



University of Dhaka

Department of Computer Science & Engineering

CSE-3111: Computer Networking Lab

Year: 3rd Semester: 1st

Lab Report No: 01

Lab Report Name: Design of a Chat application using multi-threaded socket programming

Submission date: 18.09.2025

Submitted by:

Srabon Aich (Roll-15)
Abantika Paul (Roll-21)

Submitted to:

Dr. Shabbir Ahmed, Professor, Dept. of CSE, DU
Mr. Palash Roy, Lecturer, Dept. of CSE, DU

Contents

1	Introduction	2
2	Objectives	2
3	Design Details	2
4	Implementation	3
4.1	Server Side	4
4.2	Client Side	7
5	Result Analysis	8
5.1	Server Output	9
5.2	Client Outputs	9
5.3	Description of Outputs	9
6	Discussion	10

1 Introduction

Socket programming is a way to enable communication between two machines over a network. It provides an interface for sending and receiving data between a server and clients. Sockets can be used for both connection-oriented communication (TCP) and connectionless communication (UDP).

Multi-threaded socket programming allows a server to handle multiple clients simultaneously by creating a separate thread for each client connection. This approach is essential for designing a chat application where multiple clients can send and receive messages independently without waiting for other clients.

The necessity of multi-threaded socket programming in a chat application includes:

- Enabling simultaneous communication with multiple clients.
- Preventing the server from blocking when one client is sending or receiving messages.
- Allowing the server to reply to clients individually at different times.

2 Objectives

The main objectives of this lab are:

1. To understand the concepts of socket programming and multi-threaded server design.
2. To implement a chat application where multiple clients can communicate with a single server.
3. To provide asynchronous communication, allowing the server to respond to clients individually.

3 Design Details

The implementation of the multi-threaded chat application using socket programming was completed in the following steps:

1. **Server Setup:** The server was implemented using a `ServerSocket` bound to port 12345. It continuously listens for new client connections.
2. **Client Connection:** When a new client connects, the server assigns a unique client ID and creates a new `ClientHandler` thread to manage communication with that client.
3. **Thread Pool Management:** An `ExecutorService` (cached thread pool) was used to manage multiple client handler threads efficiently.
4. **Client–Server Communication:** Each client can send messages to the server. The server receives these messages, tags them with the client ID, and displays them asynchronously.

5. **Server-to-Client Messaging:** The server console allows sending a message directly to a specific client using the format: <clientId> <message> This makes the communication bi-directional.
6. **Graceful Termination:** If a client types `bye`, the connection is closed from that client's side. If the server types `exit`, it shuts down all client connections and terminates safely.
7. **Message Handling Queue:** A blocking queue (`LinkedBlockingQueue`) was used to handle client messages asynchronously so that server output remains smooth without blocking other tasks.

