Chatet : CS354 Project 1 - Group 41

Gordon McGaffin, 23565608

Johan Visser, 21553416

Assignment presented in the partial fulfilment
of the requirement for the course
CS 354

Lecturer: Prof. William D. Tucker

TABLE OF CONTENTS

1

INTRODUCTION

This report will discuss the results of the third year Computer Science 354 project, creating a chat program using the client-server model. The task is to create a chat platform that has:

- multiple clients connecting to one server,
- a multi-threaded server for handling multiple clients,
- a system to inform users of currently connected users,
- a simple client GUI,
- a system to allow users to choose a unique nickname,
- a system that allows users to broadcast a message to all users,
- a system to allow uses to send a private message to another user,
- a system to allow users to connect to/disconnect from the server with out disruption.

With the above-mentioned functionalities in mind, we have created 'Chatet'. 'Chatet' is a chat program that allows users to send messages to other users and allows a user to receive a message from other users.

A goal of this project is to create a chat system based on the client-server model. The user should be able to navigate a user interface that allows them to either personally message another user (known as a whisper) or message all users on the system.

An in-depth discussion on the design of 'Chatet' will be presented in the remainder of this report.

2

UNIMPLEMENTED FEATURES

The project implemented all the features listed in the requirements.

3

ADDITIONAL FEATURES

- The server runs a thread that is used to input instructions into the server without blocking incoming connections from clients. This allows the server admin to shut down the server by entering ”exit” into the terminal.
- If the client attempts to run while the server is not online, it will periodically try to reconnect to the server while displaying a dialog box, which informs the user that the client program is attempting to connect to the server.

4

PROJECT FILES

4.1 SERVER FILES

- `Server.java`: This file contains the code for setting up the server: this includes setting up the socket for listening for incoming connections, creating a thread to handle each client connecting to the server, and storing the threads into a `ConcurrentHashMap`.
- `ClientHandler.java`: This file contains the code for handling each client's operations. The file implements the `Runnable` class and receives messages (a `Message.java` object) from the client (`ClientGUI` class) and handles the message depending on its contents.
- `Message.java`: This file contains the code for the message object used to transfer data between the clients and the server.

4.2 CLIENT FILES

- `ClientGUI.java`: Instantiates the client and provides a graphical interface for the user.
- `Client.java`: Implements the client networking logic. It connects to the server and starts two threads, one for receiving message objects, and another for sending. Two `ArrayBlockingQueues` are used to store received and sent messages before they are processed. Two methods are made available for sending and receiving messages, and another for disconnecting from the server.
- `Message.java`: This file contains the code for the message object used to transfer data between the clients and the server.

5

PROGRAM DESCRIPTION

The program is a chat system that is comprised of two entities, the server and the client interface. The server part of the project handles the connection of clients to the server and message passing between clients connected to the server. The client interface part of the project handles the GUI that a user interacts with and the sending and receiving of messages to and from the server to the client. The client interface is also responsible for presenting information to the user, such as received/sent messages, the status of the server and the clients connected to the server.

The program execution follows the following steps:

5.1 SERVER

- The server program is first initiated and creates a socket that is used to wait for any incoming connections.
- Once the server has set up the socket, it creates a thread that handles input to the server. The thread begins executing and waits for the server admin's input.
- The server blocks until a client connects to the open socket. The server will create a thread, known as the `ClientHandler`, that handles the client's input and output. The server then begins to execute the thread.
- Once the username is provided by the client and verified to be unique, the thread created to handle the new client is stored in a `ConcurrentHashMap` and executed, with the client's username being used as the key.
- Once the thread begins execution, the server goes back to waiting for a new client to connect to the open socket.
- If the server admin types "exit", the server will close the socket and exit.

