REPORT

Computer Science 313 28 February 2024

A Single-Server, Multi-Client Chat Program

Group 36 members:

Dylan Hogan (26338319@sun.ac.za)

- Created the user interfaces, handled username validation, and contributed to public messaging.
- Transferred terminal functionality to GUIs.
- Finalised Report.

Joel Lee (23797207@sun.ac.za)

- Built initial project structure of Server, Client and ClientHandler classes.
- Handled writing of the report and terminal communication between the server and clients.

Josef Oosthuizen (26507404@sun.ac.za)

- Implemented private and public messaging.
- Handled client disconnects displayed on the terminal and GUI.
- Commenting and Documentation.

Overview

The aim of this project was to reinforce the concepts of Java sockets and concurrency by developing a chat program based on the client-server model. This involved implementing a single server that could handle multiple clients, with additional features to enhance the user experience.

Features highlight all necessary and additional features, as well as features we failed to implement. Detailed descriptions of the files and programs follows , including source files with their dependencies and the overall program flow. Experiments section outlines three different experiments conducted in accordance with the scientific approach to test the project. Issues encountered are listed, highlighting parts of the project that could not be implemented properly. The Design section includes rationale for GUI layouts, naming conventions, class structures, and usage of methods/variables.

REPORT

Features

Additional Features

- A user cannot whisper to a client that is not connected to the server. If they attempt to do so, the message: "<*username*> *not found*" will be printed to their corresponding Text Area.
- When you send a message globally or whisper, "[You]" will be printed to the left of the message.
-

Missing Feature

- The list of active users is unable to display for the clients on their GUIs.

File Descriptions

Project Dependencies

1. All .jar files from lib that are necessary for JavaFX applications.
2. Additional maven dependencies to build JavaFX applications.

Server.java

This class represents a server in a chat application. It accepts connections from clients, starts a new thread for each client, and handles the disconnection of clients.

Client.java

This class represents a client in a chat application. It handles the connection to the server, sending and receiving messages, and updating the GUI based on received messages. On startup, the scene MainScene.fxml is loaded.

ClientHandlers.java

This class serves as a thread for each client. It handles the receiving of messages from a client and forwarding them to other clients. The class also contains an ArrayList that keeps track of clients that are connected to the server.

Message.java

This class represents a message in the chat application. It contains information about the sender, recipient, type, and content of the message.

MainController.java

This class is a controller for the initial screen of the JavaFX application. It handles events such as entering a username and IP address and clicking the "JOIN SERVER" button.

REPORT

ChatGUIController.java

This class is a controller for the chat screen of the JavaFX application. It handles events such as typing a message and clicking the "SEND MESSAGE" button. It also updates the list of active users and the display of messages.

This class contains logic for sending messages to all the clients connected to the server. When a user enters a message and sends it with the button, they will see [You] <message sent>. The other connected users will see [username] <message sent>. To whisper, a user must enter /w [username] <messageToSend>. Only the whispered to the client will see (whisper from the username) <message sent>.

MainController.fxml

This is an FXML file that describes the layout of the initial screen. It contains elements such as a TextField for entering a username and IP address, and a Button for joining the server.

ChatGUI.fxml

This is an FXML file that describes the layout of the chat screen. It contains elements such as a TextArea for displaying messages, a TextField for typing messages, and a Button for sending messages.

Project Description

Initialization: The program starts with the MainController class, which initializes the initial screen of the JavaFX application. The user enters their username and the IP address of the server they want to connect to.

Server Connection: When the user clicks the "JOIN SERVER" button, the btnJoinServerClicked method is triggered. This method creates a new Client object, which establishes a connection to the server at the given IP address.

Client-Server Interaction: Once connected, the Client object starts a new thread that constantly listens for incoming messages from the server. When a message is received, it is processed and displayed in the chat GUI.

Message Sending: In the chat GUI, the user can type a message and click the "SEND MESSAGE" button to send it. The btnSendMessageClicked method in the ChatGuiController class is triggered, which creates a new Message object and sends it to the server through the Client object.

Message Receiving: The server receives the message and forwards it to all connected clients. Each client receives the message and displays it in their chat GUI.

Private Messaging: If the user wants to send a private message, they can do so by starting their message with "/w ". The ChatGuiController class recognizes this as a command to send a private message and processes it accordingly.

