

# University of Dhaka

Department of Computer Science and  
Engineering

**CSE 3111 - Computer Networking Lab**

Batch 28 / 3rd Year 1st Semester

Session: 2021-22

---

**Implementation of a fully functional chat  
application using multi-threaded Socket  
Programming**

---

**Submitted To:**

Dr. Ismat Rahman (IsR)

Mr. Jargis Ahmed (JA)

Mr. Palash Roy (PR)

**Group No: B\_10(Even)**

**Submitted By:**

Farzana Tasnim (14)

Amina Islam (36)

# 1 Introduction

Socket programming is a fundamental technique of enabling devices to communicate over a network. It provides a mechanism for programs to send data over networks, such as the internet, using standardized protocols such as HTTP, FTP, TCP or UDP. The client-server model is a common pattern used in socket programming in which one program (the server) provides services or resources, and another program (the client) asks for the services.

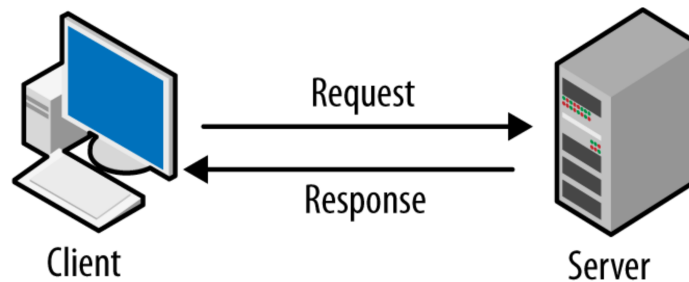


Figure 1: Client-Server Architecture.

The client starts a socket, connects with the server, sends data within the session and receives the response of the server and closes the socket upon detecting an "End of File" indicator. The server creates a socket, binds it to an IP address and port number, waits for the request, accepts the client connection, and engages in the session by reading client data and writing responses until the session ends with an "End of File," then closes the connection. The client/server session, which is surrounded by a yellow box, involves active data transfer via write and read operations, all made possible by the Socket API, thus offering reliable TCP communication via such functions as `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `read()`, `write()`, and `close()`.

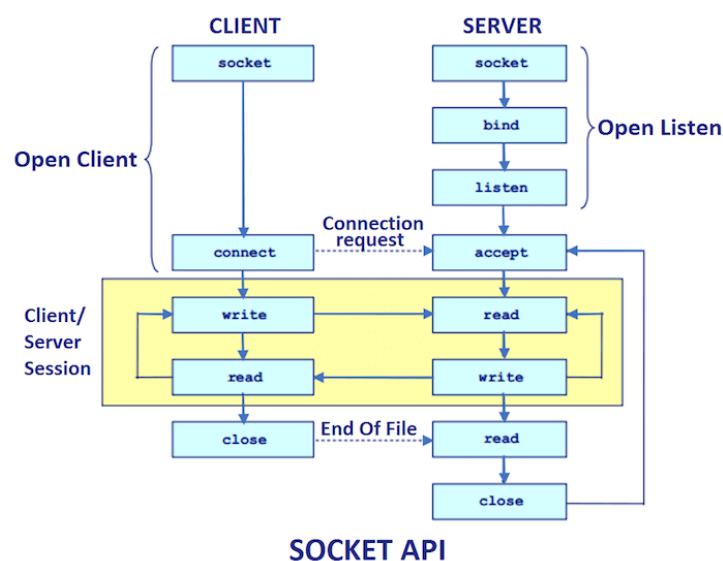


Figure 2: Client-Server communication using sockets.

A thread is a light-weight process that does not require much memory overhead, they are cheaper than processes. Multithreading is a process of executing multiple threads simultaneously in a single process.

