



Lab - 03 , Lab Report - 01

# **Design of a Chat Application for**

using Multi-Threaded Socket Programming

**CSE 3111: Computer Networking Lab**

Batch: 29 / 3rd Year 1st Semester 2024

**Date**

23 May, 2025

**Submitted by**

Aditto Raihan (Roll-09)

Jubair Ahammad Akter (Roll-59)

**Department of Computer Science and Engineering**

University of Dhaka

# 1 Introduction

**Socket programming** enables processes to communicate across a network using endpoints called *sockets*. Using the TCP protocol over sockets provides reliable, ordered, and byte-stream delivery, which is ideal for chat systems.

**Multi-threaded socket programming** extends this idea by dedicating a separate thread to each connected client (or using a worker pool), allowing the server to handle many clients concurrently. This avoids the blocking behavior of single-threaded designs where one slow client can stall everyone else.

For a chat application, concurrency is essential: multiple clients must be able to connect, send messages independently, and receive broadcasts from others with minimal latency. Our design uses:

- A TCP server socket listening on a fixed port.
- A *ClientHandler* thread per connection.
- A thread-safe list of clients to *broadcast* messages to all participants.
- Simple termination commands (`/quit` for clients, `/shutdown` for server).

## 2 Objectives

1. Design and implement a concurrent chat server that supports multiple clients simultaneously using TCP sockets and threads.
2. Implement message broadcasting so that a message from any client is visible to all connected clients in real time.
3. Provide clean termination mechanisms for both client and server, and handle abrupt disconnects gracefully.

## 3 Design Details

### System Components

- **Server:** Accepts incoming TCP connections, spawns a `ClientHandler` thread per client, maintains a concurrent client list, and broadcasts messages.
- **Client:** Connects to the server, launches a background reader thread to print messages from the server, and a foreground loop to send user input. Prompts for a display name; if empty, assigns `Anonymous1`, `Anonymous2`, etc.

## Protocol and Termination

- **Message format:** The client prepends `<Name>:`  to each message. The server broadcasts the already-tagged line to everyone.
- **Multiple sentences:** The server supports delimiting multiple sentences in a single line using `||`, broadcasting each segment.
- **Termination:** Client sends `/quit` to leave. Server console accepts `/shutdown` for graceful stop.

## Flow of Control (Textual)

1. **Server** starts, binds to port, enters accept loop.
2. For each connection, **Server** spawns **ClientHandler** and stores it in a thread-safe list.
3. **Client** connects, prompts for name (**AnonymousN** if empty), starts reader thread, then sends messages.
4. **Server** receives a line from a client, splits by `||`, and broadcasts each non-empty fragment to all clients.
5. On `/quit`, client closes; handler removes it and notifies others. On `/shutdown`, server notifies all and exits.

