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

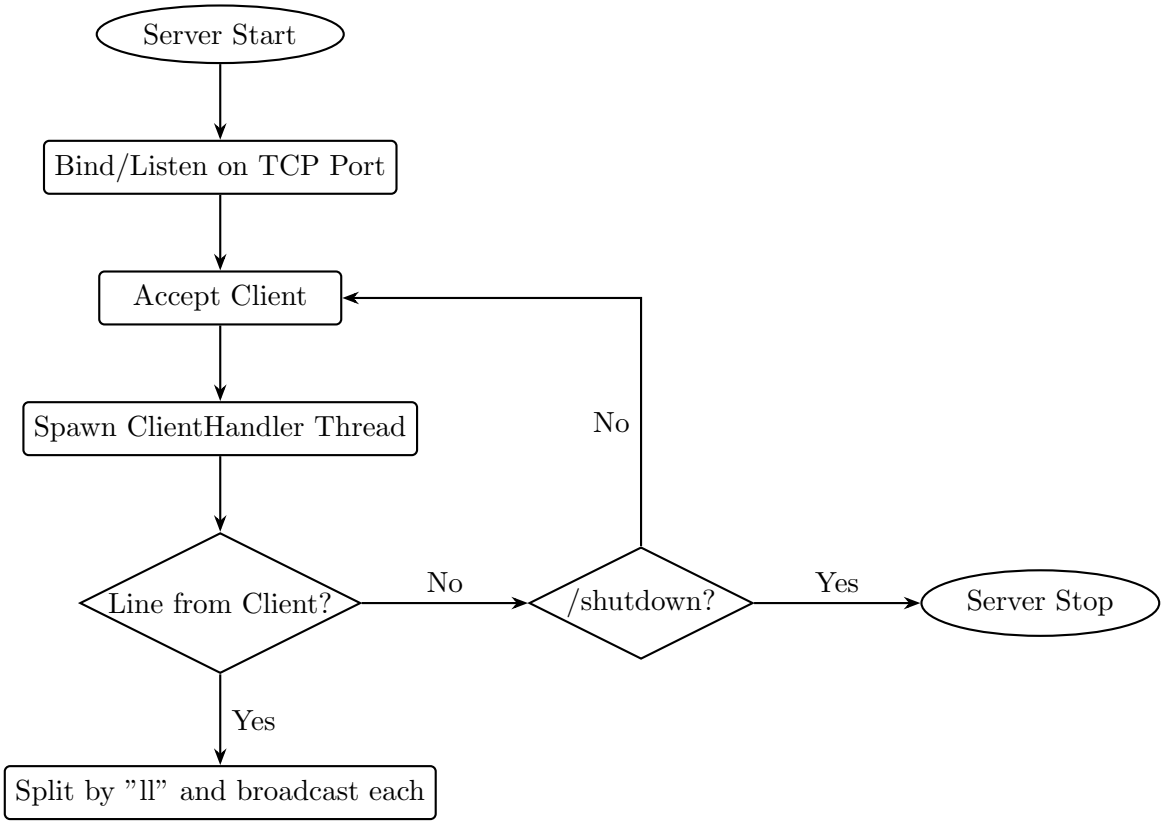


Figure 1: Server-side control flow for multi-threaded chat.