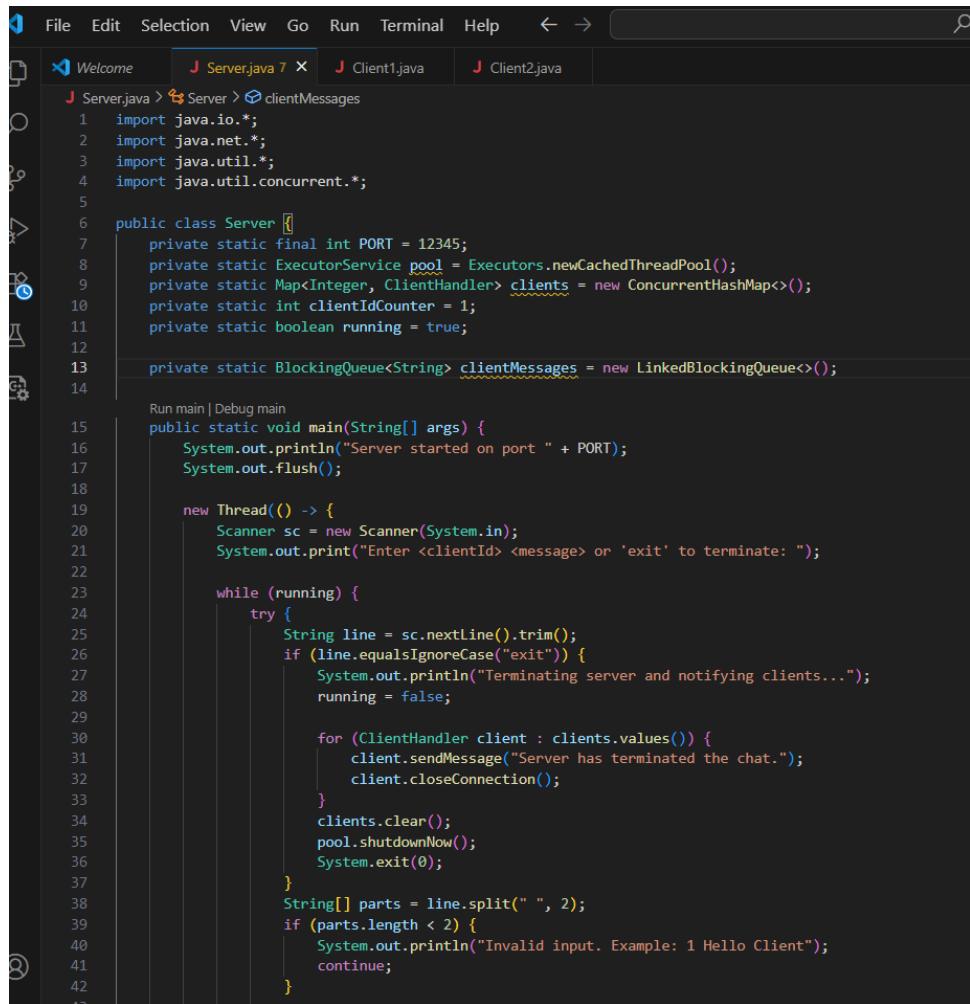
A simple flow of execution is illustrated below:

- Client connects → Server assigns ID → New thread handles communication.
- Client sends message → Server reads message → Message displayed in server console.
- Server types message → Routed to specific client.

4 Implementation

The program was implemented using Java. Screenshots of the server and client programs are shown below.