## Flowchart

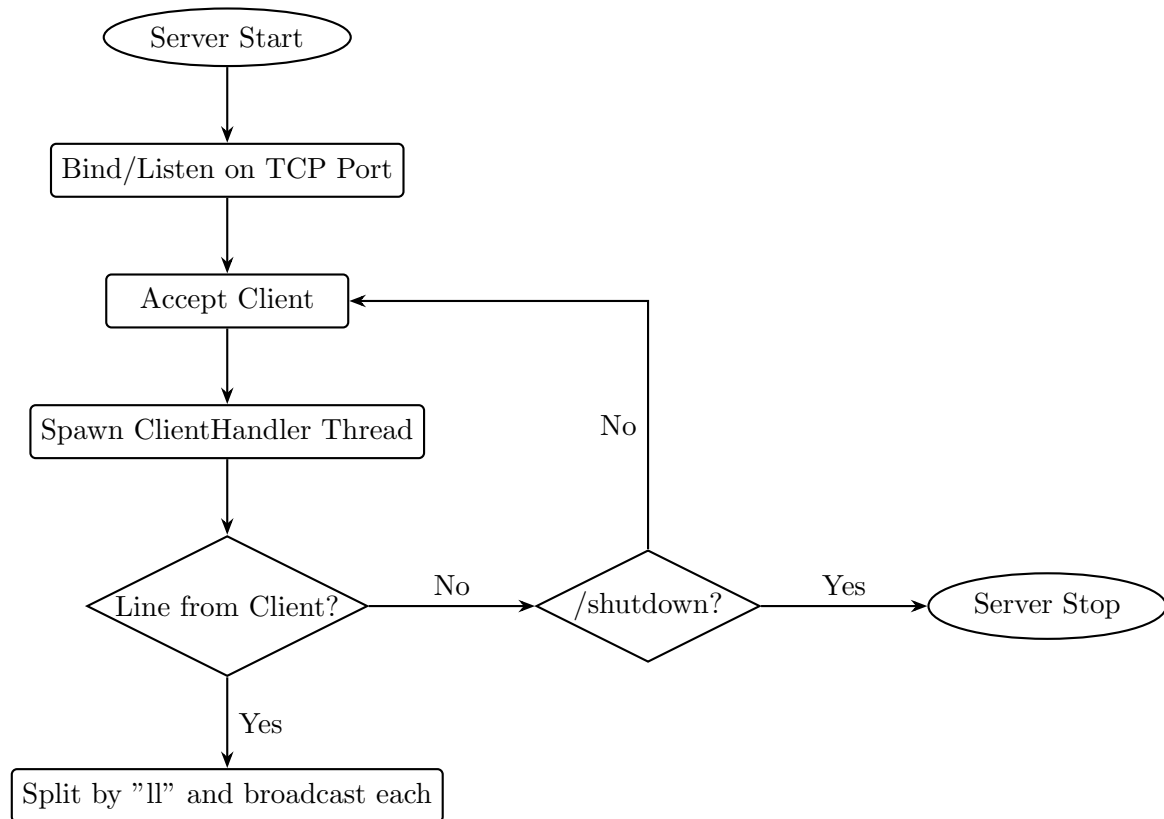


Figure 1: Server-side control flow for multi-threaded chat.

## 4 Implementation

### Environment

Ubuntu (Java SE, `default-jdk`); one machine runs the server, multiple machines/terminals run clients on the same LAN.

### Build & Run

# Server

```
javac Lab3b_09_59_server.java
```

```
java Lab3b_09_59_server
```

# Clients (update `SERVER_IP` if needed or pass as arg)

```
javac Lab3b_09_59_client.java
```

```
java Lab3b_09_59_client
```

