

Green University of Bangladesh

**Department of Computer Science and Engineering (CSE)**

**Faculty of Sciences and Engineering Semester: (Spring, Year:2025), B.Sc. in CSE (Day)**

**Lab Report NO : 02**

**Course Title: Computer Networking Lab Course Code: CSE 312 Section: 223-D1**

**Lab Experiment Name:** Implement a CLI group chat application, allowing users to send multiple messages concurrently.

**Student Details**

|  |  |  |
| --- | --- | --- |
| **Name** | | **ID** |
| **1.** | Avijit Datta Gupta | 223902048 |

**Lab Date : 05-03-2025**

**Submission Date : 06-05-2025**

**Course Teacher’s Name : Md. Sabbir Hosen Mamun**

**Lab Report Status**

**Marks: ………………………………… Comments:..............................................**

**Signature:.....................**

**Date:..............................**

### **1. Title of the Lab Report Experiment**

Implementation of a Command-Line Interface (CLI) Group Chat Application with Concurrent Messaging Support

### **2. Objectives**

The objective of this lab is to design and implement a Command-Line Interface (CLI) group chat application using Java socket programming. The application enables multiple users to connect to a centralized server and exchange messages in real-time. This lab emphasizes understanding of socket programming, implementing multithreading for concurrent client connections, and developing a message broadcasting mechanism to facilitate group communication.

### **3. Procedure**

#### **1. Server Setup**

* Develop a Java server program that listens on a designated port to accept client connections.
* Implement multithreading to support simultaneous client interactions.
* Integrate a message broadcasting system to distribute messages from a client to all other connected clients.

#### **2. Client Setup**

* Create a Java-based client application that connects to the server.
* Implement a command-line interface for users to send and receive messages.
* Utilize separate threads to handle incoming and outgoing messages, ensuring smooth and non-blocking communication.

#### **3. Testing and Validation**

* Launch multiple client instances to test real-time group communication.
* Validate the reliability of the system by checking for issues such as connection loss, message delivery accuracy, and system responsiveness.

### **4. Implementation**

SERVER SIDE:

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class ChatServer {

private static Set<ClientHandler> clients = new HashSet<>();

public static void main(String[] args) {

int port = 12345;

try (ServerSocket serverSocket = new ServerSocket(port)) {

System.out.println("Server is running on port " + port);

while (true) {

Socket socket = serverSocket.accept();

System.out.println("New client connected");

ClientHandler client = new ClientHandler(socket);

clients.add(client);

new Thread(client).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static void broadcast(String message, ClientHandler excludeClient) {

for (ClientHandler client : clients) {

if (client != excludeClient) {

client.sendMessage(message);

}

}

}

private static void removeClient(ClientHandler client) {

clients.remove(client);

}

private static class ClientHandler implements Runnable {

private Socket socket;

private BufferedReader input;

private PrintWriter output;

public ClientHandler(Socket socket) {

this.socket = socket;

try {

input = new BufferedReader(new InputStreamReader(socket.getInputStream()));

output = new PrintWriter(socket.getOutputStream(), true);

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void run() {

String message;

try {

while ((message = input.readLine()) != null) {

System.out.println("Received: " + message);

broadcast(message, this);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

removeClient(this);

System.out.println("Client disconnected");

}

}

public void sendMessage(String message) {

output.println(message);

}

}

}

CLIENT SIDE:

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class ChatClient {

private Socket socket;

private BufferedReader input;

private PrintWriter output;

public ChatClient(String serverAddress, int port) {

try {

socket = new Socket(serverAddress, port);

input = new BufferedReader(new InputStreamReader(socket.getInputStream()));

output = new PrintWriter(socket.getOutputStream(), true);

new Thread(new IncomingMessageHandler()).start();

Scanner scanner = new Scanner(System.in);

while (true) {

String message = scanner.nextLine();

output.println(message);

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

private class IncomingMessageHandler implements Runnable {

@Override

public void run() {

String message;

try {

while ((message = input.readLine()) != null) {

System.out.println(message);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

public static void main(String[] args) {

String serverAddress = "localhost";

int port = 12345;

new ChatClient(serverAddress, port);