4.1 Server Side

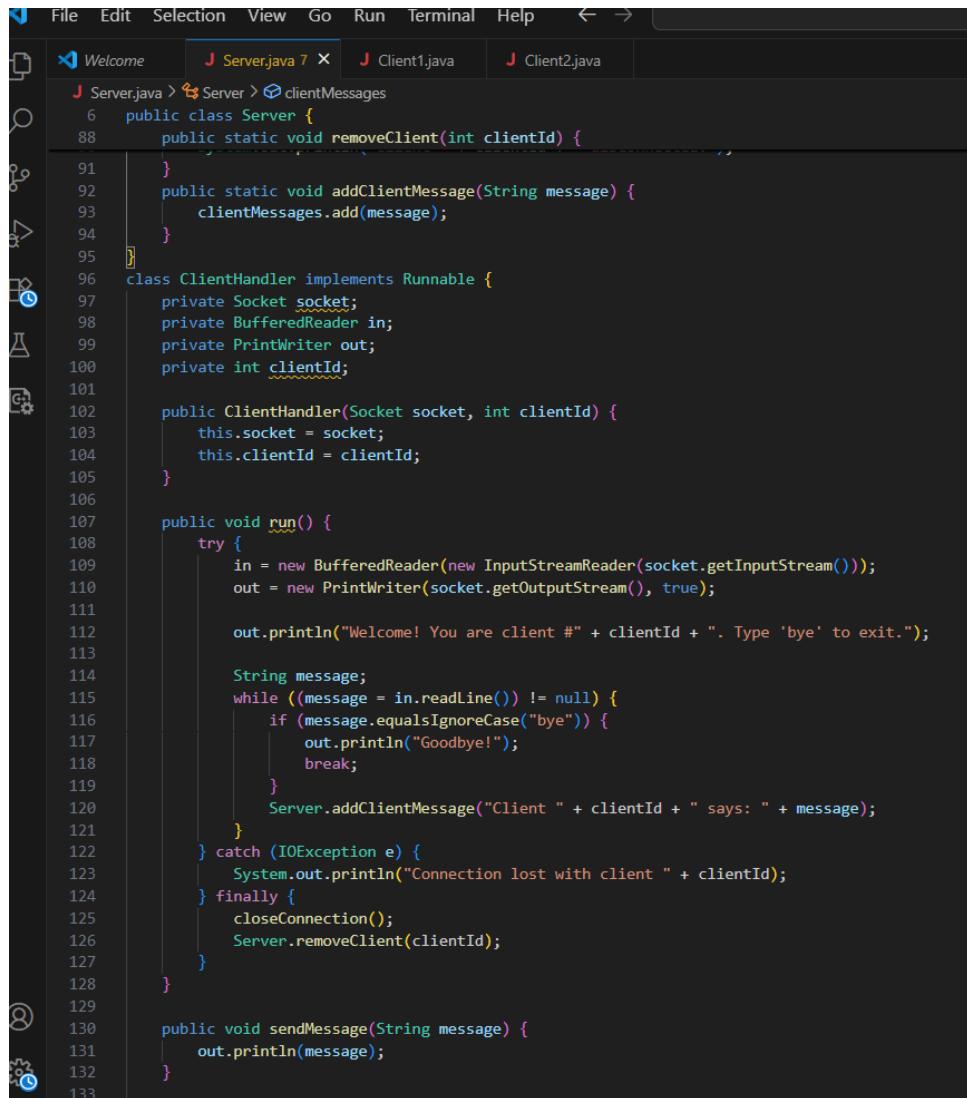


The screenshot shows a Java code editor interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Tab Bar:** Welcome, Server.java (highlighted), Client1.java, Client2.java.
- Code Area:** The Server.java file is displayed with the following content:

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.*;
4 import java.util.concurrent.*;
5
6 public class Server {
7     private static final int PORT = 12345;
8     private static ExecutorService pool = Executors.newCachedThreadPool();
9     private static Map<Integer, ClientHandler> clients = new ConcurrentHashMap<>();
10    private static int clientIdCounter = 1;
11    private static boolean running = true;
12
13    private static BlockingQueue<String> clientMessages = new LinkedBlockingQueue<>();
14
15    public static void main(String[] args) {
16        System.out.println("Server started on port " + PORT);
17        System.out.flush();
18
19        new Thread(() -> {
20            Scanner sc = new Scanner(System.in);
21            System.out.print("Enter <clientId> <message> or 'exit' to terminate: ");
22
23            while (running) {
24                try {
25                    String line = sc.nextLine().trim();
26                    if (line.equalsIgnoreCase("exit")) {
27                        System.out.println("Terminating server and notifying clients...");
28                        running = false;
29
30                        for (ClientHandler client : clients.values()) {
31                            client.sendMessage("Server has terminated the chat.");
32                            client.closeConnection();
33                        }
34
35                        clients.clear();
36                        pool.shutdownNow();
37                        System.exit(0);
38                    }
39
40                    String[] parts = line.split(" ", 2);
41                    if (parts.length < 2) {
42                        System.out.println("Invalid input. Example: 1 Hello Client");
43                        continue;
44                    }
45
46                }
47            }
48        });
49    }
50}
```

