Design of a Chat Application Using Multi-threaded Socket Programming

H.M. Mehedi Hasan **(13)**
MD. Abu Bakar Siddique **(47)**

September 16, 2025

Contents

1 Introduction

Socket programming is a fundamental technique that allows communication between processes over a network through endpoints called sockets. It forms the interface between application processes and the transport layer protocols such as TCP and UDP. This method is crucial in networked applications like chat systems where data exchange between clients and servers is constant and interactive.

Multi-threaded socket programming extends basic socket communication by employing multiple threads on the server side, enabling simultaneous handling of multiple clients. Each client is served by a dedicated thread that manages its connection independently. This concurrency is crucial for chat applications where multiple users interact in real time; it prevents blocking that occurs in basic single-threaded socket servers, thus ensuring responsiveness and scalability.

Without multi-threading, a server using basic socket programming can accept and communicate with only one client at a time, causing other client requests to wait indefinitely—making it impractical for chat applications.

2 Objectives

1. To understand the core concepts of socket programming and the necessity of multi-threading.
2. To design and implement a multi-threaded chat server that can simultaneously serve multiple clients.
3. To enable continuous bi-directional communication where clients can send multiple messages and receive replies concurrently.

3 Design Details

3.1 Implementation Process

1. **Server Setup:**
   (a) The server creates a `ServerSocket` listening on port `3000`.
   (b) It continuously accepts new client connections.
2. **Client Connection Handling:**
   (a) Upon client connection, the server creates input and output streams (`DataInputStream` and `DataOutputStream`).
   (b) A new thread (`ClientHandlerThread`) is spawned for each client, passing the socket and streams.
   (c) The thread is responsible for communicating exclusively with its associated client.
3. **ClientHandlerThread Details:**

(a) Reads the client's name upon joining.

(b) Continuously listens for messages from the client.

(c) If the client sends `"quit"`, the thread closes the connection and removes itself from the active client list.

(d) Broadcasts messages received from the client to all other connected clients.

4. **Client Setup:**

(a) The client connects to the server using the IP `localhost` and port `3000`.

(b) Sends its user name immediately after connecting.

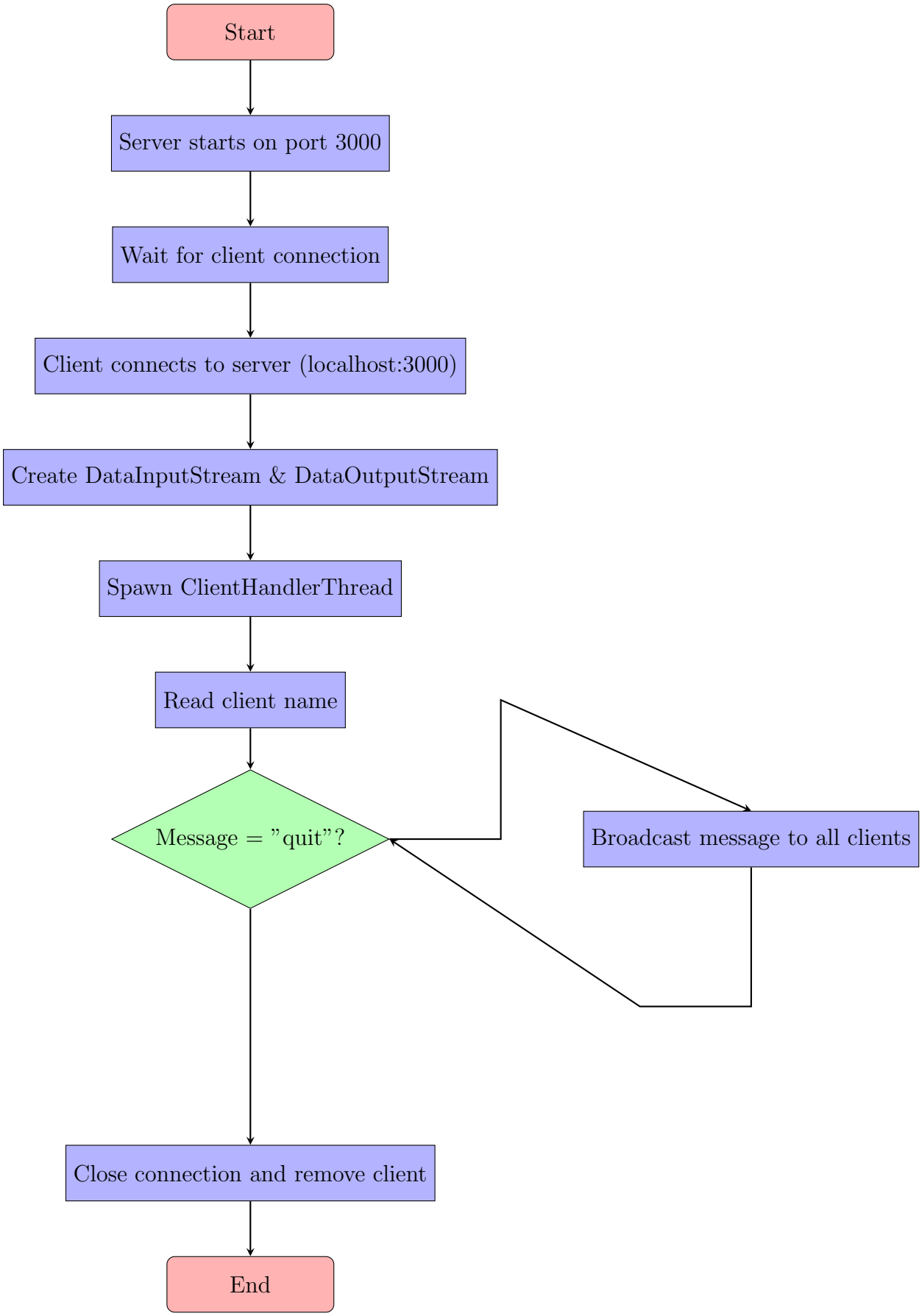
(c) Has two threads:

i. One for reading user input and sending messages to the server.

ii. One for listening to incoming messages from the server and displaying them.

(d) The client terminates when the user types `"quit"`.

3.2 Implementation Process (Flowchart)



4 Implementation

The implementation consists of two main Java classes: Server.java and Client.java. Below are the key components of each:

4.1 Server Implementation

```
1
2  import java.io.DataInputStream;
3  import java.io.DataOutputStream;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6  import java.util.Scanner;
7  import java.util.Vector;
8
9  public class Server_13_47 {
10     public static Vector<ClientHandlerThread> clientThreads = new
           Vector<>();
11     public static int selectedIndex = 0;
12
13     public static void main(String[] args) throws Exception {
14
15         ServerSocket ss = new ServerSocket(3000);
16         System.out.println("New Server Created");
17         Scanner sc = new Scanner(System.in);
18         new Thread(() -> {
19             try {
20                 while (true) {
21                     Socket s = ss.accept();
22
23                     DataInputStream in = new DataInputStream(s.
                           getInputStream());
24                     DataOutputStream out = new DataOutputStream(s.
                           getOutputStream());
25
26                     ClientHandlerThread thread = new
                           ClientHandlerThread(s, in, out, clientThreads.
                           size());
27                     clientThreads.add(thread);
28
29                     thread.start();
30                 }
31             } catch (Exception e) {
32                System.out.println("Error in accepting client" + e);
33             }
34
35         }).start();
36
37         new Thread(() -> {
38             try {
39                 while (true) {
40                     System.out.print("\033[H\033[2J");
41                     System.out.flush();
42
43                     System.out.println("0. Exit");
44                     String read = sc.nextLine();
```

```
45                      if (read.equals("0")) {
46                          System.out.println("Quitting...");
47                          System.exit(0);
48                          break;
49                      }
50
51                  }
52              } catch (Exception e) {
53                  System.out.println("Error in i/o thread");
54              }
55
56          }).start();
57
58      }
59  }
60
61  class ClientHandlerThread extends Thread {
62      public Socket s;
63      DataInputStream in;
64      DataOutputStream out;
65      String name;
66      Vector<ClientHandlerThread> clientList;
67
68      public ClientHandlerThread(Socket s, DataInputStream in,
            DataOutputStream out, int index) {
69          this.s = s;
70          this.in = in;
71          this.out = out;
72          this.clientList = Server_13_47.clientThreads;
73      }
74
75      @Override
76      public void run() {
77          try {
78
79              name = in.readUTF();
80              System.out.println(name + " joined the server!");
81
82              while (true) {
83                  String message = in.readUTF();
84
85                  if (message.equals("quit")) {
86                      System.out.println(name + " left the server!");
87                      break;
88                  }
89
90                  broadcastMessage(name + ": " + message, this);
91              }
92
93          } catch (Exception e) {
94              System.out.println(name + " disconnected unexpectedly");
95          } finally {
96              clientList.remove(this);
97              try {
98                  s.close();
99              } catch (Exception e) {
100                 System.out.println("Error closing socket");
101             }
```

```
102         }
103     }
104 
105     private void broadcastMessage(String message, ClientHandlerThread
        sender) {
106         for (ClientHandlerThread client : clientList) {
107             if (client != sender) {
108                 try {
109                     client.out.writeUTF(message);
110                 } catch (Exception e) {
111                     System.out.println("Failed to send message to " +
                        client.name);
112                 }
113             }
114         }
115     }
116 
117 }
```

Listing 1: Server.java - Main Server Class

4.2 Client Implementation

```
1  import java.io.DataInputStream;
2  import java.io.DataOutputStream;
3  import java.net.Socket;
4  import java.util.Scanner;
5  import java.util.concurrent.atomic.AtomicBoolean;
6 
7  public class Client_13_47 {
8      public static void main(String[] args) throws Exception {
9 
10         System.out.print("\033[H\033[2J");
11         System.out.flush();
12         Scanner sc = new Scanner(System.in);
13 
14         String name = "";
15 
16         System.out.print("Enter Name:");
17         name = sc.nextLine();
18 
19         Socket s = new Socket("localhost", 3000);
20         DataOutputStream out = new DataOutputStream(s.getOutputStream()
               );
21         DataInputStream in = new DataInputStream(s.getInputStream());
22 
23         System.out.println("Joined Server: "+ name);
24         System.out.print("\033[H\033[2J");
25         System.out.flush();
26         System.out.println("Welcome " + name + "!");
27         System.out.println("Type your messages below. Type 'quit' to
               exit.");
28         out.writeUTF(name);
29         AtomicBoolean running = new AtomicBoolean(true);
30         new Thread(() -> {
31             try {
32                 while (running.get()) {
```

```
                    String message = sc.nextLine();
                    if (message.equals("quit")) {
                        running.set(false);
                        out.writeUTF(message);
                        break;
                    }
                    if (!message.trim().isEmpty()) {
                        System.out.println("You: " + message);
                        out.writeUTF(message);
                    }
                }
            } catch (Exception e) {
                System.out.println("Error sending message");
            }
        }).start();

        new Thread(() -> {
            try {
                while (running.get()) {
                    String message = in.readUTF();
                    System.out.println(message);
                }
            } catch (Exception e) {
                if (running.get()) {
                    System.out.println("Connection to server lost");
                }
            }
        }).start();
    }
}
```

Listing 2: Client.java - Main Client Class

5 Result Analysis

The implemented chat application successfully demonstrates the principles of multi-threaded socket programming, allowing for real-time communication between a server and multiple clients.

5.1 Server Output

```
PS E:\CSEDU\3-1\Networking\Report\LabReport1> javac .\Server_13_47.java
PS E:\CSEDU\3-1\Networking\Report\LabReport1> java .\Server_13_47.java
New Server Created
0. Exit
Abs joined the server!
Mehedi joined the server!
Abs left the server!
Mehedi left the server!
0
Quitting...
PS E:\CSEDU\3-1\Networking\Report\LabReport1>
```

Figure 1: Server Ouput

The server console displays:

- Listens on port 3000.
- Manages all connected clients in a thread-safe Vector.
- Spawns a dedicated ClientHandlerThread per client.
- Supports broadcasting messages to multiple clients.
- Provides a console interface to exit the server.

5.2 Client Output

```
PS E:\CSEDU\3-1\Networking\Report\LabReport1> javac .\Client_13_47.java
PS E:\CSEDU\3-1\Networking\Report\LabReport1> java .\Client_13_47.java
Enter Name:Abs
Joined Server: Abs
Welcome Abs!
Type your messages below. Type 'quit' to exit.
Hello
You: Hello
Mehedi: Yooo
Whatsssapp
You: Whatsssapp
Mehedi: chat apppp
tata
You: tata
quit
PS E:\CSEDU\3-1\Networking\Report\LabReport1>
```

Figure 2: Client 1 Ouput

```
PS E:\CSEDU\3-1\Networking\Report\LabReport1> javac .\Client_13_47.java
PS E:\CSEDU\3-1\Networking\Report\LabReport1> java .\Client_13_47.java
Enter Name:Mehedi
Joined Server: Mehedi
Welcome Mehedi!
Type your messages below. Type 'quit' to exit.
Abs: Hello
Yooo
You: Yooo
Abs: Whatsssapp
chat apppp
You: chat apppp
Abs: tata
quit
PS E:\CSEDU\3-1\Networking\Report\LabReport1>
```

Figure 3: Client 2 Output

The client console displays:

- Connects to server at localhost port 3000.
- Sends user name to server on connect.
- Runs two threads:
  - For sending messages input by the user.
  - For receiving and printing messages from server.
- Supports sending multiple messages asynchronously.
- Stops on user input "quit".

6 Discussion

In this project, we explored both basic socket programming and multi-threaded socket programming. The comparison between the two approaches highlights the advantages and drawbacks of each.

6.1 Basic Socket Programming

- In basic socket programming, a server can handle only one client at a time.
- The server must wait for the current client to disconnect before accepting a new connection.
- This approach is simple to implement but becomes impractical when multiple clients need to communicate simultaneously.
- The main drawback is that client requests are processed sequentially, which may cause delays and poor performance.

6.2 Multi-Threaded Socket Programming

- In multi-threaded socket programming, a new thread is spawned for each connected client.
- This allows the server to handle multiple clients concurrently without blocking other connections.
- It improves responsiveness and scalability, as clients can send and receive messages independently.
- Although this introduces additional complexity in managing threads and shared resources, the overall performance and user experience are significantly better.

6.3 Overcoming the Drawbacks

- The limitations of basic socket programming, such as sequential client handling and poor scalability, are overcome by using multi-threading.
- With this program, we were able to broadcast messages from one client to all others efficiently, which would not be feasible in a single-threaded setup.
- Thread-based design ensures that a single client's disconnection or delay does not affect the entire server's operation.

6.4 Learning Outcomes

- We learned how to establish socket connections and exchange data using input and output streams.
- The project gave us hands-on experience with creating and managing threads in a networked environment.
- We also gained insights into concurrency issues, synchronization, and resource management in multi-threaded systems.

6.5 Challenges Faced

- Understanding how to properly manage multiple threads was a significant challenge, especially when handling simultaneous client messages.
- Ensuring that the server does not crash when a client abruptly disconnects required careful exception handling.
- Debugging concurrent code was more complex compared to basic socket programming.
- Despite these difficulties, the process enhanced our understanding of both networking and multithreading concepts.