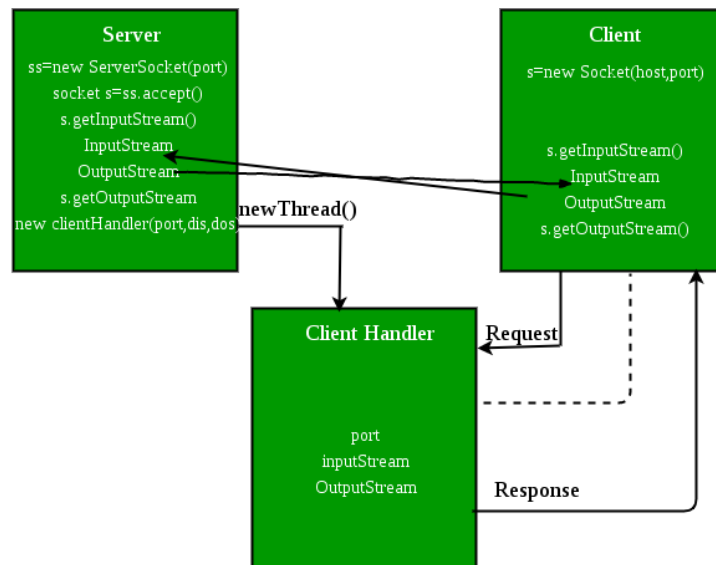


Figure 3: Multi-thread in socket programming.

Multithreaded socket programming involves creating a server application that can handle multiple client connections concurrently by using multiple threads to manage each connection. This allows the server to serve multiple clients simultaneously, improving performance and responsiveness compared to a single-threaded approach. One client's processing doesn't block others in multi-thread.

In a chat application, we need connection between two to that they can transfer message. So we need a server client concept of socket programming. To handle each client separately at a time, gives smooth service to multi-client we need multi-thread socket programming.

## 2 Objectives

1. To gather knowledge of socket programming using multiple threading
2. Create a fully functional chat application that allows multiple clients to connect simultaneously to a single server, which provides real-time communication using multi-thread socket programming.

## 3 Design Details

To design a chat application, we need to implement the server and client side code.

## 3.1 Server Side Programming

On the server side, we need to create three classes. The server class, the clientHandler class and another is WorkOfServer class.

### 3.1.1 Server class

- **Establishing the Connection:** Server socket object is initialized, and inside a loop, a socket object continuously accepts incoming connections.
- **Obtaining the Streams:** The bufferedReader object and PrintWriter object are used for the current requests' socket object.
- **Invoking the start method of WorkOfServer object :** To start the thread that will handle server work such as sending responds to the clients and close the server.
- **Invoking the start method of ClientHandler object :** Inside a loop, it will accept all the incoming request of clients and create a clientHandler object to handle each client separately.
- **Method to specific work :** There are two methods - one is sendMessageToClient and another is removeClient to send message to client and to remove client from the current client list.

### 3.1.2 ClientHandler class

- First of all, this class extends Thread so that its objects assume all properties of Threads.
- clientHandler object is created with socket, clientId, bufferedReader, PrintWriter parameters. This class will read the message that is sent from the client.
- Inside run() it performs specific operation.

### 3.1.3 WorkOfServer class

- First of all, this class extends Thread so that its objects assume all properties of Threads.
- WorkOfServer class is created with serverSocket parameter. This class will handle the work of server - to response to the message of a client and shutting down the server.
- Inside run() it performs specific operation.

## 3.2 Client Side Programming

### 3.2.1 Client Class

- Establish a Socket Connection

- Communication with server. ReadServer object created to read response from server.
- Closing the Connection

### 3.2.2 ReadServer Class

- First of all, this class extends Thread so that its objects assume all properties of Threads.
- ReadServer class is created with bufferedReader parameter. This class will read the response from the server.
- Inside run() it performs specific operation.

---

#### Algorithm 1: Algorithm of Server Side Socket Connection

---

- 1: Create a ServerSocket object, namely handshaking socket, which takes the port number as input. Also an arraylist to store the current clients.
  - 2: Create two objects of the BufferedReader and PrintWriter classes, which are used for reading and sending data, respectively.
  - 3: Create an object for WorkOfServer Thread class and pass the serverSocket as the parameters. Inside the run() method, do necessary work to handle the work of server.
  - 4: Create a plain Socket, namely a communication socket object that accepts client requests
  - 5: Create an object for ClientHandler Thread class and pass the communication socket, the object of BufferedReader and PrintWriter classes as the parameters
  - 6: Start the thread class by calling the start() method.
  - 7: In the ClientHandler Thread class, create a constructor of that class.
  - 8: Do the necessary communication until the client sends "Exit"
  - 9: Close the connection
- 