5.2 CLIENT

- When a `ClientGUI` is run it starts a `Client` instance, which attempts to connect to the server's socket.
- Once the client is connected to the server, the user is prompted to enter a username. If the name is unique, the client will receive a message from the server indicating that it may proceed. However, if the server informs the client that the name is unavailable, the user has to try a different name.
- Once the username is set, the graphical user interface is shown to the user and a new thread that checks for incoming messages is started.
- The class `Client.java` is responsible for all of the networking logic. It uses two `ArrayBlockingQueue`s to store messages before they are processed (i.e., received and sent from/to the server). Messages are sent and received on separate threads.
- When a user enters a message in the GUI and presses the send button, a new `Message` object is created with the content of the user's message, the user's username, as well as the username of the receiver as its data.
- The `Message` object is then passed to the `Client` to send. In the `Client`, the message is added to the outgoing message queue.
- On the thread responsible for sending messages an attempt is made to take the first message from the outgoing queue – this will normally be the last message added to the queue. If there are no messages the method call blocks until at least one message has been enqueued. Once a message is retrieved from the queue, it is sent to the server.
- The process for receiving messages to the client is similar.
- When a message is received, it gets displayed on the GUI.
- When the user wants to disconnect and close the client they can simply type "exit" in the input field and click the send button. This will cause all streams and sockets to be

closed before terminating the program itself.

6

EXPERIMENTS

6.1 CLIENT LIMIT EXPERIMENT

- Experiment: Observe what happens when the maximum number of clients connect to the server. The maximum number of clients is dictated by the maximum number of threads that has been designated for the server.
- Theory: The server will crash and throw an error message when another client (one more than the maximum number) attempts to connect to server.
- Test: To test this, we opened enough clients to exceed the limit set for the server. For the test, the limit for thread creation was set to three, this would allow only two clients to connect, as one thread is used for input into the server. Please see figures 6.1, 6.2, 6.3, and 6.4 below.
- Findings: The server did not crash. Instead the server blocks until a thread is available for the client to handle the client. When one of the connected clients disconnected from the server, one of the waiting client applications was able to connect.

6.2 MULTIPLE SIMULTANEOUS CLIENT CONNECTIONS

- Experiment: Observe what happens if a set of clients are waiting to connect to the server before it has started.
- Theory: The server will crash due to too many clients trying to connect to one single socket.
- Test: To test the system, we opened several clients before we started the server. Then, once all clients were attempting to connect, we started the server. Please see Figures 6.5, 6.6 and 6.7 below.

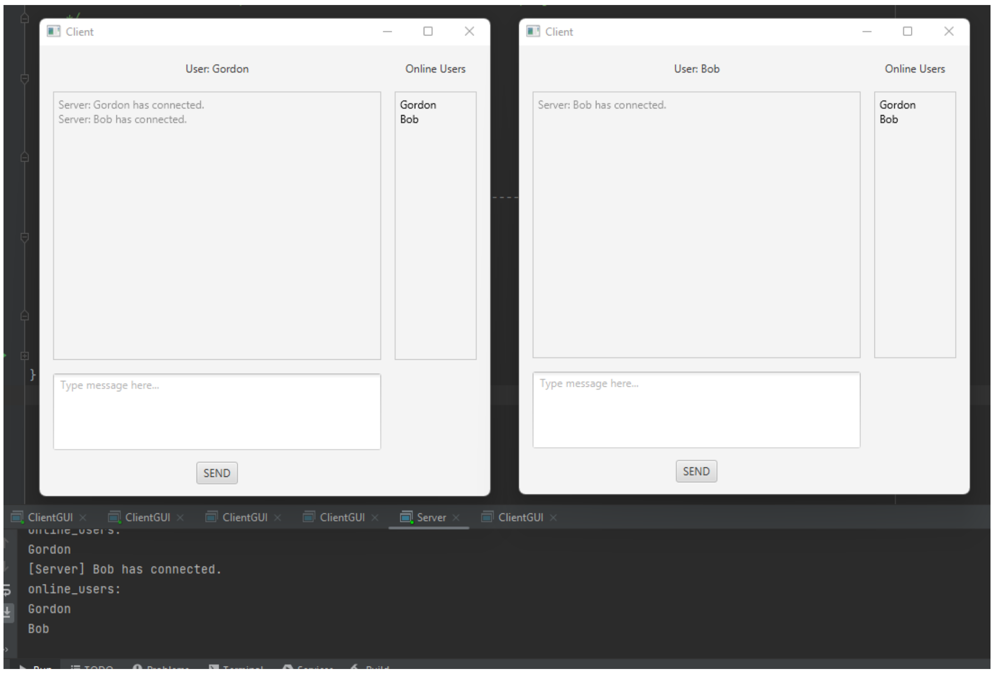
- Findings: The server did not crash. The clients connected to the server one by one. No race conditions or errors were found.