4 Implementation

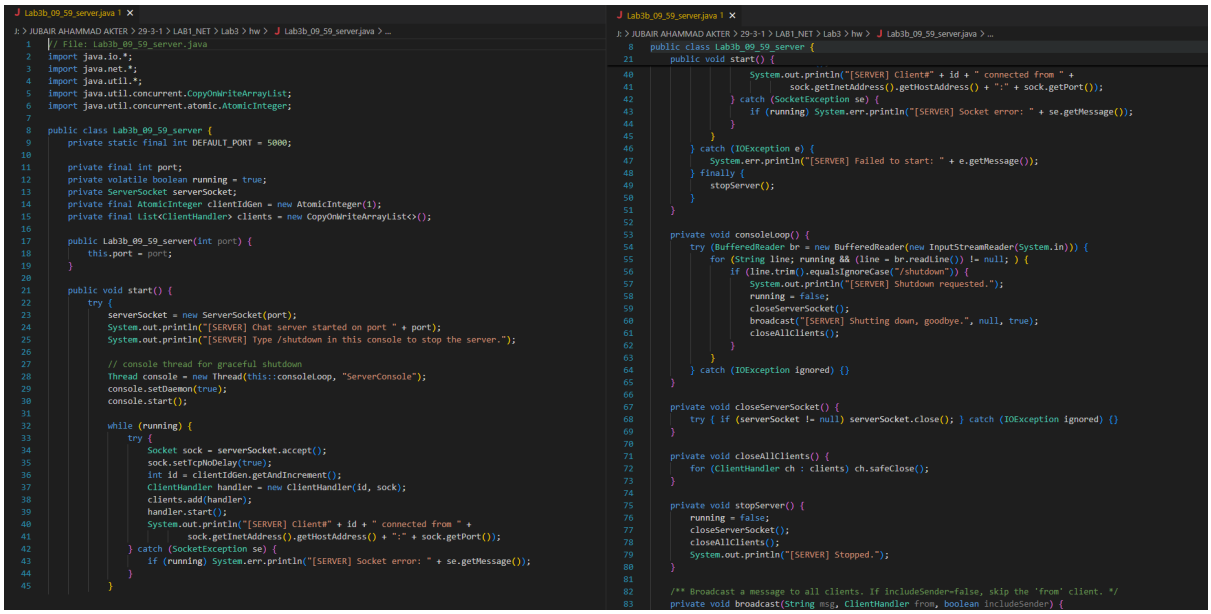
Environment

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
```

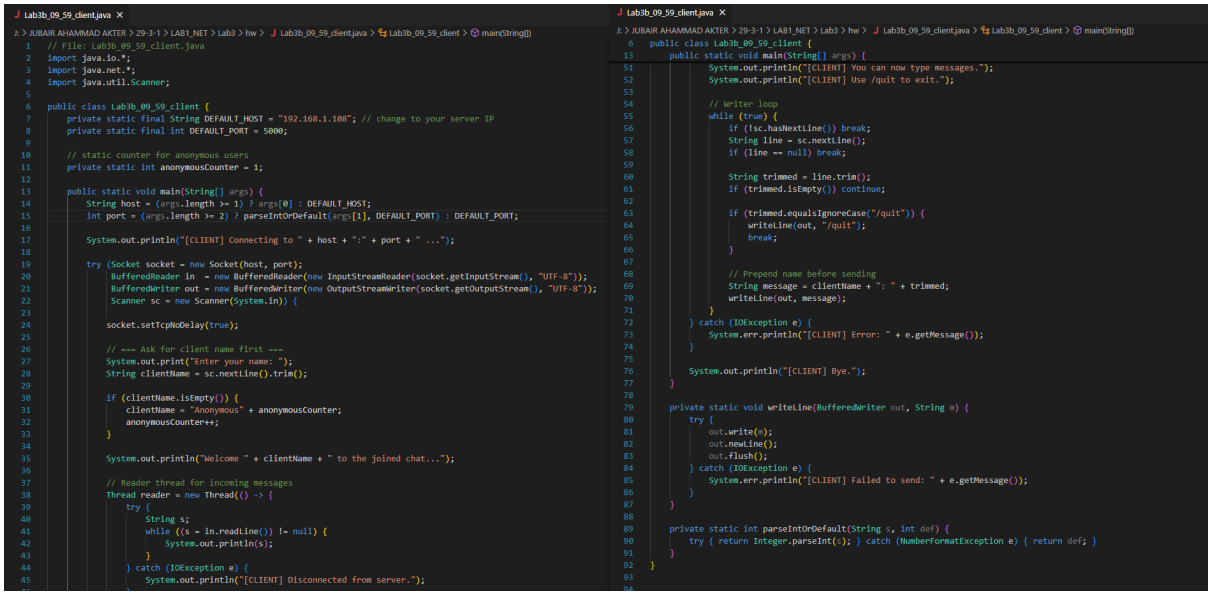
Screenshots (Server & Clients Code)



```java
// File: Lab3b_09_59_server.java
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicInteger;

public class Lab3b_09_59_server {
    private static final int DEFAULT_PORT = 5000;

    private final int port;
    private volatile boolean running = true;
    private ServerSocket serverSocket;
    private final AtomicInteger clientIdGen = new AtomicInteger(1);
    private final List<ClientHandler> clients = new CopyOnWriteArrayList<>();

    public Lab3b_09_59_server(int port) {
        this.port = port;
    }

    public void start() {
        try {
            serverSocket = new ServerSocket(port);
            System.out.println("[SERVER] Chat server started on port " + port);
            System.out.println("[SERVER] Type /shutdown in this console to stop the server.");

            // console thread for graceful shutdown
            Thread console = new Thread(this::consoleLoop, "ServerConsole");
            console.setDaemon(true);
            console.start();

            while (running) {
                try {
                    Socket sock = serverSocket.accept();
                    sock.setTcpNoDelay(true);
                    int id = clientIdGen.getAndIncrement();
                    ClientHandler handler = new ClientHandler(id, sock);
                    clients.add(handler);
                    handler.start();
                    System.out.println("[SERVER] Client#" + id + " connected from " +
                            sock.getInetAddress().getHostAddress() + ":" + sock.getPort());
                } catch (SocketException se) {
                    if (running) System.err.println("[SERVER] Socket error: " + se.getMessage());
                }
            }
```

```java
public class Lab3b_09_59_server {
    public void start() {
                    System.out.println("[SERVER] Client#" + id + " connected from " +
                            sock.getInetAddress().getHostAddress() + ":" + sock.getPort());
                } catch (SocketException se) {
                    if (running) System.err.println("[SERVER] Socket error: " + se.getMessage());
                }
            }
        } catch (IOException e) {
            System.err.println("[SERVER] Failed to start: " + e.getMessage());
        } finally {
            stopServer();
        }
    }

    private void consoleLoop() {
        try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {
            for (String line; running && (line = br.readLine()) != null; ) {
                if (line.trim().equalsIgnoreCase("/shutdown")) {
                    System.out.println("[SERVER] Shutdown requested.");
                    running = false;
                    closeServerSocket();
                    broadcast("[SERVER] Shutting down, goodbye.", null, true);
                    closeAllClients();
                }
            }
        } catch (IOException ignored) {}
    }

    private void closeServerSocket() {
        try { if (serverSocket != null) serverSocket.close(); } catch (IOException ignored) {}
    }

    private void closeAllClients() {
        for (ClientHandler ch : clients) ch.safeClose();
    }

    private void stopServer() {
        running = false;
        closeServerSocket();
        closeAllClients();
        System.out.println("[SERVER] Stopped.");
    }

    /** Broadcast a message to all clients. If includeSender=false, skip the 'from' client. */
    private void broadcast(String msg, ClientHandler from, boolean includeSender) {
```

Figure 2: Server code



```java
// File: Lab3b_09_59_client.java
import java.io.*;
import java.net.*;
import java.util.Scanner;

public class Lab3b_09_59_client {
    private static final String DEFAULT_HOST = "192.168.1.108"; // change to your server IP
    private static final int DEFAULT_PORT = 5000;

    // static counter for anonymous users
    private static int anonymousCounter = 1;

    public static void main(String[] args) {
        String host = (args.length >= 1) ? args[0] : DEFAULT_HOST;
        int port = (args.length >= 2) ? parseIntOrDefault(args[1], DEFAULT_PORT) : DEFAULT_PORT;

        System.out.println("[CLIENT] Connecting to " + host + ":" + port + " ...");

        try (Socket socket = new Socket(host, port);
             BufferedReader in  = new BufferedReader(new InputStreamReader(socket.getInputStream(), "UTF-8"));
             BufferedWriter out = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream(), "UTF-8"));
             Scanner sc = new Scanner(System.in)) {

            socket.setTcpNoDelay(true);

            // === Ask for client name first ===
            System.out.print("Enter your name: ");
            String clientName = sc.nextLine().trim();

            if (clientName.isEmpty()) {
                clientName = "Anonymous" + anonymousCounter;
                anonymousCounter++;
            }

            System.out.println("Welcome " + clientName + " to the joined chat...");

            // Reader thread for incoming messages
            Thread reader = new Thread(() -> {
                try {
                    String s;
                    while ((s = in.readLine()) != null) {
                        System.out.println(s);
                    }
                } catch (IOException e) {
                    System.out.println("[CLIENT] Disconnected from server.");
```

```java
public class Lab3b_09_59_client {
    public static void main(String[] args) {
            System.out.println("[CLIENT] You can now type messages.");
            System.out.println("[CLIENT] Use /quit to exit.");

            // Writer loop
            while (true) {
                if (!sc.hasNextLine()) break;
                String line = sc.nextLine();
                if (line == null) break;

                String trimmed = line.trim();
                if (trimmed.isEmpty()) continue;

                if (trimmed.equalsIgnoreCase("/quit")) {
                    writeLine(out, "/quit");
                    break;
                }

                // Prepend name before sending
                String message = clientName + ": " + trimmed;
                writeLine(out, message);
            }
        } catch (IOException e) {
            System.err.println("[CLIENT] Error: " + e.getMessage());
        }

        System.out.println("[CLIENT] Bye.");
    }

    private static void writeLine(BufferedWriter out, String m) {
        try {
            out.write(m);
            out.newLine();
            out.flush();
        } catch (IOException e) {
            System.err.println("[CLIENT] Failed to send: " + e.getMessage());
        }
    }

