Group19 Project 1: A Client-Server Chat Program

Nicholas van Huyssteen, Lehann Smith
20869118@sun.ac.za, 19027176@sun.ac.za

7 August 2019

1 Introduction

The purpose of this project was to implement a chat program based on the client-server model with multiple clients being able to connect and communicate simultaneously. Furthermore this project serves to demonstrate the working of- and common problems programmers typically experience when designing and implementing applications of this kind. This document will consist of three broad categories divided up into sections. In the first part will be a brief description of the program, the files it consists of and it's most significant methods and data structures. Secondly will follow a summary of our experience during the project outlined in the experiments we conducted with our nascent chat program and the major or interesting issues we encountered during implementation. Finally it will provide an overview of the pertinent features of our implementation, including design choices and running instructions.

2 Program Description

The server is started up, we create and instantiate printwriter for every client that connects, and a scanner for use with this server. When a new client with a validated name connects to the server, the server adds him to the list of active clients and broadcasts this list to all users. The server waits for input, classifies processes the input if needed, and writes it to all the clients. The server also detects when a client leaves, updates the active client lists and once again broadcasts the list of currently active users to all the connected clients. On the clientside, the user launches initGUI and is presented with

a screen that captures/validates the address he wants to connect to and his chosen name. If passed, the chatGUI is instantiated. Every client waits for input to display, passing the contents to the GUI files to show on the relevant text area's. Every client also has action listeners that detects when the user wants to send input, captures the data and uses its printwriter to send it to the server. The user may choose to disconnect, the program closes their input/output stream and the socket before terminating the client.

3 File(s) Description

ChatServer.java: This file controls all interaction on the server side of the application. We make use of a system of printwriters, one spawned for every client connected to the socket(bound to a client name) and a scanner to read any input the clients want to send to the server/ each other. The server also handles broadcasting general info, like users joining or leaving. The server has an update method that is called everytime a user is added to or taken off the list of active users. This method broadcasts the currently active list of users to every user in the central arraylist so they can update their own visual lists on the GUI. The server contains a loop that continually checks for input from clients.

ChatClient.java: The client opens a socket and generates a personal scanner to detect input from the server, and its own printwriter to write to the server. It continually checks if the scanner has new input available, classifies the input type and prints the input if appropriate. It also contains the methods responsible for receiving the names of new users to append to its list of active users.

initGUI.java: Handles the set up of the GUI proper, ask the user for a name and a server adress, sends the name to the server for validation, and captures the data if correct. Creates a new instance of ChatClient and starts chatGUI.

chatGUI.java: ChatGUI is the user interface for communication. It has a reference to the chatClient created in initGUI. It defines the look and feel of the interface, and provides the user usable functions for communication. It captures the data needed for sending messages via button press or key press from the textfield. In addition it has the method for disconnecting the client and closing the sockets and streams. It also provides the first iteration of processing messages/whispers received from the client. During the processing

the method appends the correct prefixes to outgoing messages so the server can recoqnize them appropriately. Lastly it has the method that creates the listener thread that detects users connecting and leaving.

4 Significant Data Structures

We made use of ArrayLists on the serverside to store the printwriters responsible for writing to each client, and to store the list of names (as strings) of currently connected clients. We also made use of an ObservableList on the client side to display the connected users.

5 Experiments

1: The first experiment consisted of trying to connect the chat service from two different Narga machines using public IP's. This was successful with all functions (user list, messaging, whispers, disconnecting) working as expected. From this we concluded that as long as the port is open and the server's IP can be made known to the user, our implementation will work on at least this (Narga Ubuntu distribution PC's) architecture and network.

2: The second experiment was to determine how many clients we could connect from one machine. In the Narga Computer area, we were able to connect 8/9 clients before the performance of the computer deteriorated beyond reasonable use. The server and each individual client was still functional but the machine's memory was loaded to the point of extreme input delay. The program functioned as expected when connecting to the server's public IP from a different machine. Therefore we drew the conclusion that the demands our program placed on memory far outweighed the burden of communication it engendered on small scale. We can make no conclusions as to how true this would remain given an ever increasing number of clients connecting to the server. We can however speculate that our implementation would not scale to some large number of users, given that we implemented sequential access to our most critical data structures. In particular users disconnecting and leaving would likely prove a bottleneck of some magnitude if a large enough quantity of users connected/disconnected with high frequency.

3: We tested trying to transmit certain phrases via the text input that we knew the server used for processing. Because of our system of appending prefixes, transmitting "reserved phrases" failed in all cases but one. We

could not change usernames, force updates of the user list or cause any other reserved phrase to trigger through the user input, except if the user used our whisper convention for serverside processing, they were able to "fake" being another user and whispering to a third party. This was fixed with a sanitisation check. We concluded from this that while a system of appended messages to input was certainly useful and relatively intuitive to implement, it has the downside of being vulnerable to malicious users fairly easily unless vast and thorough input sanitisation is applied with great care.

6 Issues Encountered

We at first attempted to implement the communication service with the use of Java's ObjectInputStream and ObjectOutputStream classes. We ran into problems due to the blocking calls made by reading and writing objects, and stream corruptions when we tried to solve this by creating listener threads that terminated after a set time. We then instead decided to use a system of distinct client-specific printwriters and scanners to transfer information between parties. A further issue appeared when we constructed a working version with a self contained simple GUI in parallel with a more complex, separate GUI. Bridging the separation of the GUI and the back end files proved the most time consuming challenge. The last major issue we encountered was in communicating an up to date user-list from the server to the client and displaying it on the GUI in real-time. We implemented an update method that is called whenever a change is made to the list of active users, and then writes the complete lists of active users to every connected client so they can update their visual list. We had one last curious problem. We used a flag boolean (initialised to false, and set to true whenever this client has created a socket and opened its in/output streams successfully) to restrict opening the chatGUI until all the data and connections it will need is in place. Despite the variable succesfully being changed, the if conditional was never passed, until we changed it from if(flag) to if ((flag+"").equals("true")) upon which it worked as expected.

7 Design

We made the decision to have a clickable list of the users connected, that allows the user to select someone to whisper to. Upon selection and user confirmation, it appends the whisper convention for the user to his whisper partner to the user's textfield.

8 Unimplemented Features

We do not have any explicit locks to protect the user lists, but as all of the user list data structure is done serverside in a single thread with an implicit queue forming from any client input, we do not have a race condition here.

9 Additional Features Implemented

A clickable list of currently connected users, that asks this client if they want to whisper with the client they selected from the list. The client automatically appends the convention for using the whisper function to that specific user to the textfield. We also handle a special case text input in the event that the user attempts to duplicate our serverside processing conventions to emulate a different user when whispering. A message informing the user to cease and desist appears on the message output area.

10 Compilation

The project is compiled with maven. In the repository directory open a terminal and run the script ./start.sh [n=number of clients] to compile and run the server and n clients. You can also run the script addClient.sh [n=number of clients] to add n clients without creating a new server, or addServer.sh to start a new server.

11 Execution

The project is compiled with maven. In the repository directory open a terminal and run the script ./start.sh [n=number of clients] to compile and run the server and n clients. You can also run the script addClient.sh [n=number of clients] to add n clients without creating a new server, or addServer.sh to start a new server.

12 Libraries

We did not make use of any external libraries.