**UNIVERSITY: DEDAN KIMATHI UNIVERSITY OF TECHNOLOGY**

**COURSE: COMPUTER SCIENCE**

**UNIT: DISTRIBUTED SYSTEMS**

**UNIT CODE: 3103**

**GROUP MEMBERS:**

|  |  |
| --- | --- |
| **NAME** | **REGISTRATION NUMBER** |
| **ELIZABETH MUTHONI** | **C026-01-0924/2022** |
| **GRIFFINS GITARI** | **C026-01-0914/2022** |
| **LINDSAY NJOKI** | **C026-01-0926/2022** |
| **EVANS MUNGAI** | **C026-01-0908/2022** |
| **TERESA WAIRIMU** | **C026-01-0925/2022** |
| **DANTE KADAGI** | **C026-01-0920/2022** |
| **ANERICO KAKAI** | **C026-01-0969/2022** |

**INTERNET CHAT SYSTEM**

**1. Software Engineering Requirements**

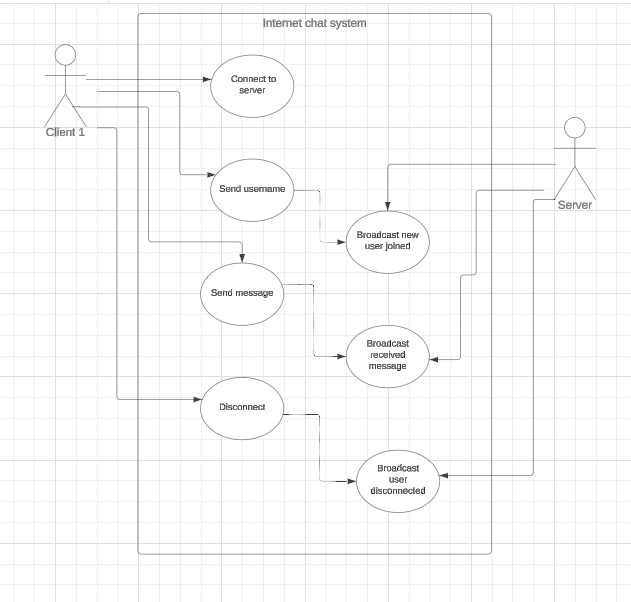
**Functional Requirements:**

1. Clients must be able to join and leave the chat at any time.
2. Messages sent by a client should be broadcast to all connected clients.
3. Clients must choose a unique nickname upon joining.
4. Clients must specify the server address to connect.

