNumber, writer);
                System.out.println("Connected: " + portNumber);
                writer.println("Connected successfully. Your port: " + portNumber);

                String line;
                while ((line = reader.readLine()) != null) {
                    if (line.equalsIgnoreCase("exit")) break;
                    System.out.println("Client " + portNumber + ": " + line);
                    writer.println("Server ack: " + line);
                }

            } catch (IOException e) {
                System.out.println("Error with client " + portNumber);
            } finally {
                connectedClients.remove(portNumber);
                System.out.println("Disconnected: " + portNumber);
                try { socket.close(); } catch (IOException e) {}
            }
        }
    }

    private static class ConsoleHandler implements Runnable {
        public void run() {
            try (BufferedReader input = new BufferedReader(new InputStreamReader(System.in))) {
                String command;
                while ((command = input.readLine()) != null) {
                    if (command.equalsIgnoreCase("exit")) {
                        System.out.println("Shutting down server...");
                        for (PrintWriter w : connectedClients.values()) {
                            w.println("Server is stopping.");
                        }
                        System.exit(0);
                    }

                    String[] tokens = command.split(":", 2);
                    if (tokens.length == 2) {
                        try {
                            int target = Integer.parseInt(tokens[0].trim());
                            String msg = tokens[1].trim();
                            PrintWriter client = connectedClients.get(target);
                            if (client != null) {
                                client.println("Server message: " + msg);
                                System.out.println("Sent to " + target);
                            } else {
                                System.out.println("Client not found: " + target);
                            }
                        } catch (NumberFormatException e) {
                            System.out.println("Invalid port format.");
                        }
                    } else {
                        System.out.println("Format: port: message");
                    }
                }
            } catch (IOException e) {
                System.out.println("Console read failed.");
            }
        }
    }
}
```

Listing 1: MessageServer.java

4.2 Client Implementation

```java
import java.io.*;
import java.net.*;

public class MessageClient {

    public static void main(String[] args) {
        String serverHost = "127.0.0.1";
        int serverPort = 12345;

        try {
            Socket conn = new Socket(serverHost, serverPort);
            System.out.println("Connected to server.");
            System.out.println("Type 'exit' to leave.");

            new Thread(new ServerReceiver(conn)).start();

            PrintWriter out = new PrintWriter(conn.getOutputStream(), true);
            BufferedReader userInput = new BufferedReader(new InputStreamReader(System.in));

            String msg;
            while (true) {
                msg = userInput.readLine();
                out.println(msg);
                if (msg.equalsIgnoreCase("exit")) break;
            }

            conn.close();

        } catch (IOException e) {
            System.out.println("Cannot connect: " + e.getMessage());
        }

        System.out.println("Disconnected from server.");
    }

    private static class ServerReceiver implements Runnable {
        private Socket sock;

        public ServerReceiver(Socket socket) {
            this.sock = socket;
        }

        public void run() {
            try {
                BufferedReader in = new BufferedReader(
                        new InputStreamReader(sock.getInputStream())
                );
                String serverMsg;
                while ((serverMsg = in.readLine()) != null) {
                    System.out.println("[Server] " + serverMsg);
                }
            } catch (IOException e) {
                System.out.println("Lost connection to server.");
            }
        }
    }
}
```

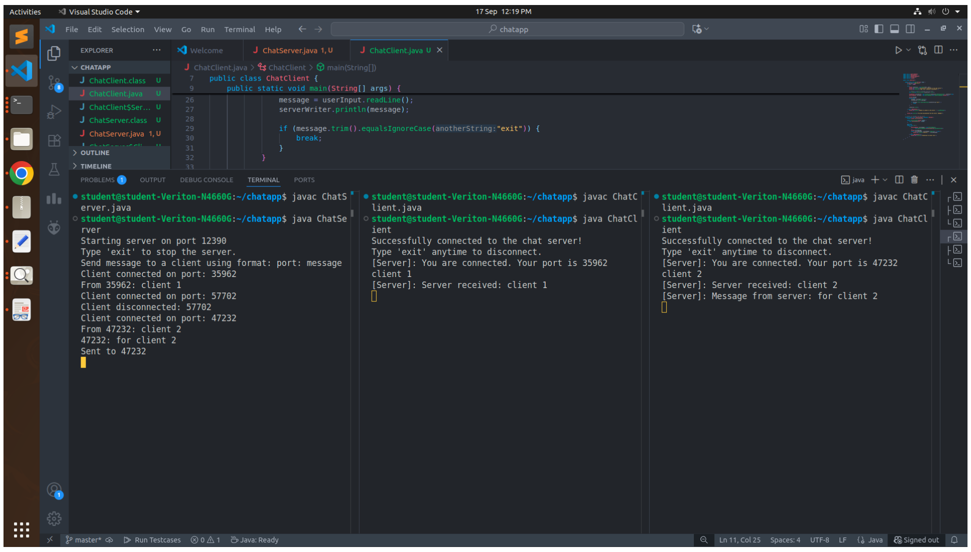
Listing 2: MessageClient.java

5 Observations

- Server displays client connections and messages.
- Clients send messages and receive responses without blocking.

6 Conclusion

Traditional socket programs allow only one client at a time, causing delays. Threaded servers enable concurrent communication, improve efficiency, and allow multiple clients



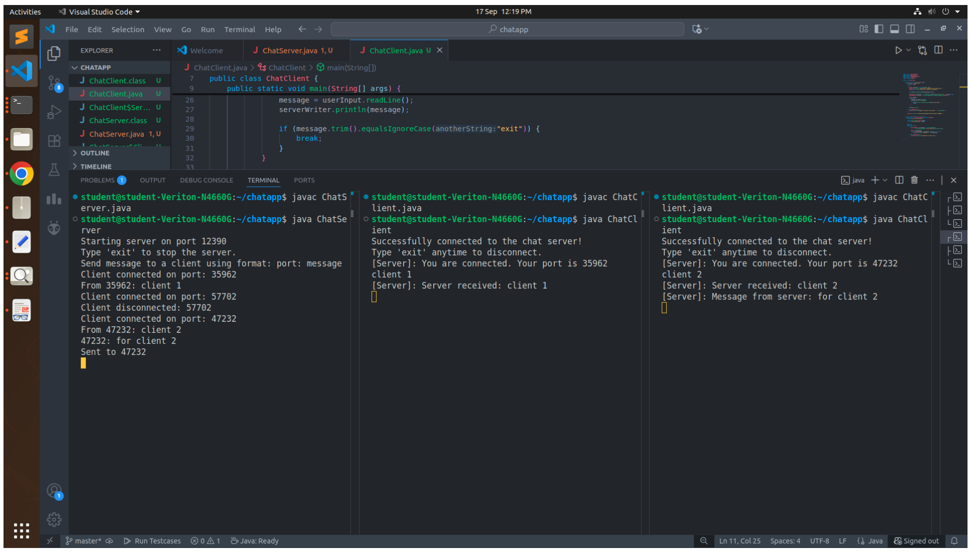
```java
public class ChatClient {
    public static void main(String[] args) {
                message = userInput.readLine();
                serverWriter.println(message);