    private static int parseIntOrDefault(String s, int def) {
        try { return Integer.parseInt(s); } catch (NumberFormatException e) { return def; }
    }
}
```

Figure 3: Client code

Here is our all Server-Client code: Networking Lab 3 Exercise

Screenshots (Server & Clients terminal Output)

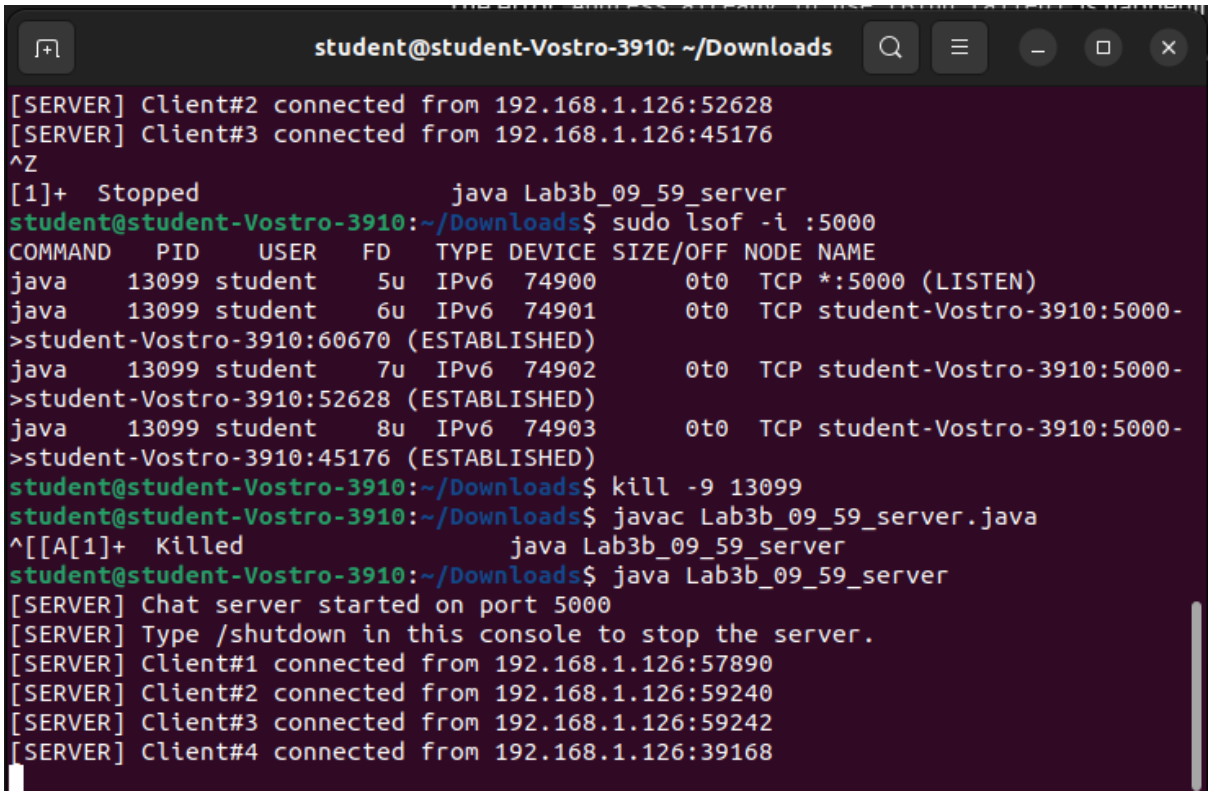


Figure 4: Server console: client connections and broadcasts.

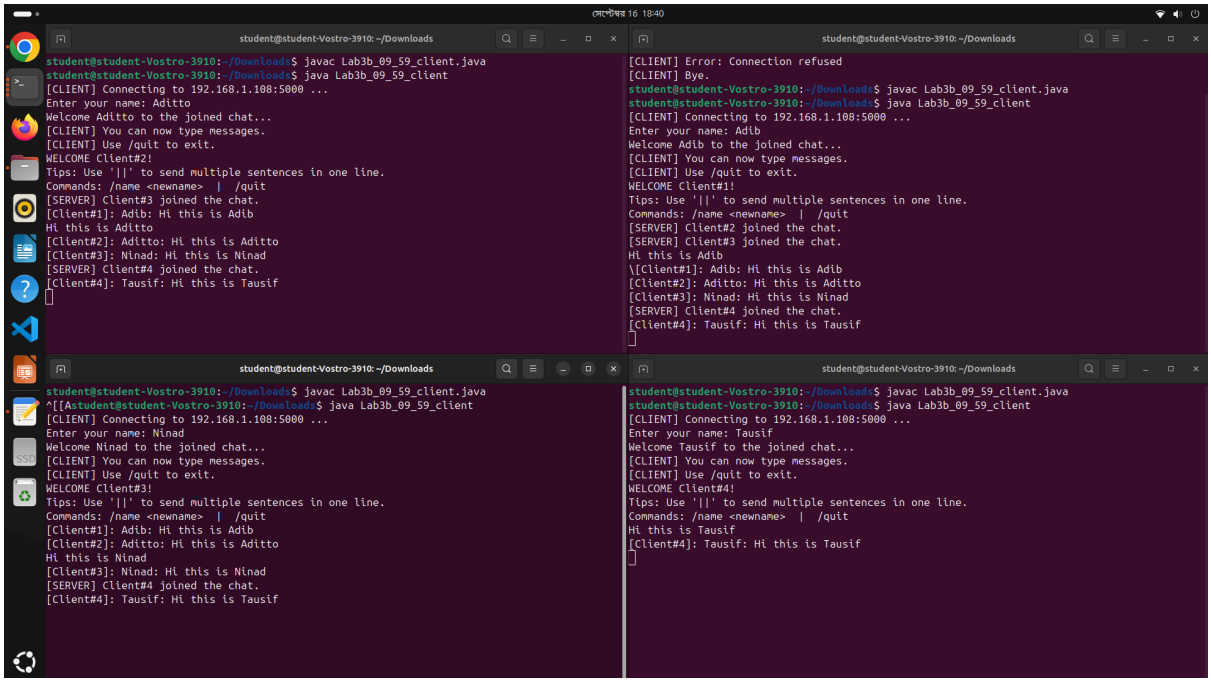


Figure 5: Client terminals: names and broadcasted messages.

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.