REPORT

Leaving the Chat: If the user wants to leave the chat, they can do so by typing "/leave". The ChatGuiController class recognizes this as a command to leave the chat and processes it accordingly.

User List Updating: The updateUsers method in the ChatGuiController class is used to update the list of active users in the chat GUI. This method is called whenever a user joins or leaves the chat.

Stability and Concurrency: The program is designed to handle multiple clients at the same time. Each client runs on its own thread, ensuring that the program can handle multiple messages at the same time without blocking. The JavaFX application also runs on its own thread, separate from the client and server threads, to ensure that the GUI remains responsive at all times.

Termination: The program ends when the user decides to leave the chat or when the server is shut down. In either case, all connections are closed properly to ensure that no resources are leaked.

Experiments

Experiment 1

Question: Does increasing the number of concurrent threads affect the performance of the server?

Hypothesis: We hypothesized that increasing the number of concurrent client connections would decrease the performance of the server as adding clients adds more processing overhead.

Experimentation: We created start and end time variables that got between method calls. This got the time difference in nanoseconds. In the first experiment, we added one client and measured. For each extra experiment, we added two more clients, and we stopped the experiment with five clients.

Dependent variable: Time taken for method calls.

Independent variable: Number of client connections.

Analysis: We tested the same principle across multiple different methods. It appears that the difference in times between increases as you add more clients, however, the difference is not very significant.

Conclusion: It appears that although our initial hypothesis is correct, the nature of multithreading and thread safety allows the server to not become bottlenecked when a new client connection is established.

REPORT

EXPERIMENT 2

Question: Does a user have to use bufferedReader and bufferedWriter to handle input and output for client-server interactions?

Hypothesis: We assumed that a client must use bufferedReader to read from the InputStream that is connected to the socket and to print to all clients we must use bufferedWriter.

Experimentation: We started by using bufferedWriter and buffereWriter to set up server-client communication. This worked as intended and the outputs were printed to the terminal for all clients to see. The problem we faced was that we could not use the same logic to print the message to all clients on a TextArea.

Dependent variable: The performance of client-server communication.

Independent variable: The choice of reading input and output from server/client.

Conclusion: Although using bufferedReader and bufferedWriter are advised for sever-client communication, when handling GUI elements it proved to be difficult to use bufferdWriter to write the message to all clients connected to the socket. It therefore is reasonable to use other methods that allow for similar results.

EXPERIMENT 3

Question: Does Hamachi hosting add additional network latency that affects the client-server interaction?

Hypothesis: We hypothesize that hosting using Hamachi will add additional network latency and will therefore give worse performance for client-server communication as opposed to local testing.

Experiments: Our first scenario was using System.nanoTime() to measure the total time it takes for a user to send a message to the server and for the server to send the message back to the client. This was tested locally on one machine and the time was recorded. We repeated this experiment by adding two extra clients. This scenario was repeated but now for hosting using Hamachi, we originally let the server connect to a client on the same machine. Then we added two clients from two different machines. The times were taken after each test in each scenario and were later analysed.

Dependent variable: Message latency.

Independent variable: Choice of network setup.

Analysis: There are noticeable time differences between sending/receiving on Hamachi versus local hosting on the same machine. Our hypothesis was proven correct as the local network for hosting on the same machine had higher bandwidth and less latency than a network setup with Hamachi.

Conclusion: Hamachi does come with a slight performance difference, however the need for clients to communicate over multiple machines to a single server is a crucial feature. So the trade-off for performance is worth it.

REPORT

Issues Encountered

- **Username Validation:**
  This problem was persistent for a long time. Originally, we tried to validate on the login button click against a global list of users. We never succeeded implementing this.
- **Hamachi Server:**
  Linux causing many issues.
- **Displaying connected clients on GUI:**
  Tried many different things such as ArrayLists, ObservableLists, TextAreas, ListViews and more. We could not get the global list of activeUsernames to display for all clients on their GUIs.
- **Clients only notified about disconnection when client uses /leave**
  Unexpected disconnection does not notify other clients.

Design

We implemented a *Message class* so that we could pass a message object between input and output streams. We chose to send this object rather than sending a string so that we could handle validation logic between classes.