                if (message.trim().equalsIgnoreCase(anotherString:"exit")) {
                    break;
                }
            }
```

```
student@student-Veriton-N4660G:~/chatapp$ javac ChatS
erver.java
student@student-Veriton-N4660G:~/chatapp$ java ChatSe
rver
Starting server on port 12390
Type 'exit' to stop the server.
Send message to a client using format: port: message
Client connected on port: 35962
From 35962: client 1
Client connected on port: 57702
Client disconnected: 57702
Client connected on port: 47232
From 47232: client 2
47232: for client 2
Sent to 47232
```

```
student@student-Veriton-N4660G:~/chatapp$ javac ChatC
lient.java
student@student-Veriton-N4660G:~/chatapp$ java ChatCl
ient
Successfully connected to the chat server!
Type 'exit' anytime to disconnect.
[Server]: You are connected. Your port is 35962
client 1
[Server]: Server received: client 1
```

```
student@student-Veriton-N4660G:~/chatapp$ javac ChatC
lient.java
student@student-Veriton-N4660G:~/chatapp$ java ChatCl
ient
Successfully connected to the chat server!
Type 'exit' anytime to disconnect.
[Server]: You are connected. Your port is 47232
client 2
[Server]: Server received: client 2
[Server]: Message from server: for client 2
```

Figure 1: Active clients and server logs



```java
public class ChatClient {
    public static void main(String[] args) {
                message = userInput.readLine();
                serverWriter.println(message);

                if (message.trim().equalsIgnoreCase(anotherString:"exit")) {
                    break;
                }
            }
```

```
student@student-Veriton-N4660G:~/chatapp$ javac ChatS
erver.java
student@student-Veriton-N4660G:~/chatapp$ java ChatSe
rver
Starting server on port 12390
Type 'exit' to stop the server.
Send message to a client using format: port: message
Client connected on port: 35962
From 35962: client 1
Client connected on port: 57702
Client disconnected: 57702
Client connected on port: 47232
From 47232: client 2
47232: for client 2
Sent to 47232
```

```
student@student-Veriton-N4660G:~/chatapp$ javac ChatC
lient.java
student@student-Veriton-N4660G:~/chatapp$ java ChatCl
ient
Successfully connected to the chat server!
Type 'exit' anytime to disconnect.
[Server]: You are connected. Your port is 35962
client 1
[Server]: Server received: client 1
```

```
student@student-Veriton-N4660G:~/chatapp$ javac ChatC
lient.java
student@student-Veriton-N4660G:~/chatapp$ java ChatCl
ient
Successfully connected to the chat server!
Type 'exit' anytime to disconnect.
[Server]: You are connected. Your port is 47232
client 2
[Server]: Server received: client 2
[Server]: Message from server: for client 2
```

Figure 2: Client sending and receiving messages

to interact seamlessly.

Key Takeaways:

- Learn network socket operations in Java.
- Manage multi-threaded servers with shared data safely.
- Achieve real-time client-server message exchange.

Challenges:

- Safely accessing shared resources from multiple threads.
- Closing client sockets correctly.
- Handling unexpected disconnections smoothly.