Networks CS 3516

Client-Server Chat Application

Andrew Shanaj

June 9, 2020

# Abstract

This report outlines the design and development of my Client-Server Chat Application based on a single server, multiple client model. The project was written in Java alongside the JavaFX framework for the GUI. The program was designed and tested on the Windows operating system but should also work on UNIX systems based on Java’s flexibility. For compilation, I used Gradle. The design of the project is modular in nature and makes maximum use of abstract data types and makes use of a few different design patterns. Particular attention was paid into the communication protocol between a client and server as well as tying together the JavaFX controllers. This report contains my design, testing and evaluation as well as annotated screen shots demonstrating the program has been successfully implemented.

# Content

Project Description …………………………………………………pg 4

Project Design …………………………………………………......pg 5

Testing and Evaluation ……………………………………………pg 13

Future Development ……………………………………………….pg 14

Conclusion …………………………………………………………pg 14

Appendices …………………………………………………………pg 14

# Project Description

This project is a Chat Program based on the client -server model using socket commands. The Transfer Control Protocol (TCP) was used over my localhost network. Socket communication is at the heart of the project. In order to develop the project a few things needed to be done.

The Server:

A single server program should be able to handle all requests from our clients. A multi-service solution was implemented by threading each client that joined our server. The following have been implemented:

1. Server Operations (Connect/Disconnect requests)
2. Handling Connections/disconnections
3. Clients must have unique names
4. Clients must be informed of changes in the list of connected users.

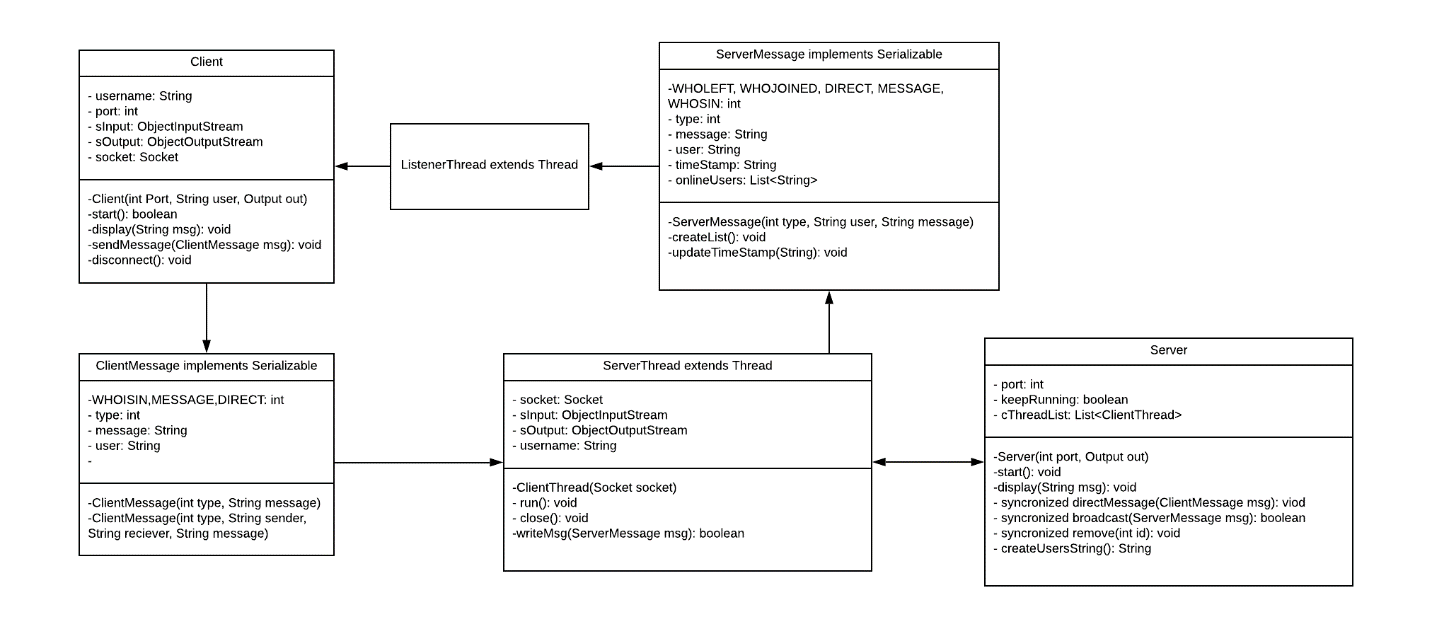
The Client:

A client must be able to connect to the server based on the port number given over our localhost network. The following have been implemented:

1. A list of online users is displayed
2. Other Client Connection/Disconnection actions are displayed
3. Private chatrooms for private messages as well as a public chat room
4. Client is always listening to messages/actions
5. Client is able to disconnect without disrupting the server

# Project Design

Client-Server UML:



(Fig 1.1)

First, I wish I could resize the UML, but I can’t. I did paste in a vector image though so you should be able to zoom in.