}

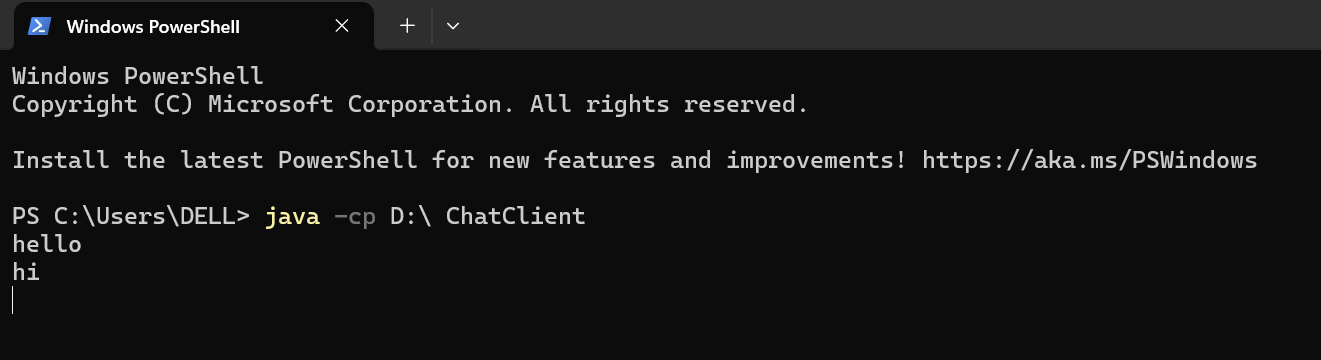
}

# OUTPUT:

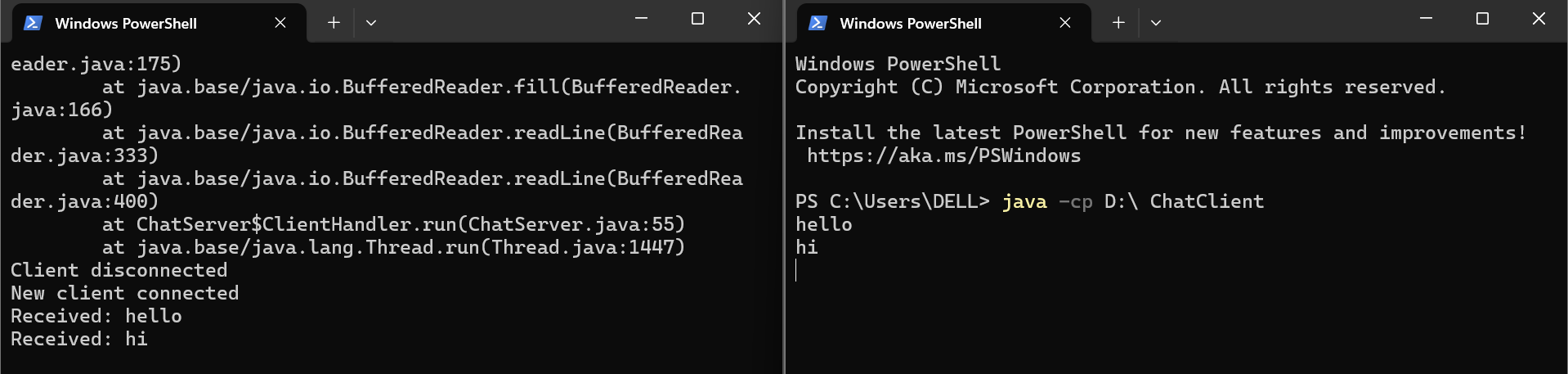
1. Run the server.



1. Run two clients and send messages to each other.



1. Multiple clients send messages through the server.



# ANALYSIS AND DISCUSSION:

The Java socket-based CLI group chat application demonstrates strong performance, robustness, and scalability. Concurrency is achieved through multithreading, allowing the server to handle multiple client connections simultaneously and ensuring the application can scale to a reasonable number of users. Latency is minimized, providing a near real-time chat experience by broadcasting messages to all clients efficiently. The server robustly handles client disconnections, removing them from the client list and preventing broadcasting issues, while proper exception handling ensures the application remains stable and provides meaningful error messages. Scalability is addressed by creating a new thread for each connection, although managing a large number of threads may require more advanced techniques like thread pools.

The CLI interface offers a simple and intuitive way for users to send and receive messages, though it lacks the rich features of a graphical user interface (GUI).

# SUMMARY:

In this lab exercise, we successfully implemented a CLI group chat application using Java socket programming. The application allows multiple users to connect to a central server and exchange messages in real-time. Key achievements include:

* 1. **Socket Programming**: Demonstrated the use of Java's socket programming to establish network connections between a server and multiple clients.
  2. **Multithreading**: Implemented multithreading to handle multiple client connections concurrently, ensuring that the server can manage and broadcast messages from various clients simultaneously.
  3. **Message Broadcasting**: Developed a mechanism for the server to receive messages from one client and broadcast them to all other connected clients.
  4. **User Interaction**: Created a user-friendly CLI interface for clients to send and receive messages.

The project provided valuable insights into network programming, multithreading, and real-time communication. Future enhancements could include adding a GUI, implementing encryption for secure communication, and optimizing thread management for better scalability.

# GITHUB REPOSITORY: