**INTERNATE CHAT SYSTEM**

**Names Registration number**

Kialyulo Mark C026-01-0897/2022

George Kihoro C026-01-0930/2022

Alvin Kiptoo C026-01-0958/2022

Morgan Nganyi C026-01-0954/2022

Mark Kagiri C026-01-0901/2022

Jerry Odhiambo C026-01-1070/2021

# Software Engineering Requirements

## Functional Requirements:

### Connection Handling:

* The system shall allow multiple remote clients to connect to a central server.
* The system shall enable clients to join and leave the chat seamlessly.

### Messaging:

* The system shall send messages typed by a client to the server and broadcast them to all other connected clients.
* The system shall include the sender's chosen nickname in each message.
* The system shall buffer incoming messages and display them once the user finishes typing.

### Error Handling:

* The system shall handle client crashes or blocked connections gracefully, ensuring minimal impact on other users.
* The system shall implement proper exception handling to maintain robustness.

## Non-functional Requirements:

### Performance:

* The system shall ensure efficient message delivery with minimal latency.
* The system shall be scalable to handle multiple simultaneous connections.

### Usability:

* The system shall provide a simple text-mode interface for clients.
* The system shall offer clear user prompts and feedback.

# Algorithm design

## Client Side:

* Establish a TCP connection to the server.
* Allow the user to input their nickname.
* Continuously read user input and send messages to the server.
* Display messages received from the server.

## Server Side:

* Listen for incoming connections from clients.
* Maintain a list of connected clients.
* Broadcast received messages to all clients except the sender.
* Handle client disconnections.

# Java Implementation

## Server:

* We implemented a Server class that listens for client connections using Server Socket.
* We used Socket for each client connection to manage communication.
* We Maintained a list of connected clients using threads for each client.

**Here is the code for server side:**

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class ChatServer {

private static final int PORT = 22223;

private static Set<ClientHandler> clientHandlers = Collections.synchronizedSet(new HashSet<>());

public static void main(String[] args) {

System.out.println("Chat server started...");

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

while (true) {

Socket clientSocket = serverSocket.accept();

ClientHandler clientHandler = new ClientHandler(clientSocket);

clientHandlers.add(clientHandler);

new Thread(clientHandler).start();

System.out.println("New client connected: " + clientSocket.getInetAddress().getHostAddress());

}

} catch (IOException e) {

System.out.println("Server error: " + e.getMessage());

}

}

public static void broadcastMessage(String message, ClientHandler excludeClient) {

synchronized (clientHandlers) {

for (ClientHandler clientHandler : clientHandlers) {

if (clientHandler != excludeClient) {

clientHandler.sendMessage(message);

}

}

}

}

public static void removeClient(ClientHandler clientHandler) {

clientHandlers.remove(clientHandler);

System.out.println("Client disconnected: " + clientHandler.getNickname());

}

}

class ClientHandler implements Runnable {

private Socket socket;

private PrintWriter out;

private BufferedReader in;

private String nickname;

public ClientHandler(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

try {

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

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

out.println("Enter your nickname:");

nickname = in.readLine();

System.out.println(nickname + " has joined the chat.");

ChatServer.broadcastMessage(nickname + " has joined the chat.", this);

String message;

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

if (message.equalsIgnoreCase("/exit")) {

break;

}

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

ChatServer.broadcastMessage(nickname + ": " + message, this);

}

} catch (IOException e) {

System.out.println("Error in client handler: " + e.getMessage());

} finally {

try {

socket.close();

} catch (IOException e) {

System.out.println("Error closing socket: " + e.getMessage());

}

ChatServer.broadcastMessage(nickname + " has left the chat.", this);

ChatServer.removeClient(this);

}

}

public void sendMessage(String message) {

out.println(message);

}

public String getNickname() {

return nickname;

}

}

## Client:

* We Implemented a Client class that connects to the server using Socket.
* We Used separate threads for reading messages from the server and sending user messages.

**Here is the code for Client side**

import java.io.\*;

import java.net.\*;

import java.util.Scanner;

public class chatclient {

private static final String SERVER\_ADDRESS = "localhost"; // Replace with server IP address if not running locally

private static final int SERVER\_PORT = 22223;

public static void main(String[] args) {

try (Socket socket = new Socket(SERVER\_ADDRESS, SERVER\_PORT);

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

BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

Scanner scanner = new Scanner(System.in);

System.out.println("Connected to the chat server");

System.out.println(in.readLine()); // Read the prompt for nickname

String nickname = scanner.nextLine();

out.println(nickname); // Send nickname to the server

// Thread to read messages from the server

new Thread(() -> {

String serverMessage;

try {

while ((serverMessage = in.readLine()) != null) {

System.out.println(serverMessage);

}

} catch (IOException e) {

System.out.println("Error reading from server: " + e.getMessage());

}

}).start();

// Main thread to send user messages to the server

while (true) {

String userMessage = scanner.nextLine();

out.println(userMessage);

if (userMessage.equalsIgnoreCase("/exit")) {

break;

}

}

} catch (IOException e) {

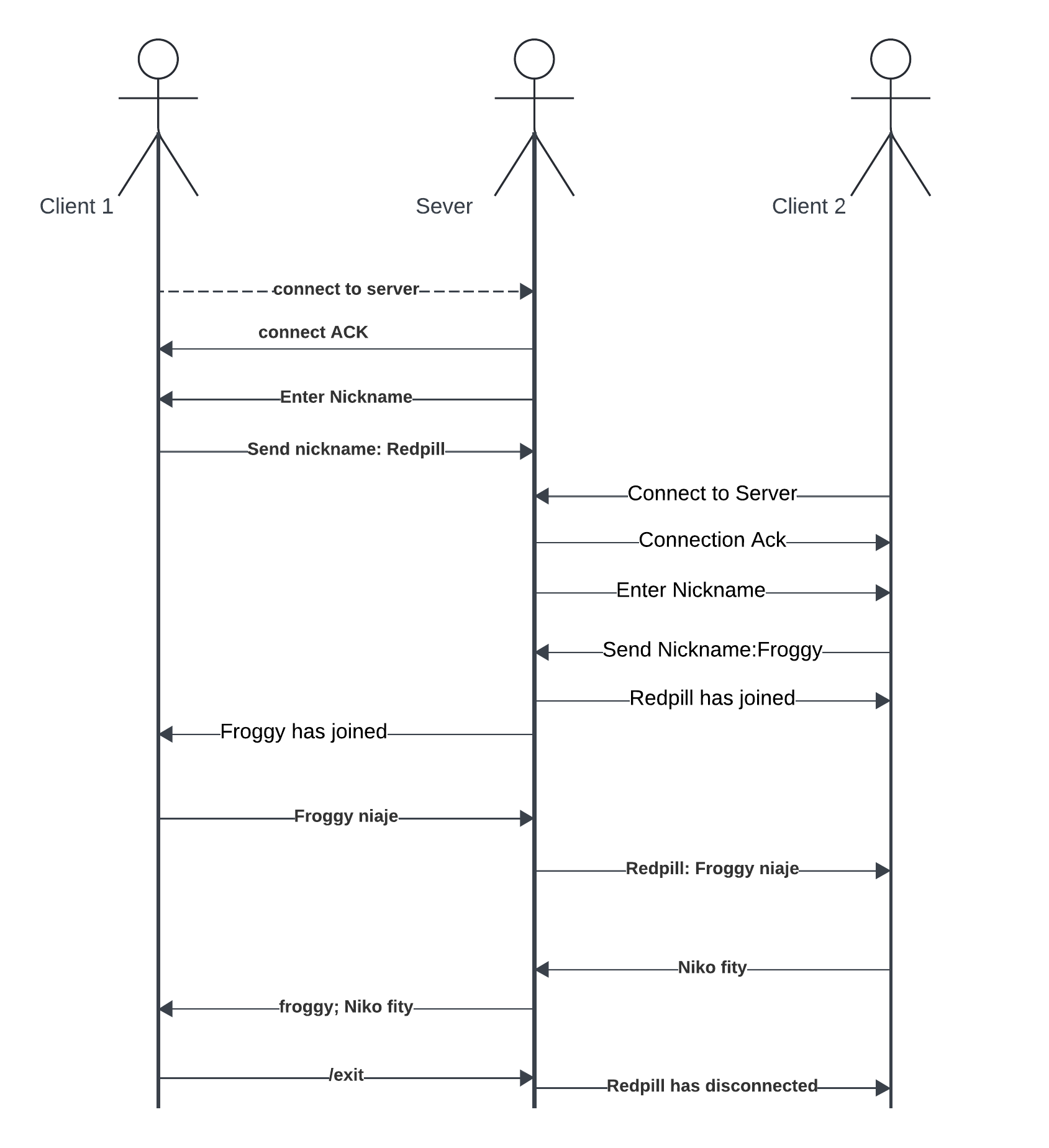
System.out.println("Client error: " + e.getMessage());

}

}

}

**Here is a diagram representing the system**

**[](https://lucid.app/lucidchart/225802d6-a854-48d0-b521-ee0ba0b51dbc/edit?crop=content&page=0&signature=8198f8fa03abafcc65e555657ba610052ec2d054b62348f7cc9da0a4543102f1)**

**possible bugs and unexpected behaviors**

Clients unable connect if the server IP address or port is incorrect, or if there are network issues.

Connection Issues when the Server is not listening

Delays in message delivery due to network latency or server processing delays.

Client Reconnection Issues especially after disconnecting from the server

**Alternative Implementation**

The UDP protocol can also be used to implement the chat system.

The following are some of UDP consideration:

**Advantages:**

* Lower latency due to connectionless nature.
* Suitable for real-time applications like voice chat.

**Disadvantages:**

* Unreliable delivery—messages can be lost or arrive out of order.
* No built-in congestion control.

**Comparison between UDP and TCP**

TCP: Reliable, ordered, and ensures all data is transmitted.

UDP: Faster but less reliable and ordered.