The way that our Client and Server system works is like this. We initialize the server, and when a new clients request to join, the server creates a client dedicated thread that is continuously reading or sending messages to the client. The client object also creates a listener thread that continuously reads messages through the socket. These messages are either a ServerMessage or ClientMessage attribute, since sending an object is the simplest way to stream data.

**ServerMessage and ClientMessage:**

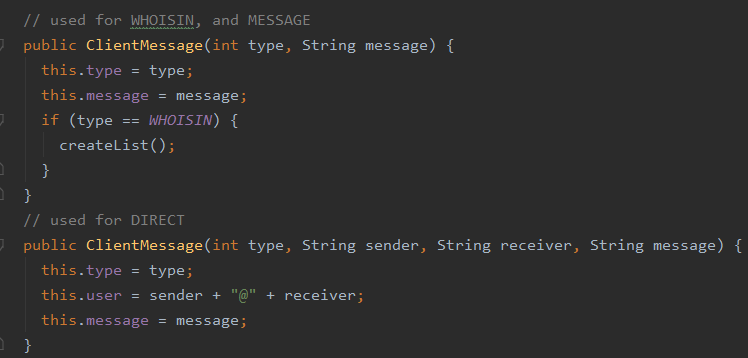
The ServerMessage and ClientMessage data structures are a way to easily send and receive data. They both have a type attribute that determines a command/action.

**ClientMessage types**: *Client to the Server*

*WHOISIN* – asks the server for all the users that are currently online, then creates a List<String> with all online users.

*MESSAGE* – an ordinary text message that is meant to be broadcasted to everyone.

*DIRECT* – used to send a direct message to a specific user. We send the sender and who should be receiving the message.



(Fig 1.2)

**ServerMessage types:** *Server to the client(s)*

*WHOLEFT* – notify who left the server, broadcasted to everyone.

*WHOJOINED* – notify who joined the server, broadcasted to everyone.

*DIRECT* – send a direct message to a user, we also include who sent the message.

*MESSAGE* – send an ordinary message, broadcasted to everyone.

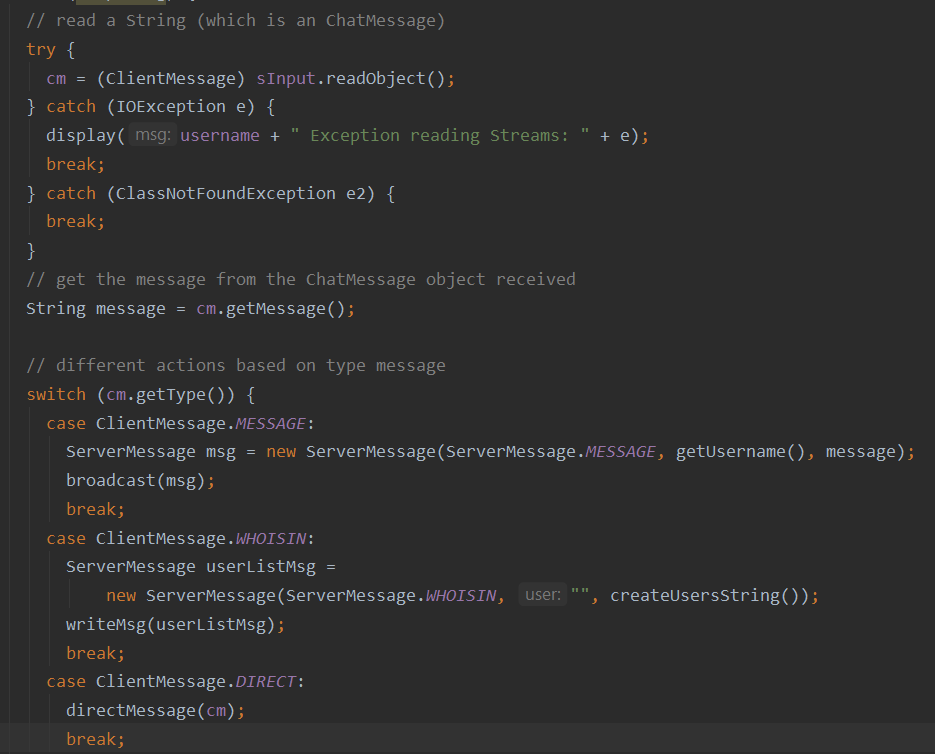
*WHOISIN­* – returns all users that are currently in the server.