## Screenshots (Server & Clients Code)

```

1 J Lab09_09_50_server.java : X
2
3 JUBAIR AHAMMAD AKTER > 29-3-17 LAB1.NET Lab3 > hw > J Lab09_09_50_server.java ...
4
5 1 // file: Lab09_09_50_server.java
6 2 import java.io.*;
7 3 import java.net.*;
8 4 import java.util.*;
9 5 import java.util.concurrent.CopyOnWriteArrayList;
10 6 import java.util.concurrent.atomic.AtomicInteger;
11 7
12 8 public class Lab09_09_50_server {
13 9     private static final int DEFAULT_PORT = 5000;
14 10
15 11     private final int port;
16 12     private volatile boolean running = true;
17 13     private ServerSocket serverSocket;
18 14     private final AtomicInteger clientIdGen = new AtomicInteger(1);
19 15     private final List<ClientHandler> clients = new CopyOnWriteArrayList<>();
20 16
21 17     public Lab09_09_50_server(int port) {
22 18         this.port = port;
23 19     }
24 20
25 21     public void start() {
26 22         try {
27 23             serverSocket = new ServerSocket(port);
28 24             System.out.println("[SERVER] Chat server started on port " + port);
29 25             System.out.println("[SERVER] Type /shutdown in this console to stop the server.");
30 26
31 27             // console thread for graceful shutdown
32 28             Thread console = new Thread(this::consoleLoop, "ServerConsole");
33 29             console.setDaemon(true);
34 30             console.start();
35 31
36 32             while (running) {
37 33                 try {
38 34                     Socket sock = serverSocket.accept();
39 35                     sock.setTcpNoDelay(true);
40 36                     int id = clientIdGen.getAndIncrement();
41 37                     ClientHandler handler = new ClientHandler(id, sock);
42 38                     clients.add(handler);
43 39                     handler.start();
44 40                     System.out.println("[SERVER] Clients" + id + " connected from " +
45 41                         sock.getInetAddress().getHostAddress() + ":" + sock.getPort());
46 42                     catch (SocketException se) {
47 43                         if (running) System.err.println("[SERVER] Socket error: " + se.getMessage());
48 44
49 45                     }
50 46                 } catch (IOException e) {
51 47                     System.err.println("[SERVER] Failed to start: " + e.getMessage());
52 48                 } finally {
53 49                     stopServer();
54 50                 }
55 51             }
56 52
57 53             private void consoleLoop() {
58 54                 try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {
59 55                     for (String line; running && (line = br.readLine()) != null; ) {
60 56                         if (line.trim().equalsIgnoreCase("/shutdown") ) {
61 57                             System.out.println("[SERVER] Shutdown requested.");
62 58                             running = false;
63 59                             closeServerSocket();
64 60                             broadcast([SERVER] Shutting down, goodbye., null, true);
65 61                             closeAllClients();
66 62                         }
67 63                     } catch (IOException ignored) {}
68 64                 }
69 65
70 66                 private void closeServerSocket() {
71 67                     try { if (serverSocket != null) serverSocket.close(); } catch (IOException ignored) {}
72 68
73 69                     private void closeAllClients() {
74 70                         for (ClientHandler ch : clients) ch.safeClose();
75 71
76 72                     private void stopServer() {
77 73                         running = false;
78 74                         closeServerSocket();
79 75                         closeAllClients();
80 76                         System.out.println("[SERVER] Stopped.");
81 77
82 78                     // Broadcast a message to all clients. If includeSender=false, skip the 'from' client. */
83 79                     private void broadcast(String msg, ClientHandler from, boolean includeSender) {

```

Figure 2: Server code

```

1 Lab3_09_09_client.java X
2 > JUBAR AHAMMAD AKTER > 29-3-1 > LAB1.NET > Lab3 > hw > > J Lab3_09_09_client.java > > Lab3_09_09_client > > main(String[])
3 // File: Lab3_09_09_client.java
4 import java.io.*;
5 import java.net.*;
6 import java.util.Scanner;
7
8 public class Lab3_09_09_client {
9     private static final String DEFAULT_HOST = "192.168.1.108"; // change to your server IP
10    private static final int DEFAULT_PORT = 5000;
11
12    // static counter for anonymous users
13    private static int anonymousCounter = 1;
14
15    public static void main(String[] args) {
16        String host = (args.length >= 1) ? args[0] : DEFAULT_HOST;
17        int port = (args.length >= 2) ? Integer.parseInt(args[1]) : DEFAULT_PORT;
18
19        System.out.println("(CLIENT) Connecting to " + host + " : " + port + " ...");
20
21        try (Socket socket = new Socket(host, port);
22            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream(), "UTF-8"));
23            BufferedWriter out = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream(), "UTF-8"));
24            Scanner sc = new Scanner(System.in)) {
25
26            socket.setTcpNoDelay(true);
27
28            // --- Ask for client name first ---
29            System.out.print("Enter your name: ");
30            String clientName = sc.nextLine().trim();
31
32            if (clientName.isEmpty()) {
33                clientName = "Anonymous" + anonymousCounter;
34                anonymousCounter++;
35            }
36
37            System.out.println("Welcome ~ " + clientName + " to the joined chat...");
38
39            // Reader thread for incoming messages
40            Thread reader = new Thread() -> {
41                try {
42                    String s;
43                    while ((s = in.readLine()) != null) {
44                        System.out.println(s);
45                    }
46                } catch (IOException e) {
47                    System.out.println("(CLIENT) Disconnected from server.");
48                }
49            };
50
51            public class Lab3_09_09_client {
52                public static void main(String[] args) {
53                    System.out.println("(CLIENT) You can now type messages.");
54                    System.out.println("[CLIENT] Use /quit to exit.");
55
56                    // Writer loop
57                    while (true) {
58                        if (sc.hasNextLine()) break;
59                        String line = sc.nextLine();
60                        if (line == null) break;
61
62                        String trimmed = line.trim();
63                        if (trimmed.isEmpty()) continue;
64
65                        if (trimmed.equalsIgnoreCase("/quit")) {
66                            writeLine(out, "/quit");
67                            break;
68                        }
69
70                        // Prepand name before sending
71                        String message = clientName + ": " + trimmed;
72                        writeLine(out, message);
73
74                        } catch (IOException e) {
75                            System.err.println("(CLIENT) Error: " + e.getMessage());
76                        }
77
78                        System.out.println("(CLIENT) Bye.");
79
80                        private static void writeLine(BufferedWriter out, String m) {
81                            try {
82                                out.write(m);
83                                out.newLine();
84                                out.flush();
85                            } catch (IOException e) {
86                                System.err.println("(CLIENT) Failed to send: " + e.getMessage());
87                            }
88                        }
89
90                        private static int parseIntOrDefault(String s, int def) {
91                            try { return Integer.parseInt(s); } catch (NumberFormatException e) { return def; }
92                        }
93                    }

```

Figure 3: Client code

Here is our all Server-Client code: [Networking Lab 3 Exercise](#)

## Screenshots (Server & Clients terminal Output)

```
student@student-Vostro-3910: ~/Downloads
[SERVER] Client#2 connected from 192.168.1.126:52628
[SERVER] Client#3 connected from 192.168.1.126:45176
^Z
[1]+  Stopped                  java Lab3b_09_59_server
student@student-Vostro-3910:~/Downloads$ sudo lsof -i :5000
COMMAND  PID  USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
java    13099  student  5u  IPv6  74900      0t0  TCP *:5000 (LISTEN)
java    13099  student  6u  IPv6  74901      0t0  TCP student-Vostro-3910:5000-
>student-Vostro-3910:60670 (ESTABLISHED)
java    13099  student  7u  IPv6  74902      0t0  TCP student-Vostro-3910:5000-
>student-Vostro-3910:52628 (ESTABLISHED)
java    13099  student  8u  IPv6  74903      0t0  TCP student-Vostro-3910:5000-
>student-Vostro-3910:45176 (ESTABLISHED)
student@student-Vostro-3910:~/Downloads$ kill -9 13099
student@student-Vostro-3910:~/Downloads$ javac Lab3b_09_59_server.java
^[[A[1]+  Killed                  java Lab3b_09_59_server
student@student-Vostro-3910:~/Downloads$ java Lab3b_09_59_server
[SERVER] Chat server started on port 5000
[SERVER] Type /shutdown in this console to stop the server.
[SERVER] Client#1 connected from 192.168.1.126:57890
[SERVER] Client#2 connected from 192.168.1.126:59240
[SERVER] Client#3 connected from 192.168.1.126:59242
[SERVER] Client#4 connected from 192.168.1.126:39168
```

Figure 4: Server console: client connections and broadcasts.

```
student@student-Vostro-3910:~/Downloads
student@student-Vostro-3910:~/Downloads$ javac Lab3b_09_59_client.java
student@student-Vostro-3910:~/Downloads$ java Lab3b_09_59_client
[CLIENT] Connecting to 192.168.1.108:5000 ...
Enter your name: Aditto
Welcome Aditto to the joined chat...
[CLIENT] You can now type messages.
[CLIENT] Use /quit to exit.
WELCOME Client#2!
Tips: Use '|' to send multiple sentences in one line.
Commands: /name <newname> | /quit
[SERVER] Client#3 joined the chat.
[Client#1]: Adib: Hi this is Adib
Hi this is Aditto
[Client#2]: Aditto: Hi this is Aditto
[Client#3]: Ninad: Hi this is Ninad
[SERVER] Client#4 joined the chat.
[Client#4]: Tausif: Hi this is Tausif

student@student-Vostro-3910:~/Downloads
student@student-Vostro-3910:~/Downloads$ javac Lab3b_09_59_client.java
student@student-Vostro-3910:~/Downloads$ java Lab3b_09_59_client
[CLIENT] Connecting to 192.168.1.108:5000 ...
Enter your name: Adib
Welcome Adib to the joined chat...
[CLIENT] You can now type messages.
[CLIENT] Use /quit to exit.
WELCOME Client#1!
Tips: Use '|' to send multiple sentences in one line.
Commands: /name <newname> | /quit
[SERVER] Client#2 joined the chat.
[SERVER] Client#3 joined the chat.
Hi this is Adib
[Client#1]: Adib: Hi this is Adib
[Client#2]: Aditto: Hi this is Aditto
[Client#3]: Ninad: Hi this is Ninad
[SERVER] Client#4 joined the chat.
[Client#4]: Tausif: Hi this is Tausif

student@student-Vostro-3910:~/Downloads
student@student-Vostro-3910:~/Downloads$ javac Lab3b_09_59_client.java
student@student-Vostro-3910:~/Downloads$ java Lab3b_09_59_client
[CLIENT] Connecting to 192.168.1.108:5000 ...
Enter your name: Ninad
Welcome Ninad to the joined chat...
[CLIENT] You can now type messages.
[CLIENT] Use /quit to exit.
WELCOME Client#3!
Tips: Use '|' to send multiple sentences in one line.
Commands: /name <newname> | /quit
[Client#1]: Adib: Hi this is Adib
[Client#2]: Aditto: Hi this is Aditto
Hi this is Ninad
[Client#3]: Ninad: Hi this is Ninad
[SERVER] Client#4 joined the chat.
[Client#4]: Tausif: Hi this is Tausif

student@student-Vostro-3910:~/Downloads
student@student-Vostro-3910:~/Downloads$ javac Lab3b_09_59_client.java
student@student-Vostro-3910:~/Downloads$ java Lab3b_09_59_client
[CLIENT] Connecting to 192.168.1.108:5000 ...
Enter your name: Tausif
Welcome Tausif to the joined chat...
[CLIENT] You can now type messages.
[CLIENT] Use /quit to exit.
WELCOME Client#4!
Tips: Use '|' to send multiple sentences in one line.
Commands: /name <newname> | /quit
Hi this is Tausif
[Client#4]: Tausif: Hi this is Tausif
```

Figure 5: Client terminals: names and broadcasted messages.

## 5 Result Analysis

**Functional correctness.** The server accepted multiple simultaneous clients; messages from any client appeared on every other client with the “`Name: message`” format. New clients received join notifications, and disconnections were handled gracefully.

**Responsiveness.** Using a dedicated handler thread per client kept I/O responsive even when some clients typed slowly or paused.

**Robustness.** The server tolerated abrupt client exits; handler threads cleaned up and removed themselves from the broadcast list.

## 6 Discussion

### Basic vs. Multi-Threaded Socket Programming

**Basic (single-threaded)** servers typically block on I/O: handling one client at a time leads to head-of-line blocking and poor scalability. A long `read()` or a slow client delays everyone else.

**Multi-threaded** servers assign each client to a separate thread (or to a thread pool with non-blocking I/O), enabling concurrency. In our design:

- Each `ClientHandler` performs blocking reads without affecting others.
- A thread-safe list (e.g., `CopyOnWriteArrayList`) supports concurrent broadcasts safely.

### Drawbacks and Mitigation

- **Thread per client overhead:** Many threads can exhaust resources. A scalable alternative is NIO (non-blocking I/O) with a selector or a fixed thread pool.
- **Broadcast contention:** Many concurrent writers may contend on output streams. Batching or a message queue per client can help.
- **Failure handling:** Unexpected disconnects must remove clients promptly; our handler ensures cleanup in `finally`.

### Learning and Challenges

We practiced TCP socket APIs, concurrency, thread-safe collections, and simple application-level protocols (naming, commands, termination). Debugging focused on port conflicts, LAN IP mismatches, and ensuring broadcast visibility across clients.