```
File Edit Selection View Go Run Terminal Help ← → ⌘ Networking
Welcome J Server.java 7 × J Client1.java J Client2.java
J Server.java > Server > clientMessages
6  public class Server {
15    public static void main(String[] args) {
43
44        int clientId;
45        try {
46            clientId = Integer.parseInt(parts[0]);
47        } catch (NumberFormatException e) {
48            System.out.println("Invalid client ID. Example: 1 Hello Client");
49            continue;
50        }
51
52        String msg = parts[1];
53        ClientHandler client = clients.get(clientId);
54        if (client != null) {
55            client.sendMessage("Server: " + msg);
56        } else {
57            System.out.println("Client " + clientId + " not found.");
58        }
59
60        } catch (Exception e) {
61            System.out.println("Error reading input. Try again.");
62        }
63    }
64 }.start();
65 new Thread(() -> {
66     while (running) {
67         try {
68             String msg = clientMessages.take();
69             System.out.println(msg);
70         } catch (InterruptedException e) {
71             break;
72         }
73     }
74 }).start();
75 try (ServerSocket serverSocket = new ServerSocket(PORT)) {
76     while (running) {
77         Socket clientSocket = serverSocket.accept();
78         int clientId = clientIdCounter++;
79         ClientHandler handler = new ClientHandler(clientSocket, clientId);
80         clients.put(clientId, handler);
81         pool.execute(handler);
82         System.out.println("New client connected: Client " + clientId + " - " + clientSocket);
83     }
84 } catch (IOException e) {
85     if (running) e.printStackTrace();
}
```



The screenshot shows a Java code editor interface with the following details:

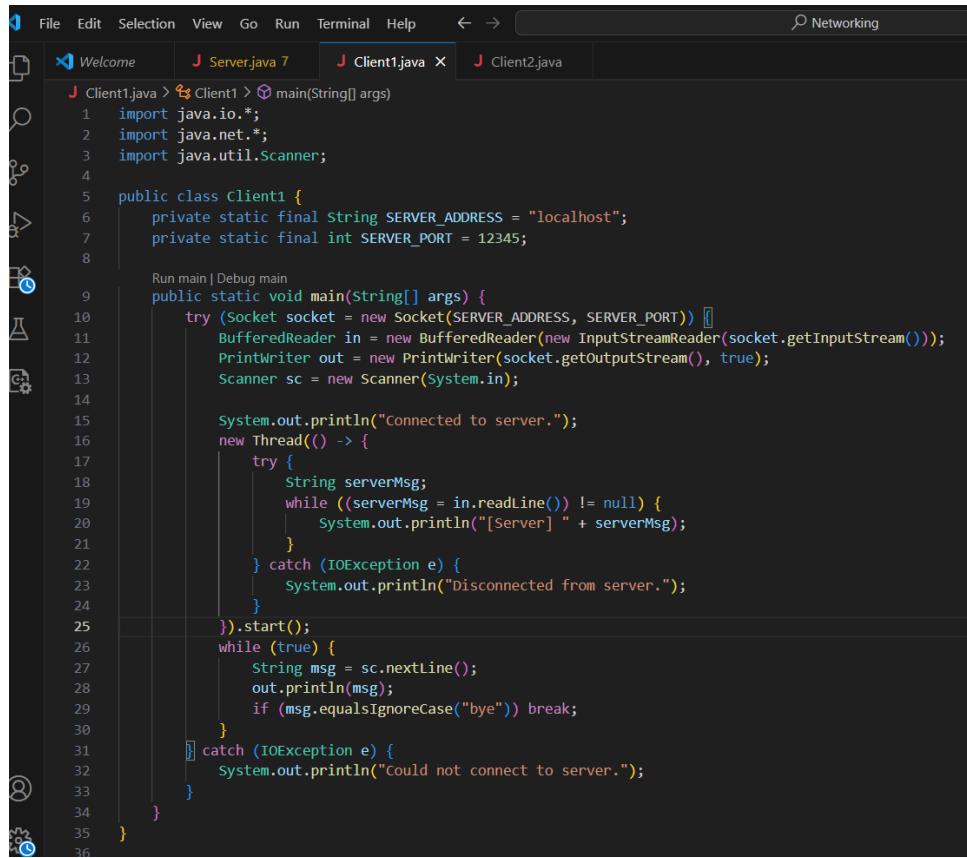
- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Tab Bar:** Welcome, Server.java 7 X, Client1.java, Client2.java.
- Code Area:** The code is for a `Server` class. It includes methods for removing clients, adding client messages, and handling client connections. It uses `BufferedReader` and `PrintWriter` for socket communication, and `ArrayList` for client messages. It handles `IOException` and provides a `sendMessage` method for administrator input.

```
6  public class Server {
88     public static void removeClient(int clientId) {
91     }
92     public static void addClientMessage(String message) {
93         clientMessages.add(message);
94     }
95 }
96 class ClientHandler implements Runnable {
97     private Socket socket;
98     private BufferedReader in;
99     private PrintWriter out;
100    private int clientId;
101
102    public ClientHandler(Socket socket, int clientId) {
103        this.socket = socket;
104        this.clientId = clientId;
105    }
106
107    public void run() {
108        try {
109            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
110            out = new PrintWriter(socket.getOutputStream(), true);
111
112            out.println("Welcome! You are client #" + clientId + ". Type 'bye' to exit.");
113
114            String message;
115            while ((message = in.readLine()) != null) {
116                if (message.equalsIgnoreCase("bye")) {
117                    out.println("Goodbye!");
118                    break;
119                }
120                Server.addClientMessage("Client " + clientId + " says: " + message);
121            }
122        } catch (IOException e) {
123            System.out.println("Connection lost with client " + clientId);
124        } finally {
125            closeConnection();
126            Server.removeClient(clientId);
127        }
128    }
129
130    public void sendMessage(String message) {
131        out.println(message);
132    }
133 }
```

Figure 1: Implementation of the server-side code, showing multiple client connections, message handling, administrator input, and termination of client connections.

4.2 Client Side

Client1

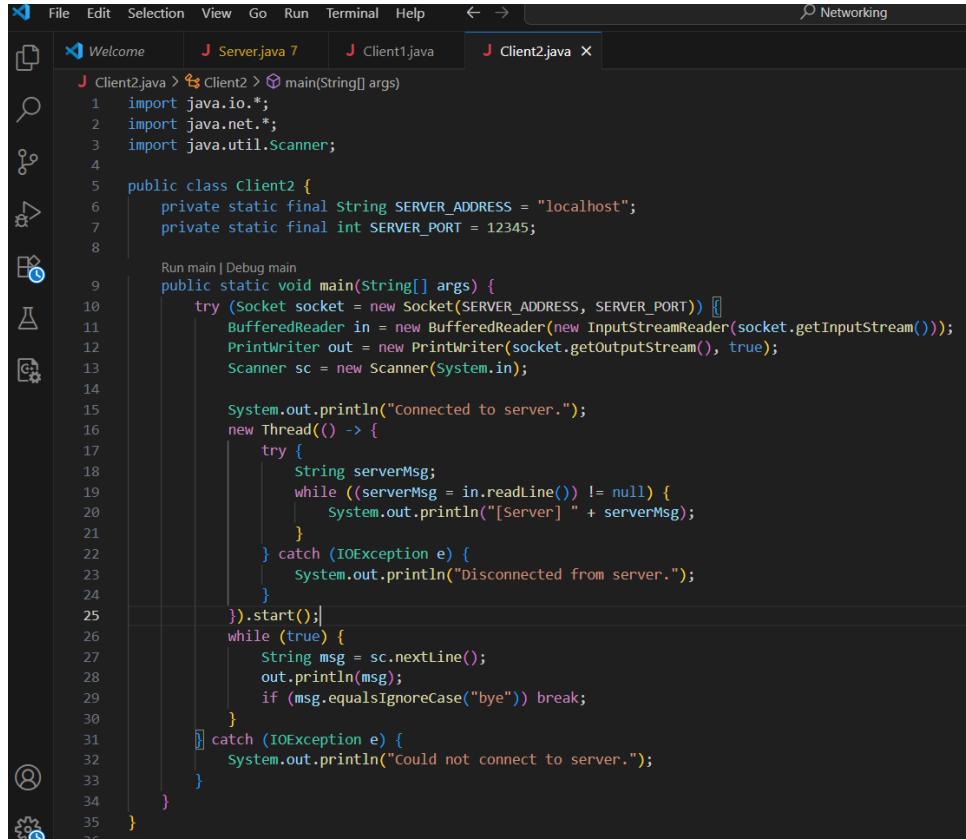


The screenshot shows a Java IDE interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Toolbar:** Welcome, Client1 > Client1 > main(String[] args), Client1.java (selected), Client2.java.
- Code Area:** The code for Client1.java is displayed. It imports java.io.* and java.net.*. It defines a static final SERVER_ADDRESS as "localhost" and SERVER_PORT as 12345. The main method creates a socket to the server, reads messages from the server, and prints them to the console. It also handles user input and disconnects if the user types "bye".

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.Scanner;
4
5 public class Client1 {
6     private static final String SERVER_ADDRESS = "localhost";
7     private static final int SERVER_PORT = 12345;
8
9     public static void main(String[] args) {
10        try (Socket socket = new Socket(SERVER_ADDRESS, SERVER_PORT)) {
11            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
12            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
13            Scanner sc = new Scanner(System.in);
14
15            System.out.println("Connected to server.");
16            new Thread(() -> {
17                try {
18                    String serverMsg;
19                    while ((serverMsg = in.readLine()) != null) {
20                        System.out.println("[Server] " + serverMsg);
21                    }
22                } catch (IOException e) {
23                    System.out.println("Disconnected from server.");
24                }
25            }).start();
26            while (true) {
27                String msg = sc.nextLine();
28                out.println(msg);
29                if (msg.equalsIgnoreCase("bye")) break;
30            }
31        } catch (IOException e) {
32            System.out.println("Could not connect to server.");
33        }
34    }
35}
```

Client2



The screenshot shows a Java IDE interface with the title bar "Client2". The menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and Networking. The left sidebar has icons for Welcome, Client2, Client1.java, and Client2.java. The main editor area displays the following Java code:

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.Scanner;
4
5 public class Client2 {
6     private static final String SERVER_ADDRESS = "localhost";
7     private static final int SERVER_PORT = 12345;
8
9     public static void main(String[] args) {
10         try (Socket socket = new Socket(SERVER_ADDRESS, SERVER_PORT)) {
11             BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
12             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
13             Scanner sc = new Scanner(System.in);
14
15             System.out.println("Connected to server.");
16             new Thread(() -> {
17                 try {
18                     String serverMsg;
19                     while ((serverMsg = in.readLine()) != null) {
20                         System.out.println("[Server] " + serverMsg);
21                     }
22                 } catch (IOException e) {
23                     System.out.println("Disconnected from server.");
24                 }
25             }).start();
26             while (true) {
27                 String msg = sc.nextLine();
28                 out.println(msg);
29                 if (msg.equalsIgnoreCase("bye")) break;
30             }
31         } catch (IOException e) {
32             System.out.println("Could not connect to server.");
33         }
34     }
35 }
```

Figure 2: Implementation of the client-side code for Client1 and Client2, showing code handling messages received from the server.

5 Result Analysis

In this section, the outputs of the multi-threaded chat application for both the server and clients are shown. Multiple clients were able to connect to the server simultaneously, send messages, and receive responses without blocking. The server could also terminate the chat, notifying all clients.

5.1 Server Output

Server

```
Windows PowerShell          Windows PowerShell          Windows PowerShell
PS C:\Users\User\Documents\Networking> javac Server.java
PS C:\Users\User\Documents\Networking> java Server
Server started on port 12345
Enter <clientId> <message> or 'exit' to terminate: New client connected: Client 1 - Socket[addr=/127.0.0.1,port=59580,localport=12345]
New client connected: Client 2 - Socket[addr=/127.0.0.1,port=59581,localport=12345]
Client 1 says: Hello
Client 2 says: Hi!
2 How are you?
Client 2 says: Fine
1 Hi
Client 1 disconnected.
exit
Terminating server and notifying clients...
Connection lost with client 2
Client 2 disconnected.
PS C:\Users\User\Documents\Networking> |
```

Figure 3: Server console showing multiple client connections, messages received from clients, and server-initiated termination of chat.

5.2 Client Outputs

Client1

```
Windows PowerShell          Windows PowerShell          Windows PowerShell
PS C:\Users\User\Documents\Networking> javac Client1.java
PS C:\Users\User\Documents\Networking> java Client1
Connected to server.
[Server] Welcome! You are client #1. Type 'bye' to exit.
Hello
[Server] Server: Hi
bye
[Server] Goodbye!
Disconnected from server.
PS C:\Users\User\Documents\Networking> |
```

Client2

```
Windows PowerShell          Windows PowerShell          Windows PowerShell
PS C:\Users\User\Documents\Networking> javac Client2.java
PS C:\Users\User\Documents\Networking> java Client2
Connected to server.
[Server] Welcome! You are client #2. Type 'bye' to exit.
Hi!
[Server] Server: How are you?
Fine
[Server] Server has terminated the chat.
|
```

Figure 4: Client consoles showing received messages from the server and chat termination notification.

5.3 Description of Outputs

The outputs of the multi-threaded chat application demonstrate the successful implementation of concurrent client-server communication:

- **Server Output:** The server console shows that multiple clients were able to connect simultaneously. Each client is assigned a unique identifier, and messages received from each client are displayed in real-time. The server also has the capability to terminate the chat, which is reflected in the console by a termination message sent to all connected clients. This confirms that the server handles multiple client threads efficiently without blocking or crashing.
- **Client Outputs:** The client consoles display messages received from the server as well as messages sent by other clients, indicating that messages are correctly relayed by the server to all participants. When the server terminates the chat, the clients receive a notification about the termination, ensuring proper shutdown and communication closure. This demonstrates that the clients can handle asynchronous messages from the server while maintaining their own input-output flow.

Overall, the outputs confirm that the multi-threaded chat application is functioning correctly, providing real-time communication among multiple clients with robust server control.

6 Discussion

In this experiment, we explored the differences between basic socket programming and multi-threaded socket programming, and how the latter helps overcome the limitations of the former.

Basic Socket Programming

- Typically allows only one client to connect at a time.
- The server blocks while waiting for a client request, which limits scalability.
- Difficult to manage multiple client interactions concurrently.
- Inefficient for chat applications where simultaneous communication is essential.

Multi-threaded Socket Programming

- Each client is handled by a separate thread, allowing multiple clients to communicate with the server simultaneously.
- Non-blocking communication is achieved through thread pooling and message queues.
- Server can send targeted messages to specific clients while still listening for new connections.
- More robust and scalable, suitable for real-world applications such as chat servers and messaging systems.

Drawbacks of Basic Approach and Improvements

- **Drawback:** Single client restriction. **Improvement:** Multi-threading allows multiple clients.
- **Drawback:** Blocking I/O leads to idle waiting. **Improvement:** Separate threads and queues enable asynchronous message handling.
- **Drawback:** Hard to terminate connections gracefully. **Improvement:** Implemented a controlled shutdown mechanism from both client and server sides.

Learning and Challenges

- Learned how to implement client–server communication using sockets in Java.
- Understood how multi-threading enables parallelism in networking applications.
- Faced challenges in managing shared resources (client list, message queue), which were solved using thread-safe data structures like `ConcurrentHashMap`.
- Experienced issues in handling abrupt client disconnections, later fixed with exception handling and cleanup methods.

Overall, this lab provided hands-on experience with concurrent socket programming, and highlighted the importance of synchronization and graceful connection handling in network applications.