Everything else in this datatype is getter and setters except our createList() function in the ServerMessage, which just manipulates the message variable after receiving a WHOISIN type

**Server and ServerThread:**

1. First we create our server, which jumps into our start() function. From there it creates a ServerSocket object, then enters an infinite loop that waits for Clients to join.
2. Once a Client tries to join, we accept the Client aka socket, and then create a ServerThread, add it to our threadList (in order to keep track of it), and repeat searching and accepting.

The ServerThread is a thread that links directly to a client’s socket.

1. The ServerThread does all its prep work such as initialize its Socket Input/Output Stream, and then it broadcasts a ServerMessage object to all the other Clients that it joined the server.
2. After it goes into an infinite loop reading and forwarding messages.
3. Once it reads a ClientMessage from its socket input stream, it we check what Type of message it is, and perform its specified command.

(Fig 1.3)

The ServerThread also has the *close* function which closes it streams and socket as well as a *writeMsg* function that takes a ServerMessage datatype and streams it to its socket.

The *bradcast* and *directMessage* functions are the functions that allow for communication between different clients. Both functions are synchronized and loop through our threadList. Broadcast also makes sure that all threads can be written to, and if not disconnect them.

**Client**

The Client object simply connects to the server, and then creates a thread that listens for incoming ServerMessages. The Client object also sends a ClientMessage to the server.

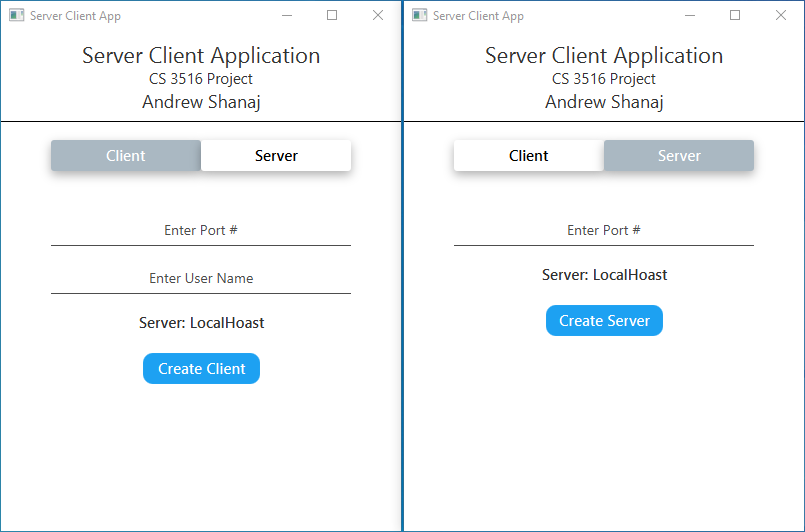
When our ListenerThread reads a ServerMessage it sends it to our singleton ScreenController, which is didn’t get to refactor, and I can’t think of a better name for it.

**ScreenController –** The middleman between the JavaFX objects/controllers and our Client object.

The ScreenController singleton does a few things

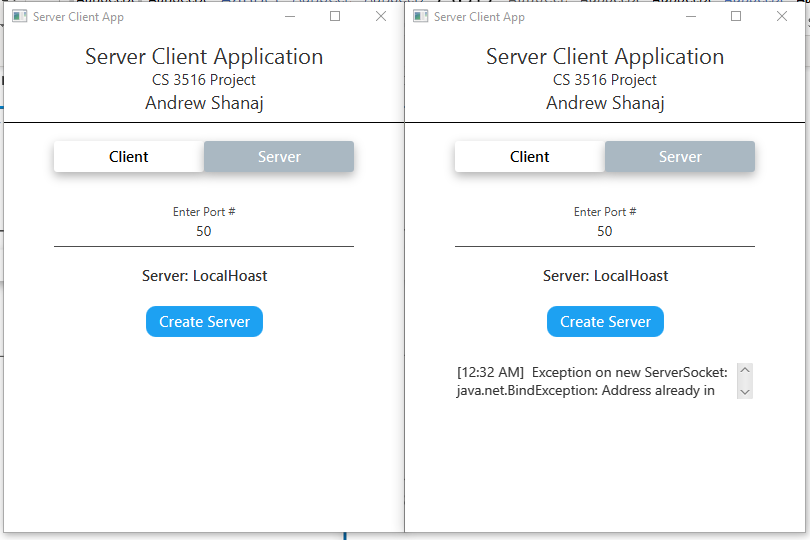
1. Changes the scenes of our GUI
2. Decides what to do with the received ServerMessage. It displays messages, both Server messages and private messages, or creates a notification
3. Keeps text message data persistent.
4. Updates our GUI Client list.

**GUI and JavaFX Controllers**



(Fig 1.4)

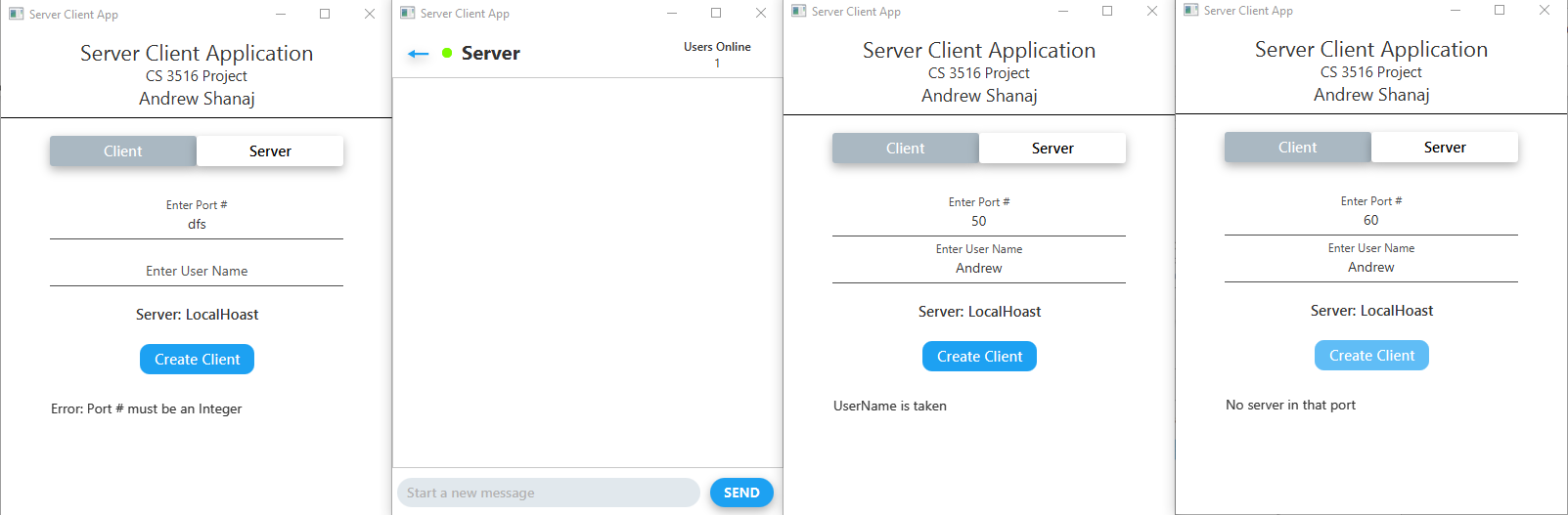
When we start our app we our greeted with our ClientStartPage, but we would need to create a server first. We can create our server depending on the port number we use.



(Fig 1.5)

We created our server with port 50 first to the left, and then tried to create the server we the same port again but ran into an error; the port is already used. This is where the Server’s gui ends.

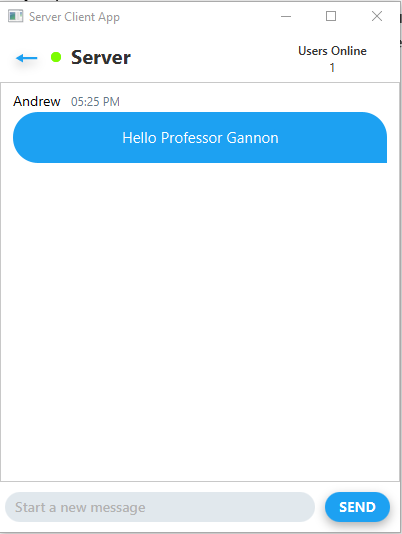
We also have error checking in our Client side of creation.



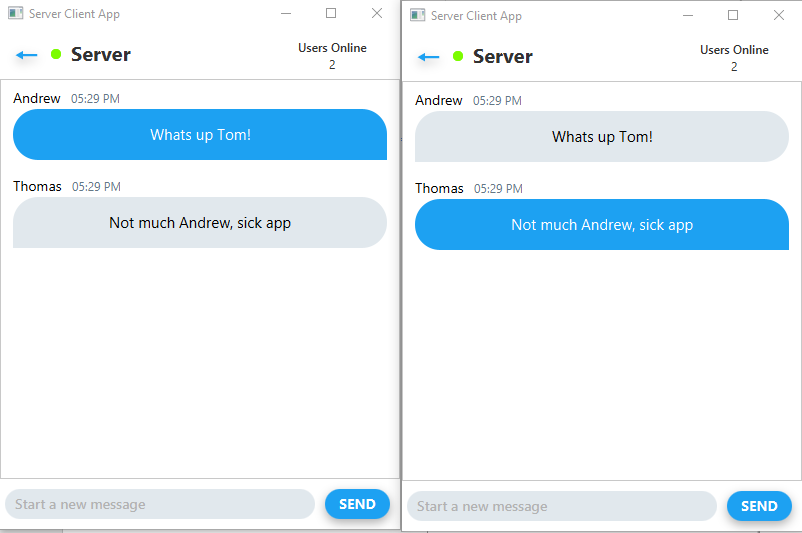
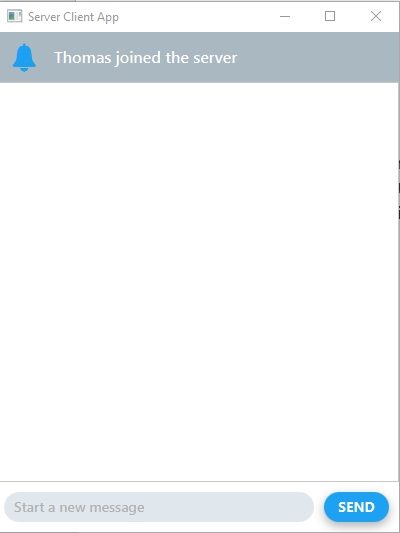
(Fig 1.6)

The first instance is an error checking that the port number is in fact an integer. The second instance, we entered the server with the User Name Andrew, and in the third instance we try to create a client with the username Andrew again, prompting an error message as the assignment desired. The way that I accomplished this is by creating a Tester client that simply gets all the current users in the server and compares if we should continue with creating our new client with the given username. The fourth instance we try to enter a server that hasn’t been created yet, which also prompts the error. This logic is inside our ClientStartPage controller.

When we enter the server, we jump into our Server wide Chatroom. This screen displays the number of users currently online on the top right, a back button on our top left, and a input bar with a send button in the bottom as showed in figure 1.6. From here we can send a server wide message.

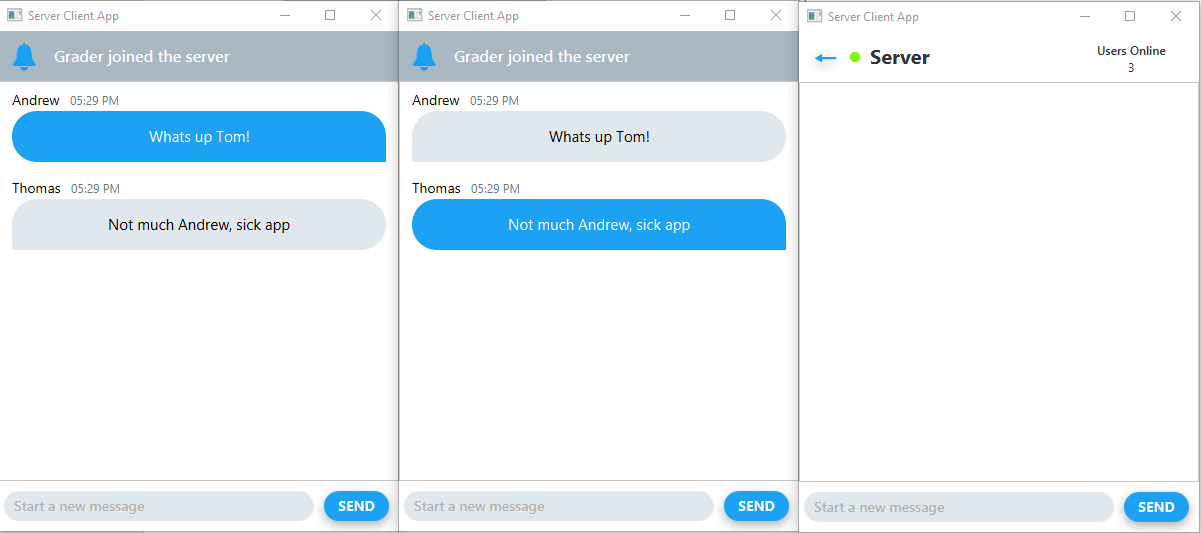


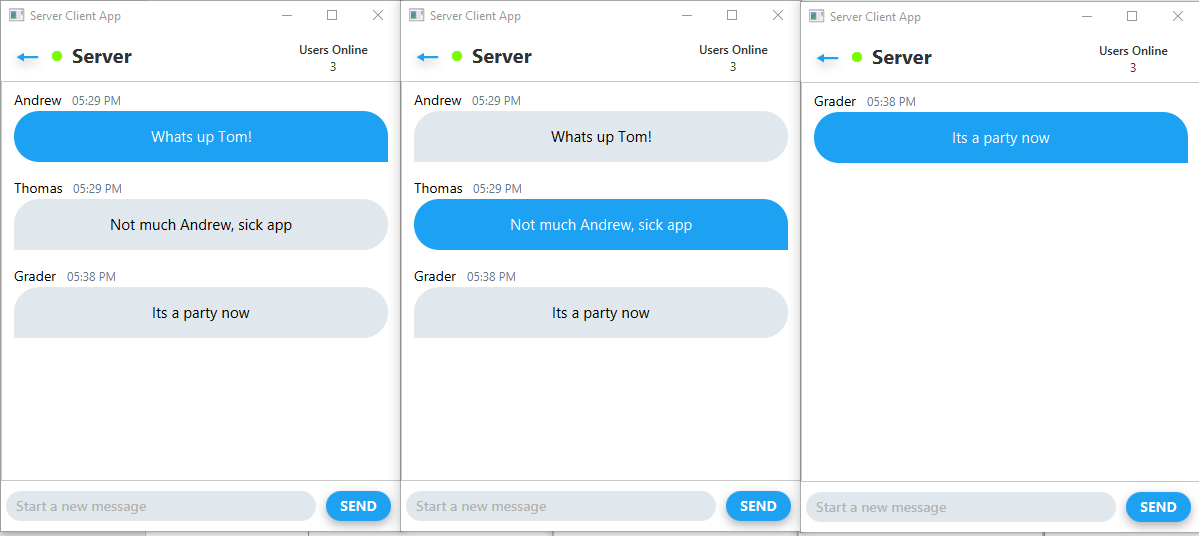
This isn’t too interesting so let’s add another client in our server.

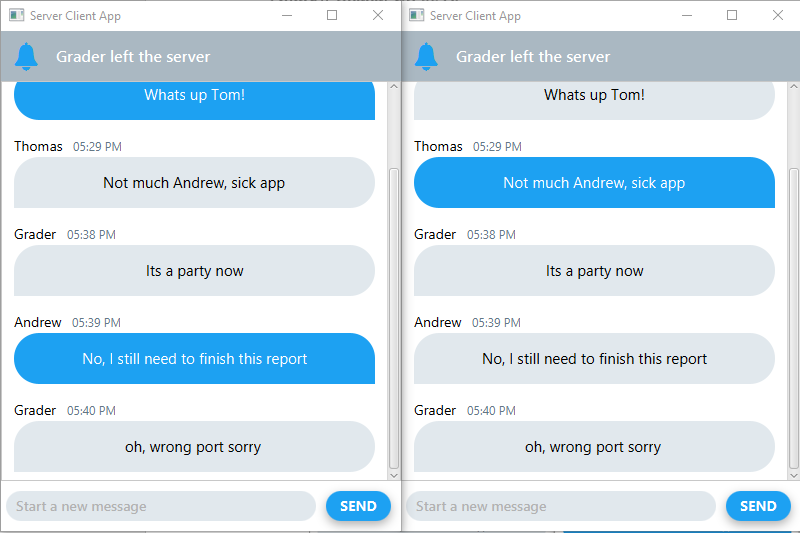


(Fig 1.7)

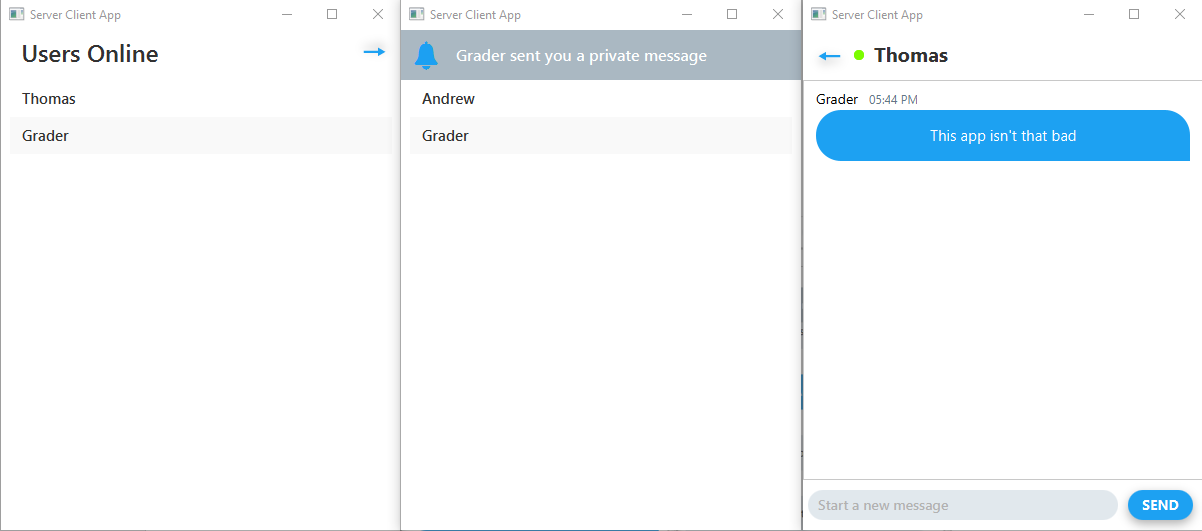
When another client enters the chat room, we get a notification pop up for 2 seconds and displays a message, as shown in instance 1 of fig 1.7. This happens in our Screencontroller when it parses our ServerMessages. This notification tab is threaded, allowing him to type while it is being displayed within those 2 seconds. The 2nd instance is Andrew’s application and the 3rd is Thomas’s, who exchanged some information. Senders chat bubbles are blue, and other clients are grey. Inspired by Twitter.





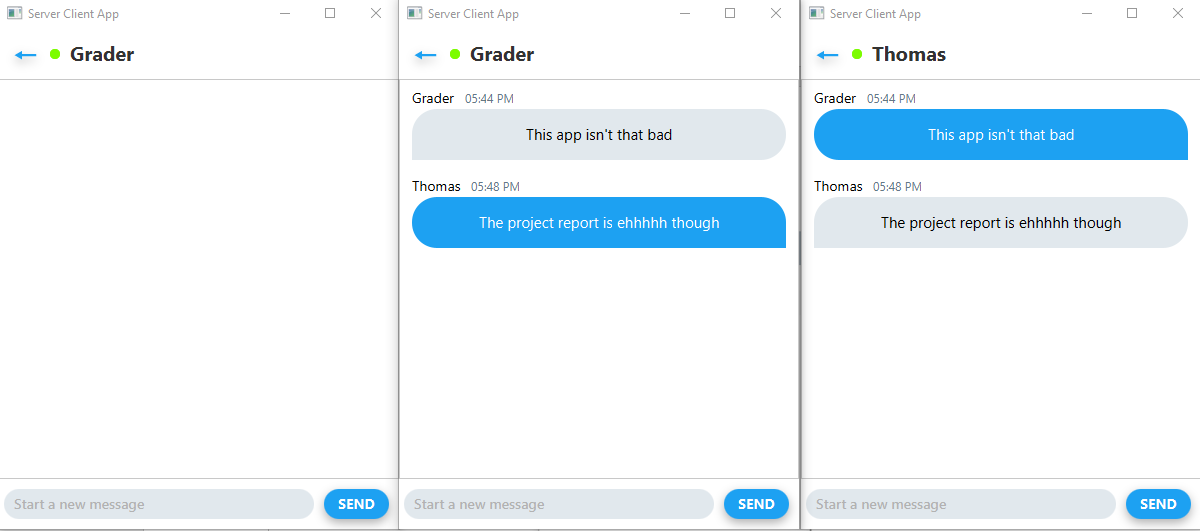


We can receive a who left, who joined and direct message notification. Lets bring another client back though.