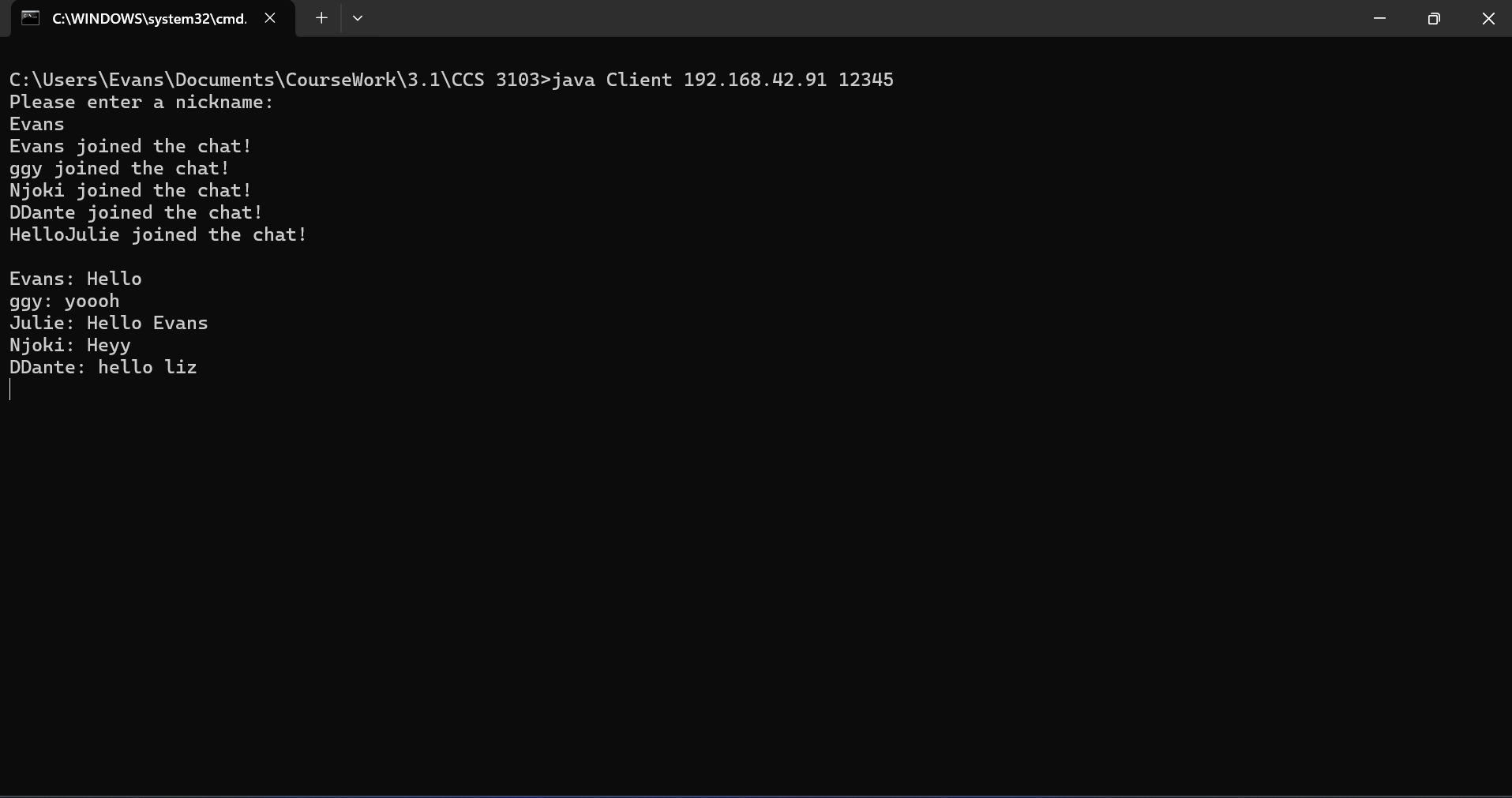
**Non-Functional Requirements:**

1. The system must handle client crashes and blocked connections:
   * + - If a client crashes or experiences a network failure, the server will detect the disconnection and remove the client from the list of connected clients.
       - If a client's connection is blocked for a period of time, the server will assume the client has disconnected and remove it from the list of connected clients.
2. The system should preferably run under Linux.
3. The client runs in a single text-mode window.

**Use case diagram**



**User Interface Design**



**2. Algorithm Design**

**System Architecture:**

Client-Server Model: A central server to which multiple clients connect. The server will be responsible for managing client connections, receiving messages from clients, and broadcasting those messages to all connected clients.

Interactions:

1.Client Connection:

Clients connect to the server by establishing a TCP connection with the server specifying the server address and port.

2.Message Sending:

A client sends a message to the server containing their nicknames.

The server broadcasts the message to all connected clients informing that a new user has joined the chat.

The client sends a message to the server.

The server receives the message and broadcasts it to all connected clients including the original sender.

Each client receives the message displaying it on their chat window along with the sender’s username.

3.Client Disconnection:

Clients can disconnect at any time they want to leave the chat.

The server notifies all remaining clients of the disconnection.

3. Implementation

Server Code:

Server Class: Handles client connections, message broadcasting, and maintains client list.

Client handler class: Manages communication with individual clients, including reading incoming messages and sending broadcasts.

Client Code:

Client Class: Manages connection to the server, sending messages, and receiving broadcasts.

User Interface: Single text-mode window for message input and display.

**3. Testing**

Unit Tests: Test individual components , server, client handler, for robustness and error handling.

Integration Tests: Test the complete system by connecting multiple clients and verifying message broadcast.

Stress Tests: Test the system's behavior under a high load of messages and clients.

**Report of possible bugs and/or unexpected behaviors.**

ConcurrentModificationException:

If the broadcast method is called while iterating over the connections list, it might throw a ConcurrentModificationException because another thread could modify the list simultaneously.

Resource Leak:

If an exception occurs while setting up the PrintWriter or BufferedReader, the Socket might not be closed properly, leading to resource leaks.

Unhandled Exceptions:

The shutdown method should handle exceptions more gracefully and ensure that all resources are closed properly.

Exceptions within the run method of ConnectionHandler should provide more details for debugging purposes.

Nickname Handling:

If two clients choose the same nickname, it could lead to confusion. Consider checking for unique nicknames.

Server Shutdown:

When the server is shutting down, it should notify connected clients about the shutdown.

**4. Possible alternative implementation with UDP**

Differences:

UDP is connectionless, there is no persistent connection between client and server.

TCP ensures message delivery and order, whereas UDP does not guarantee delivery or order.

UDP might be suitable for real-time applications where speed is crucial, and occasional message loss is acceptable.

Critical Comparison:

TCP is more reliable due to its built-in error checking and retransmission. UDP is faster but less reliable.

TCP has more overhead due to connection management and error checking. UDP has lower overhead, making it faster.

TCP simplifies development with its built-in mechanisms. UDP requires additional handling for reliability and order.

**How to run the code:**

1. Compile the server code: **javac Server.java**
2. Run the server code: **java Server.java**
3. Compile the client code: **javac Client.java**
4. Run the client code : **java Client (ip address) (port).** Replace (ip address) with the server’s ip address and (port) with the port number
5. Enter your nickname when the system prompts: “Please enter a nickname:”
6. If successful, the program will inform the user that he/she has joined the chat.

**Appendix**

**Server Code**

import java**.**io**.**IOException;

import java**.**net**.**ServerSocket;

import java**.**net**.**Socket;

import java**.**io**.**PrintWriter;

import java**.**io**.**BufferedReader;

import java**.**io**.**InputStreamReader;

import java**.**util**.**ArrayList;

import java**.**util**.**concurrent**.**ExecutorService;

import java**.**util**.**concurrent**.**Executors;

**public** **class** Server **implements** Runnable {

**private** ArrayList<ConnectionHandler> *connections*;

**private** ServerSocket *server*;

**private** **boolean** *done*;

**private** ExecutorService *pool*;

**public** Server() {

*connections* = new ArrayList<>();

*done* = false;

    }

    @**Override**

**public** **void** run(){

        try {

*server* = new ServerSocket(12345);

            System.*out*.println("Server is running on port 12345"); *// Print that the server is running*

*pool* = Executors.newCachedThreadPool();

            while (!*done*) {

                Socket *client* = *server*.accept();

                ConnectionHandler *handler* = new ConnectionHandler(*client*);

*connections*.add(*handler*);

*pool*.execute(*handler*);

            }

        } catch (IOException *e*) {

            shutdown();

        }

    }

**public** **void** broadcast(String *message*){

        System.*out*.println("Broadcasting: " + *message*); *// Display messages on the server*

        for (ConnectionHandler *ch* : *connections*) {

            if (*ch* != null) {

*ch*.sendMessage(*message*);

            }

        }

    }

**public** **void** shutdown() {

        try {

*done* = true;

            if (!*server*.isClosed()) {

*server*.close();

            }

            for (ConnectionHandler *ch* : *connections*) {

*ch*.shutdown();

            }

        } catch (IOException *e*) {

*// ignore*

        }

    }

**class** ConnectionHandler **implements** Runnable {

**private** Socket *client*;

**private** BufferedReader *in*;

**private** PrintWriter *out*;

**private** String *nickname*;

**public** ConnectionHandler(Socket *client*) {

*this*.*client* = *client*;

        }

        @**Override**

**public** **void** run() {

            try {

*out* = new PrintWriter(*client*.getOutputStream(), true);

*in* = new BufferedReader(new InputStreamReader(*client*.getInputStream()));

                System.*out*.println("Connection from " + *client*.getInetAddress()); *// Display client IP address*

*out*.println("Please enter a nickname:");

*nickname* = *in*.readLine();

                System.*out*.println(*nickname* + " connected!"); *// Display nickname on server*

                broadcast(*nickname* + " joined the chat!");

                String *message*;

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

                    if (*message*.startsWith("/nick")) {

                        String[] *messageSplit* = *message*.split(" ", 2);

                        if (*messageSplit*.*length* == 2) {

                            broadcast(*nickname* + " renamed themselves to " + *messageSplit*[1]);

                            System.*out*.println(*nickname* + " renamed themselves to " + *messageSplit*[1]); *// Display nickname change on server*

*nickname* = *messageSplit*[1];

*out*.println("Successfully changed nickname to " + *nickname*);

                        } else {

*out*.println("No nickname provided!");

                        }

                    } else if (*message*.startsWith("bye")) {

                        broadcast(*nickname* + " left the chat!");

                        shutdown();

                    } else {

                        broadcast(*nickname* + ": " + *message*);

                        System.*out*.println(*nickname* + ": " + *message*); *// Display messages on the server*

                    }

                }

            } catch (IOException *e*) {

                shutdown();

            }

        }

**public** **void** sendMessage(String *message*) {

*out*.println(*message*);

        }

**public** **void** shutdown() {

            try {

*in*.close();

*out*.close();

                if (!*client*.isClosed()) {

*client*.close();

                }

            } catch (IOException *e*) {

*// ignore*

            }

        }

    }

**public** **static** **void** main(String[] *args*) {

        Server *server* = new Server();

*server*.run();

    }

}

**Client Code**

import **java.io.IOException**;

import **java.net.Socket**;

import **java.io.InputStreamReader**;

import **java.io.PrintWriter**;

import **java.io.BufferedReader**;

**public** **class** Client **implements** Runnable {

**private** **Socket** *client*;

**private** **BufferedReader** *in*;

**private** **PrintWriter** *out*;

**private** **boolean** *done*;

**private** **String** *ipAddress*;

**private** **int** *port*;

**public** Client(**String** *ipAddress*, **int** *port*) {

*this*.*ipAddress* = ipAddress;

*this*.*port* = port;

    }

    @**Override**

**public** **void** run() {

        try {

            client = new Socket(ipAddress, port);

            out = new PrintWriter(*client*.getOutputStream(), true);

            in = new BufferedReader(new InputStreamReader(*client*.getInputStream()));

**InputHandler** *inHandler* = new InputHandler();

**Thread** *t* = new Thread(inHandler);

*t*.start();

**String** *inMessage*;

            while ((inMessage = *in*.readLine()) != null) {

*System*.*out*.println(inMessage);

            }

        } catch (**IOException** *e*) {

            shutdown();

        }

    }

**public** **void** shutdown() {

        done = true;

        try {

*in*.close();

*out*.close();

            if (!*client*.isClosed()) {

*client*.close();

            }

        } catch (**IOException** *e*) {

*// ignore*

        }

    }

**class** InputHandler **implements** Runnable {

        @**Override**

**public** **void** run() {

            try {

**BufferedReader** *inReader* = new BufferedReader(new InputStreamReader(*System*.*in*));

                while (!done) {

**String** *message* = *inReader*.readLine();

                    if (*message*.equals("/quit")) {

*inReader*.close();

                        shutdown();

                    } else {

*out*.println(message);

                    }

                }

            } catch (**IOException** *e*) {

                shutdown();

            }

        }

    }

**public** **static** **void** main(**String**[] *args*) {

        if (*args*.*length* != 2) {

*System*.*out*.println("Usage: java Client <IP Address> <Port>");

            return;

        }

**String** *ipAddress* = args[0];

**int** *port* = *Integer*.parseInt(args[1]);

**Client** *client* = new Client(ipAddress, port);

*client*.run();

    }

}

·