---

#### Algorithm 2: Algorithm of Client Side Socket Connection

---

- 1: Create a Socket object which takes IP address and port number as input.
  - 2: Create two objects of the PrintWriter and BufferedReader classes, which are used for sending and reading data, respectively.
  - 3: Create an object of ReadServer class which will read the response of server. Inside the run() method, do the necessary work.
  - 4: Client can send the data to the server side using the println() function, and Client can read any data using the readLine() function
  - 5: Client can continue its communication with the server until all the client sends "Exit"
  - 6: Close the connection
- 

**Termination Condition for Server and Client Side :** After starting a server, if **all the clients get disconnected** and the server types "**shutdown**", then the server will terminate. For the client, if client types "**exit**", then client side will terminate.

## 4 Implementation

### 4.1 Server Side

```
J Server.java U X
ChatApp > J Server.java > ClientHandler > ClientHandler(Socket, int, BufferedReader, PrintWriter)
1 package ChatApp;
2
3 import java.io.*;
4 import java.net.*;
5 import java.util.*;
6
7 public class Server {
8     private static final int PORT = 5800;
9     public static ArrayList<ClientHandler> clients = new ArrayList<>();
10    private static ServerSocket serverSocket;
11    private static int clientCounter = 1;
12
13    Run | Debug
14    public static void main(String[] args) {
15        try {
16            serverSocket = new ServerSocket(PORT);
17            System.out.println("Server started on port " + PORT);
18
19            PrintWriter out;
20            BufferedReader in;
21
22            WorkOfServer workOfServer = new WorkOfServer(serverSocket);
23            workOfServer.start();
24
25            while (true) {
26                try {
27                    Socket clientSocket = serverSocket.accept();
28                    System.out.println("New client connected: " + clientSocket + " (Client " + clientCounter + ")");
29                }
30            }
31        }
32    }
33}
```

Figure 4: Server-side code 1

```
24 while (true) {
25     try {
26         Socket clientSocket = serverSocket.accept();
27         System.out.println("New client connected: " + clientSocket + " (Client " + clientCounter + ")");
28
29         out = new PrintWriter(clientSocket.getOutputStream(), autoFlush:true);
30         in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
31
32         ClientHandler clientHandler = new ClientHandler(clientSocket, clientCounter++, in, out);
33         synchronized (clients) {
34             clients.add(clientHandler);
35         }
36         clientHandler.start();
37     } catch (SocketException e) {
38         break;
39     }
40 }
41
42 } catch (IOException e) {
43     e.printStackTrace();
44 } finally {
45     try {
46         if (serverSocket != null && !serverSocket.isClosed()) {
47             serverSocket.close();
48         }
49     } catch (IOException e) {
50         e.printStackTrace();
51     }
52 }
```

Figure 5: Server-side code 2

```

48     }
49     } catch (IOException e) {
50         e.printStackTrace();
51     }
52 }
53
54 public static void sendMessageToClient(int clientId, String message) {
55     synchronized (clients) {
56         for (ClientHandler client : clients) {
57             if (client.getClientId() == clientId) {
58                 client.sendMessage(message);
59                 System.out.println("Sent to Client " + clientId + ": " + message);
60                 return;
61             }
62         }
63         System.out.println("Client " + clientId + " not found.");
64     }
65 }
66
67 public static synchronized void removeClient(ClientHandler client) {
68     clients.remove(client);
69     System.out.println("Client " + client.getClientId() + " disconnected. Active clients: " + clients.size());
70 }
71
72 }

```

Figure 6: Server-side code 3

```

72 }
73
74 class ClientHandler extends Thread {
75     private Socket socket;
76     private PrintWriter out;
77     private BufferedReader in;
78     private final int clientId;
79     private static final String endDetect = "<EOM>";
80
81     public ClientHandler(Socket socket, int clientId, BufferedReader in, PrintWriter out) {
82         this.socket = socket;
83         this.clientId = clientId;
84         this.in = in;
85         this.out = out;
86     }
87
88     public int getClientId() {
89         return clientId;
90     }
91
92     public void sendMessage(String message) {
93         out.println(message);
94     }
95
96     @Override
97     public void run() {
98         try {
99

```

Figure 7: Server-side code 4

```

@Override
public void run() {
    try {
        StringBuilder messageBuilder = new StringBuilder();
        String line;

        while ((line = in.readLine()) != null) {
            if (line.equalsIgnoreCase("EXIT")) {
                break;
            }
            if (line.equals(endDetect)) {
                if (messageBuilder.length() > 0) {
                    String message = messageBuilder.toString();
                    System.out.println("Received from Client " + clientId + ": " + message);
                    // Echo back to the client
                    out.println("Server: Received message from Client " + clientId + " successfully!!");
                    messageBuilder.setLength(newLength:0); // Clear builder for next message
                }
            } else {
                messageBuilder.append(line).append("\n");
            }
        }
    } catch (IOException e) {
        System.out.println("Error handling Client " + clientId + ": " + e.getMessage());
    } finally {
        try {

```

Figure 8: Server-side code 5

```

120         System.out.println("Error handling Client " + clientId + ": " + e.getMessage());
121     } finally {
122         try {
123             out.close();
124             in.close();
125             socket.close();
126             Server.removeClient(this);
127         } catch (IOException e) {
128             e.printStackTrace();
129         }
130     }
131 }
132 }
133
134 class WorkOfServer extends Thread {
135
136     ServerSocket ss;
137     Scanner scanner = null;
138
139     WorkOfServer(ServerSocket ss) {
140         this.ss = ss;
141     }
142
143     @Override
144     public void run() {
145         scanner = new Scanner(System.in);

```

Figure 9: Server-side code 6

```

@Override
public void run() {
    scanner = new Scanner(System.in);
    try {
        while (true) {
            String input = scanner.nextLine();
            if (input.equalsIgnoreCase("SHUTDOWN") && Server.clients.isEmpty()) {
                System.out.println(x:"Shutting down server...");
                try {
                    ss.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                break;
            } else if (input.toUpperCase().startsWith(prefix:"SEND ")) {
                String[] parts = input.split(regex:" ", limit:3);
                if (parts.length < 3) {
                    System.out.println(x:"Invalid command. Use: SEND <client_id> <message>");
                    continue;
                }
                try {
                    int clientId = Integer.parseInt(parts[1]);
                    String message = parts[2];
                    Server.sendMessageToClient(clientId, "Server: " + message);
                } catch (NumberFormatException e) {
                    System.out.println(x:"Invalid client ID. Use: SEND <client_id> <message>");
                }
            }
        }
    }
}

```

Figure 10: Server-side code 7

```

163         try {
164             int clientId = Integer.parseInt(parts[1]);
165             String message = parts[2];
166             Server.sendMessageToClient(clientId, "Server: " + message);
167         } catch (NumberFormatException e) {
168             System.out.println(x:"Invalid client ID. Use: SEND <client_id> <message>");
169         }
170     } else if (!Server.clients.isEmpty()) {
171         System.out.println(x:"Cannot shutdown: Clients are still connected.");
172     } else {
173         System.out.println(x:"Invalid command. Use 'SEND <client_id> <message>' or 'SHUTDOWN'");
174     }
175 }
176 } finally {
177     scanner.close();
178 }
179 }
180
181
182
183 }

```

Figure 11: Server-side code 8

## 4.2 Client Side



```

J Clientjava U X
ChatApp > J Clientjava > ReadServer > run()
1 package ChatApp;
2
3 import java.io.*;
4 import java.net.*;
5
6 public class Client {
7     private static final String SERVER_ADDRESS = "localhost";//10.33.28.18
8     private static final int SERVER_PORT = 5000;
9     private static final String endDetect = "<EOM>";
10
11     Run | Debug
12     public static void main(String[] args) {
13         Socket socket = null;
14         PrintWriter out = null;
15         BufferedReader in = null;
16         BufferedReader read = new BufferedReader(new InputStreamReader(System.in));
17
18         try {
19             socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
20             System.out.println("Connected to server at " + SERVER_ADDRESS + ":" + SERVER_PORT);
21             System.out.println(x:"Enter messages (type 'SEND' to send message, 'EXIT' to quit):");
22
23             out = new PrintWriter(socket.getOutputStream(), autoFlush:true);
24             in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
25
26             ReadServer rs = new ReadServer(in);
27             rs.start();
28
29             StringBuilder messageBuffer = new StringBuilder();
30             String userInput;

```

Figure 12: Client-side code 1

```

27
28     StringBuilder messageBuffer = new StringBuilder();
29     String userInput;
30
31     while ((userInput = read.readLine()) != null) {
32         if (userInput.equalsIgnoreCase(anotherString:"EXIT")) {
33             out.println(x:"EXIT");
34             break;
35         } else if (userInput.equalsIgnoreCase(anotherString:"SEND")) {
36             if (messageBuffer.length() > 0) {
37                 out.println(messageBuffer.toString());
38                 out.println(endDetect);
39                 messageBuffer.setLength(newLength:0);
40             }
41         } else {
42             messageBuffer.append(userInput).append(str:"\n");
43         }
44     }
45 } catch (IOException e) {
46     System.out.println("Error connecting to server: " + e.getMessage());
47 } finally {
48     try {
49         if (out != null) out.close();
50         if (in != null) in.close();
51         if (socket != null) socket.close();
52         read.close();
53     } catch (IOException e) {

```

Figure 13: Client-side code 2

```

51     public static void main(String[] args) {
52         try {
53             if (socket != null) socket.close();
54             read.close();
55         } catch (IOException e) {
56             e.printStackTrace();
57         }
58     }
59 }
60
61 class ReadServer extends Thread {
62     BufferedReader str;
63
64     ReadServer(BufferedReader bf) {
65         this.str = bf;
66     }
67
68     @Override
69     public void run() {
70         try {
71             String response;
72             while ((response = str.readLine()) != null) {
73                 //System.out.println("hello");
74                 System.out.println(response);
75             }
76         } catch (IOException e) {
77             System.out.println(x:"Server disconnected.");

```

Figure 14: Client-side code 3

```

66     }
67
68     @Override
69     public void run() {
70         try {
71             String response;
72             while ((response = str.readLine()) != null) {
73                 //System.out.println("hello");
74                 System.out.println(response);
75             }
76         } catch (IOException e) {
77             System.out.println(x:"Server disconnected.");
78         }
79     }
80 }

```

Figure 15: Client-side code 4

## 5 Result Analysis

1.Starting the server.

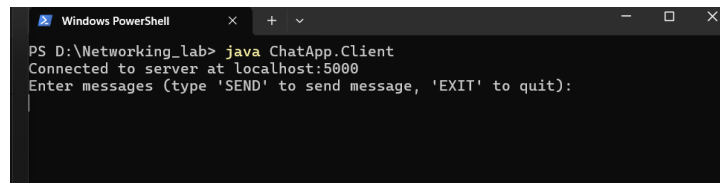
```

PS D:\Networking_lab> java ChatApp.Server
Server started on port 5000

```

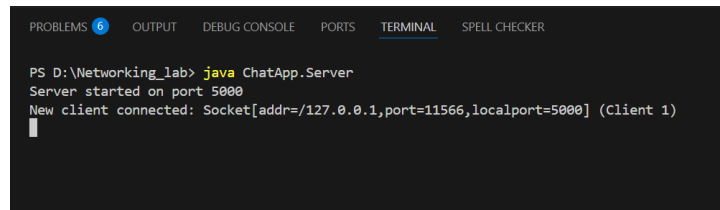
Figure 16: Waiting for clients to connect.

2.Client 1 requested to server.



```
PS D:\Networking_lab> java ChatApp.Client
Connected to server at localhost:5000
Enter messages (type 'SEND' to send message, 'EXIT' to quit):
```

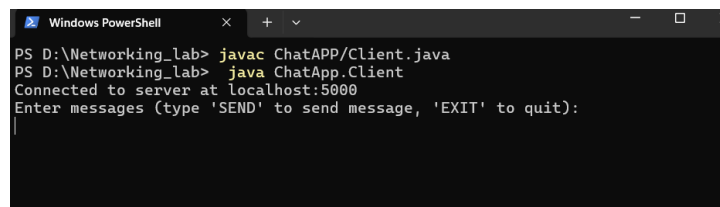
Figure 17: Client 1.



```
PS D:\Networking_lab> java ChatApp.Server
Server started on port 5000
New client connected: Socket[addr=/127.0.0.1,port=11566,localport=5000] (Client 1)
```

Figure 18: Server.