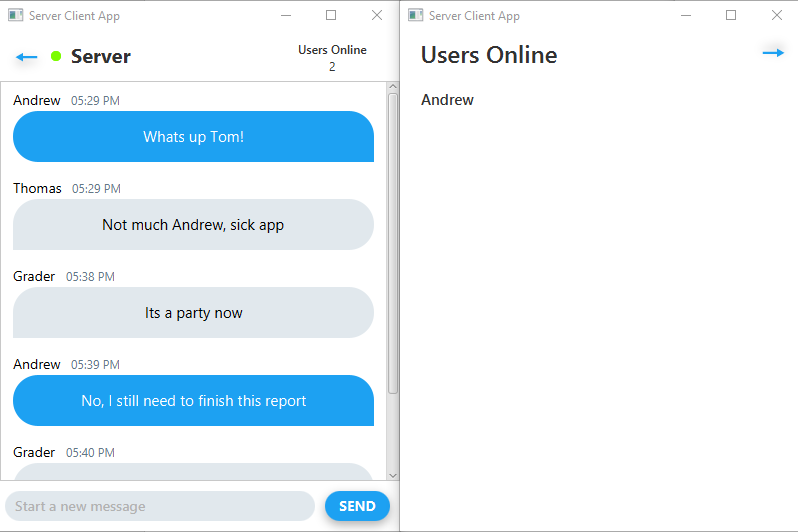
(Fig 1.8)

In figure instance 1 and 2, Andrew and Thomas, respectively pressed the back arrow to display a list of users in the server. Whenever a client goes into the list few, he receives a message from the server with all the active users. While in instance 3, the Grader selected Thomas to send a personal message to, notifying Thomas since he isn’t currently in that chat room.



(Fig 1.9)

Thomas then entered the private chat room with the Grader and responds. Andrew also joined a chat room with the Grader just to show that the chats are private.



(Fig 1.8)

All the messages stay persistent, and as users safely exits the server, we update our list as well as our counter which concludes my application.

# Testing and Evaluation

For testing I created a few jUnit tests as I was starting my project. I used my output object to check if what I wanted was correct. These jUnit tests were pretty basic, I tested Server creation as well as client creation, making sure that I am able to create a server as well as clients being able to join the server. As I started creating my GUI, most of my testing became GUI based testing. I added a error detection handler in my ClientStartPage that shows errors that come up to begin with. Once I was able to create my clients, the testing that ensued was making sure that every client is able to receive Server wide messages, the list stays updated as clients join or leave and that direct messages are truly personal between two clients. After a lot of testing, I couldn’t conclude any errors from the Sever, Client side of things.

While my Server and Client operated as expected after a bunch of bug fixing, my GUI could’ve used some bug improvements, which brings me up to my next sections.

# Future Development

While I was able to complete all the deliverables that were asked for, there is still a good amount of work that can be done. For starters, cleaning up my GUI and making it more appealing would be awesome. A few GUI bugs persist, such as when you create a server and get an error message, then create a new Server, that error message persists, even though the server does get created. Making my client list more appealing, instead of just a simple list with their names on it was also something that I wanted to get to. If I had more time, adding in a way to send images would’ve been possible. Simply creating a new type in our ClientMessage and ServerMessage that corresponds to an image, followed by an image type would’ve made this work. It would’ve even made it possible for users to add in their own avatar image that can be displayed to everyone. One of the coolest and most time-consuming development that I wish I had time to add would’ve been to create a live video chat feature. This would’ve meant that I had to rework the way that my Server and Client work. I would need to be able to send a continuous stream of data, possibly images at a given framerate and then display them. Maybe once my schedule clears up, I will get to it!

# Conclusion

This report described the successful design and development of my application. I am quite proud of how it turned out and I learned a bunch about socket communication and how to manage a Server and client communication protocol. I also expended my knowledge of threads greatly as well as the development of a JavaFx application.

# Appendices

References:

<http://tutorials.jenkov.com/>

[https://docs.oracle.com](https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html)

Test Cases and Test Results:



