Client-Server Chat Program

GROUP 7

Contents

Members & Roles

GROUP NAME: The Four Horsemen of Narga

Horseman 1:
Gideon Daniel Botha (26894319@sun.ac.za) - Implemented GUI using JavaFX, Integrated front-end with back-end, Javadocs

Horseman 2:
Priyal Bhana (27040607@sun.ac.za) - Implemented ClientHandler.java, Project report, VPN using ZeroTier

Horseman 3:
Sulaiman Bandarkar (24234850@sun.ac.za) - Implemented Client.java, Project report, VPN using Zerotier, README.md, Makefile

Horseman 4:
Thaakir Fernandez (26479443@sun.ac.za) - Implemented Server.java, Assisted all group members with debugging, and added Javadoc and comments.

Project Overview

This project implements a multiple-client single-server chat application using Java. The goal was to design a scalable and efficient client-server communication system that supports group chats and private messaging (called Whispers). This report provides an overview of implementation features, program flow, design considerations, and experiments conducted by the group members listed above.

Features

Implemented features:
Everything that was listed in the rubric by the provided project specification.

Additional features implemented:
None.

Unimplemented features:
None.

File Descriptions

- Server.java: Handles incoming client connections, assigns each client to a ClientHandler, and manages the active client list.
- ClientHandler.java: Manages client communication, processes messages, and facilitates broadcasting of group and private messaging.
- Client.java: Implements the client-side logic, handling user input and communication with the server. The GUI is implemented within this class.
- Makefile: Automates compilation of the project.
- README.md: Provides instructions on how to compile the program and gives instructions on how to execute the program from the command line using the Makefile.

Program Description

Overview of Components and Interactions

The chat application follows a client-server architecture with three main components that work together to create a multi-user chat system with both global and private messaging capabilities:

1. Server: Acts as the central hub for all client connections, assigns ClientHandler instances, and maintains a ConcurrentHashMap for active users, indexed by usernames. The Server places each ClientHandler on its own thread. Each client is assigned a handler that facilitates message processing, user notifications and connection handling.
2. ClientHandler: This is a class that implements runnable. It processes individual client messages on the server side, handles private and group messaging and removes clients from the hashmap upon disconnection. Each ClientHandler maintains a dedicated communication channel with its corresponding Client through BufferedReader and BufferedWriter streams over a Socket connection. Additionally, the ClientHandler receives notifications about user disconnections and prints them to the Server's console.
3. Client: Connects to the server via a Socket and authenticates the username upon login. It allows users to send and receive messages and listens for incoming messages in a separate thread. The client application uses JavaFX for the GUI, which includes a text area for global chat, an input field for sending messages, and an interactive list of active users all handled using multi-threading implementations so that a user may receive and send messages concurrently.

Server and Client connection

1. The Server starts by initializing a ServerSocket on port 1234 and listens for connections.
2. The Server enters its main loop, waiting for client connections with ServerSocket.accept().
3. The Client connects to the server using the IP address provided by the ZeroTier VPN and the port number (1234).

- The Server reads and authenticates the proposed username from the Client (asks the user to retry if it's unique and non-empty).
- If login is successful, a new thread is created and a ClientHandler is placed on that thread for the Client to handle the Client's messages and they are added to the ClientList of online users.
- The Server notifies all existing clients of the new user. Messages are processed by ClientHandler, either via group chat or sent privately and Client can disconnect, notifying others of the exit via the group chat window.

Message Handling

1. *Global Chat Messages:*

- Client: The user enters text in the global input field and clicks "Send" or presses Enter.
- Client: Sends the message to the Server and displays it in the global chat window.
- Server: ClientHandler receives the message and calls groupChat().
- Server: ClientHandler sends the message to all other connected clients by accessing the clientList hashmap.
- Other Clients: Receive the message displayed in their global chat windows.

2. *Private Messages (Whispers):*

- Client: The user selects a recipient from the active user's list, clicks on their name, and clicks on "Whisper?". A private chat window opens for the selected user to type, send and view messages sent and received with the other client.
- Client: in the backend, "@username " is appended to the message and sends it to the Server. ClientHandler listens for and recognises the "@" prefix, calls whisper() and extracts the username from clientList as the recipient.
- Server: ClientHandler finds the recipient's handler and sends the message only to that user.
- Recipient Client: Receives the whisper, opens a private chat window if one doesn't exist, and displays the message.

3. *User Join/Leave Notifications:*

- When a user joins: The server broadcasts a notification and the updated user list to all clients.
- When a user leaves (by typing "/exit" or disconnecting): ClientHandler recognises this command and removes the client from the list, broadcasts a notification of the client's disconnection, and terminates the connection stably via the removeClient() method in ClientHandler and finally calls close() in the Client class to close all open connections and streams associated with this client.

Concurrency and Stability Features

1. *Thread Management:*

- Each ClientHandler runs in its thread, allowing concurrent handling of multiple clients.
- The Client runs a separate thread to listen for server messages, preventing UI blocking.

2. *Synchronization:*

- The Server uses a ConcurrentHashMap to safely manage the list of connected clients across multiple threads.
- Critical sections in ClientHandler are synchronized to prevent race conditions during message broadcasting.

3. *Connection Monitoring:*

- The Client continuously checks the connection status with isConnectionActive().
- The ClientHandler detects disconnections through IOException during message reading.
- Both Client and ClientHandler properly close resources (sockets, streams) when connections terminate.

4. *Error Handling:*

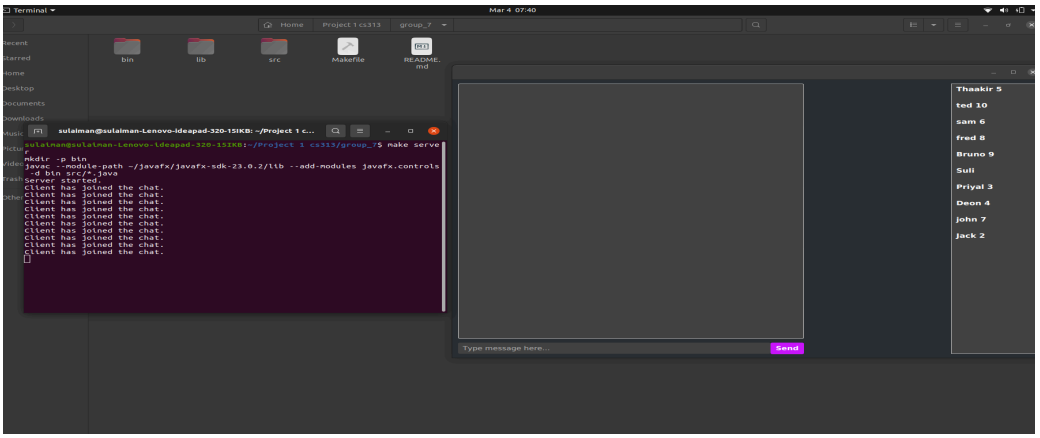
- Comprehensive try-catch blocks handle potential exceptions during network operations.
- Graceful disconnection procedures ensure all resources are properly released.
- The Client UI remains responsive even during network operations through the use of Platform.runLater() for JavaFX thread safety.

Experiments

1. How many clients can join the server before something breaks?

   Hypothesis:
   We predict that about 10 clients would be the maximum the server can handle.

   Results:

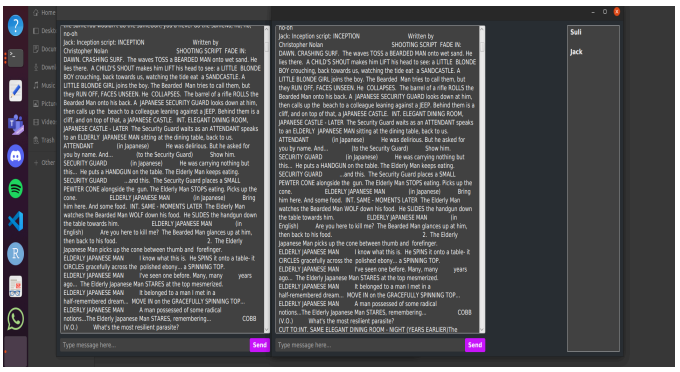


   Conclusion:
   The server can handle up to 10 clients. It can support more clients, but we did not want to continue in case someone's laptop crashed. At the 10th client, it took some time for the username login to pop up.

2. What's the maximum length of a message that can be sent without distortion?

   Hypothesis:
   We predict that no matter how long a single message is, it will be sent and received by another client. We tested it by sending the movie script of Inception.

Results:

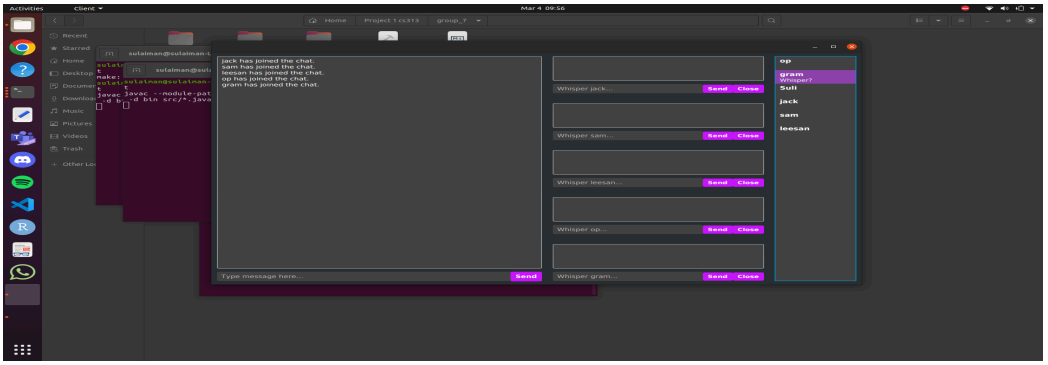


Conclusion:
Both messages were sent completely from one client to another with no distortions in the text.

3. How many whisper windows can be opened at the same time?

Hypothesis:
We predict you can open as many whisper windows as clients that are online.

Results:



Conclusion:
You can open as many whisper windows as possible but eventually, the chats will be too small to scale properly.

Issues Encountered

- VPN installation and setup - We overcame this and successfully managed to get 3 group members using LINUX connected to the ZeroTier VPN
- JavaFX installation and setup - We recognised we had a Java and JavaFX compatibility issue due to being on different versions of each. Once the necessary version updates were applied, the issue was resolved.
- Switching from strings to objects - We realised later on that using objects instead of strings for client-server communication would have made our GUI implementation as well as future submissions easier to implement. However, due to time restrictions, we were unable to

successfully refactor our code to use objects and decided to continue using our existing string implementation. Despite this, we plan to make the switch after the first demo.
- Multithreading issues which caused bugs in the whisper window - resolved by swapping between the JavaFX threads and backend threads to prevent errors and race conditions.

Design

- Use of ConcurrentHashMap: Ensured thread-safe management of active clients.
- Multi-threading: Each client runs in its own thread to enable concurrent messaging.
- Separation of Concerns: Server, client, and handler functionalities are modularised for maintainability.
- Server Console Logging: The server logs relevant events for easier debugging.
- JavaFX for GUI implementation.