3. Client 2 requested to server.



```
PS D:\Networking_lab> java ChatApp.Client
Connected to server at localhost:5000
Enter messages (type 'SEND' to send message, 'EXIT' to quit):
```

Figure 19: Client 2.



```
PS D:\Networking_lab> java ChatApp.Server
Server started on port 5000
New client connected: Socket[addr=/127.0.0.1,port=11566,localport=5000] (Client 1)
New client connected: Socket[addr=/127.0.0.1,port=11625,localport=5000] (Client 2)
```

Figure 20: Server accepted request.

4. Client 1 wrote the message in the terminal and added "send" to send to server. Server received the message and sent acknowledgment to client.

```
Windows PowerShell
PS D:\Networking_lab> java ChatApp.Client
Connected to server at localhost:5000
Enter messages (type 'SEND' to send message, 'EXIT' to quit):
Hello,server.
How are you?
send
Server: Received message from Client 1 successfully!!
```

Figure 21: Client 1.

```
PROBLEMS 6 OUTPUT DEBUG CONSOLE PORTS TERMINAL SPELL CHECKER
PS D:\Networking_lab> java ChatApp.Server
Server started on port 5000
New client connected: Socket[addr=/127.0.0.1,port=11566,localport=5000] (Client 1)
New client connected: Socket[addr=/127.0.0.1,port=11625,localport=5000] (Client 2)
Received from Client 1:
Hello,server.
How are you?
```

Figure 22: Server.

5.Then the server replied to client 1 and got acknowledgment (<send> <client id> <msg>).

```
PROBLEMS 6 OUTPUT DEBUG CONSOLE PORTS TERMINAL SPELL CHECKER
PS D:\Networking_lab> java ChatApp.Server
Server started on port 5000
New client connected: Socket[addr=/127.0.0.1,port=11566,localport=5000] (Client 1)
New client connected: Socket[addr=/127.0.0.1,port=11625,localport=5000] (Client 2)
Received from Client 1:
Hello,server.
How are you?

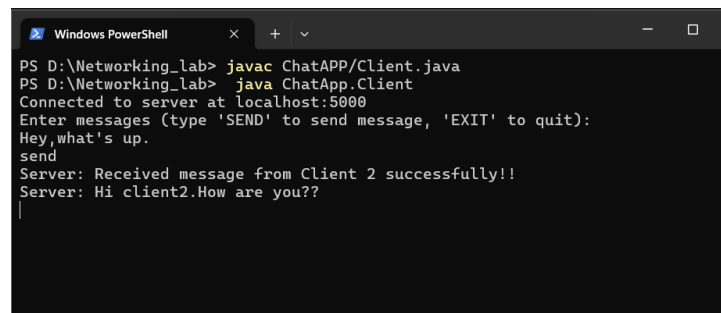
send 1 Hi,I am good.what about you?
Sent to Client 1: Server: Hi,I am good.what about you?
```

Figure 23: Server.

```
Windows PowerShell
PS D:\Networking_lab> java ChatApp.Client
Connected to server at localhost:5000
Enter messages (type 'SEND' to send message, 'EXIT' to quit):
Hello,server.
How are you?
send
Server: Received message from Client 1 successfully!!
Server: Hi,I am good.what about you?
```

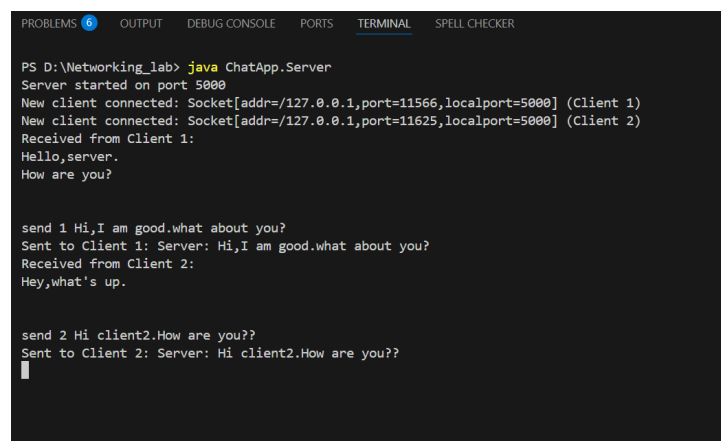
Figure 24: Client 1.

6. Similarly, client 2 requested to connect and sent message and got a reply.



```
Windows PowerShell
PS D:\Networking_lab> javac ChatAPP/Client.java
PS D:\Networking_lab> java ChatApp.Client
Connected to server at localhost:5000
Enter messages (type 'SEND' to send message, 'EXIT' to quit):
Hey,what's up.
send
Server: Received message from Client 2 successfully!!
Server: Hi client2.How are you??
```

Figure 25: client 2.



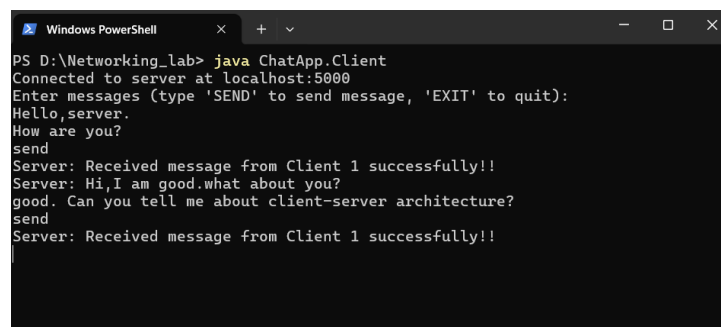
```
PROBLEMS 6 OUTPUT DEBUG CONSOLE PORTS TERMINAL SPELL CHECKER
PS D:\Networking_lab> java ChatApp.Server
Server started on port 5000
New client connected: Socket[addr=/127.0.0.1,port=11566,localport=5000] (Client 1)
New client connected: Socket[addr=/127.0.0.1,port=11625,localport=5000] (Client 2)
Received from Client 1:
Hello,server.
How are you?

send 1 Hi,I am good.what about you?
Sent to Client 1: Server: Hi,I am good.what about you?
Received from Client 2:
Hey,what's up.

send 2 Hi client2.How are you??
Sent to Client 2: Server: Hi client2.How are you??
```

Figure 26: Server.

7. When multiple clients send messages simultaneously, the server chose to respond to a specific client and was not restricted to reply to clients.



```
Windows PowerShell
PS D:\Networking_lab> java ChatApp.Client
Connected to server at localhost:5000
Enter messages (type 'SEND' to send message, 'EXIT' to quit):
Hello,server.
How are you?
send
Server: Received message from Client 1 successfully!!
Server: Hi,I am good.what about you?
good. Can you tell me about client-server architecture?
send
Server: Received message from Client 1 successfully!!
```

Figure 27: client 1.

```
Windows PowerShell
PS D:\Networking_lab> javac ChatAPP/Client.java
PS D:\Networking_lab> java ChatApp.Client
Connected to server at localhost:5000
Enter messages (type 'SEND' to send message, 'EXIT' to quit):
Hey,what's up.
send
Server: Received message from Client 2 successfully!!
Server: Hi client2.How are you??

I am fine. where do you live?
send
Server: Received message from Client 2 successfully!!
```

Figure 28: Client 2.

```
PROBLEMS 6 OUTPUT DEBUG CONSOLE PORTS TERMINAL SPELL CHECKER
PS D:\Networking_lab> java ChatApp.Server
Server started on port 5000
New client connected: Socket[addr=/127.0.0.1,port=11566,localport=5000] (Client 1)
New client connected: Socket[addr=/127.0.0.1,port=11625,localport=5000] (Client 2)
Received from Client 1:
Hello,server.
How are you?

send 1 Hi,I am good.what about you?
Sent to Client 1: Server: Hi,I am good.what about you?
Received from Client 2:
Hey,what's up.

send 2 Hi client2.How are you??
Sent to Client 2: Server: Hi client2.How are you??
Received from Client 2:

I am fine. where do you live?

Received from Client 1:
good. Can you tell me about client-server architecture?

send 2 Dhaka,Bangladesh.
```

Figure 29: Server.

8. If clients sent "exit" ,they became disconnected from server.Then server got acknowledgement and displayed the active client count.

```
Windows PowerShell
PS D:\Networking_lab> java ChatApp.Client
Connected to server at localhost:5000
Enter messages (type 'SEND' to send message, 'EXIT' to quit):
Hello,server.
How are you?
send
Server: Received message from Client 1 successfully!!
Server: Hi,I am good.what about you?
good. Can you tell me about client-server architecture?
send
Server: Received message from Client 1 successfully!!
exit
Server disconnected.
PS D:\Networking_lab> |
```

Figure 30: client 1.

```
PROBLEMS 6 OUTPUT DEBUG CONSOLE PORTS TERMINAL SPELL CHECKER

PS D:\Networking_lab> java ChatApp.Server
Server started on port 5000
New client connected: Socket[addr=/127.0.0.1,port=11566,localport=5000] (Client 1)
New client connected: Socket[addr=/127.0.0.1,port=11625,localport=5000] (Client 2)
Received from Client 1:
Hello,server.
How are you?

send 1 Hi,I am good.what about you?
Sent to Client 1: Server: Hi,I am good.what about you?
Received from Client 2:
Hey,what's up.

send 2 Hi client2.How are you??
Sent to Client 2: Server: Hi client2.How are you??
Received from Client 2:

I am fine. where do you live?

Received from Client 1:
good. Can you tell me about client-server architecture?

send 2 Dhaka,Bangladesh.
Sent to Client 2: Server: Dhaka,Bangladesh.
Client 1 disconnected. Active clients: 1
```

Figure 31: Server.

9.The server could not terminate as long as there were active clients connected.

```
PROBLEMS 6 OUTPUT DEBUG CONSOLE PORTS TERMINAL SPELL CHECKER

PS D:\Networking_lab> java ChatApp.Server
Server started on port 5000
New client connected: Socket[addr=/127.0.0.1,port=11566,localport=5000] (Client 1)
New client connected: Socket[addr=/127.0.0.1,port=11625,localport=5000] (Client 2)
Received from Client 1:
Hello,server.
How are you?

send 1 Hi,I am good.what about you?
Sent to Client 1: Server: Hi,I am good.what about you?
Received from Client 2:
Hey,what's up.

send 2 Hi client2.How are you??
Sent to Client 2: Server: Hi client2.How are you??
Received from Client 2:

I am fine. where do you live?

Received from Client 1:
good. Can you tell me about client-server architecture?

send 2 Dhaka,Bangladesh.
Sent to Client 2: Server: Dhaka,Bangladesh.
Client 1 disconnected. Active clients: 1
shutdown
Cannot shutdown: Clients are still connected.
```

Figure 32: Server.

10. To close the connection, the server entered 'shutdown' after ensuring that all clients were disconnected.

```

Received from Client 1:
good. Can you tell me about client-server architecture?

send 2 Dhaka,Bangladesh.
Sent to Client 2: Server: Dhaka,Bangladesh.
Client 1 disconnected. Active clients: 1
shutdown
Cannot shutdown: Clients are still connected.
Client 2 disconnected. Active clients: 0
shutdown
Shutting down server...
PS D:\Networking_lab>

```

Figure 33: Server.

## 6 Discussion

In basic socket programming, it **cannot handle multiple client requests at the same time**. A server can only provide service to the client that connects first. Other clients cannot connect with the server until the first disconnects. To solve this problem, the server opens different threads for each client, and every client communicates with the server using its **own thread**. One client's processing doesn't block others.

In basic socket programming, **resources are underutilized**. On multi-core systems, it typically utilizes only a single core, wasting available computing resources. Also, a slow client that takes a long time to send or receive data will block all other clients (**head-of-line blocking**). But in multi-threading, all these problems are solved as it handles each client separately by opening a thread for each client. This is what multi-threaded socket programming does.

We learned how to send multiple lines to the server at a time, how to handle each client separately, how to send messages to a particular client, and how to handle the termination condition for both the client and server sides.

While implementing multi-line message sending from clients to the server, the server was unable to respond to the clients properly. When multiple clients sent messages to the server, it only responded to the last client. After introducing the multi-line feature on the client side, the server had to send a response twice to the client. Although the clients sent "exit" to the server, they remained connected, resulting in an infinite loop. Additionally, the server could shut down at any time without responding to the clients.