6.3 WORD LIMIT

- Experiment: Find the maximum amount of words that can be sent from one client to another client.
- Theory: Since the data being sent is only text, a massive amount of data would have to be sent in order to cause the server to be overloaded with work or crash.
- Test: Different data sets of varying sizes were used to test if the server would be overloaded or if the server would crash.
- Test Data: The data used for testing is provided by https://www.lipsum.com/, a website that allows us to generate "dummy" text of varying sizes. The text sizes are as follows:
  - 92 words, 602 bytes
  - 540 words, 3644 bytes
  - 921 words, 6271 bytes
  - 1856 words, 12548 bytes
  - 3577 words, 24248 bytes
  - 7203 words, 48311 bytes

  - 13405 words, 90601 bytes
  - The website does not allow larger data sets to be generated.
  - The final data set used was Calculus Eighth Edition by James Steward. Please see figure 6.8 below.

- Findings: the client and server did not struggle to send data of any size. There will be a limit to the amount of data sent, however users communicating through this program are very unlikely to reach it.



Server: Gordon has connected.
Server: Bob has connected.

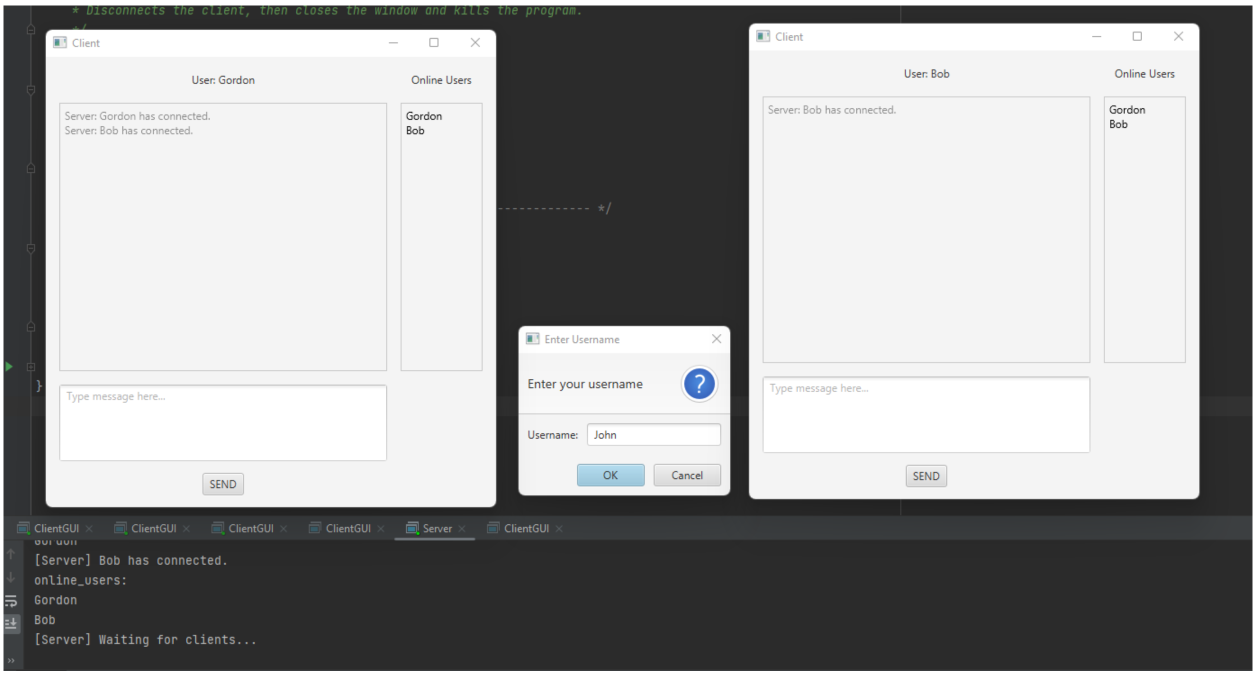
Gordon
Bob

Server: Bob has connected.

Gordon
Bob

```
online_users:
Gordon
[Server] Bob has connected.
online_users:
Gordon
Bob
```

Figure 6.1: Two clients connected to the server.



```
* Disconnects the client, then closes the window and kills the program.
```

Server: Gordon has connected.
Server: Bob has connected.

Gordon
Bob

Enter your username

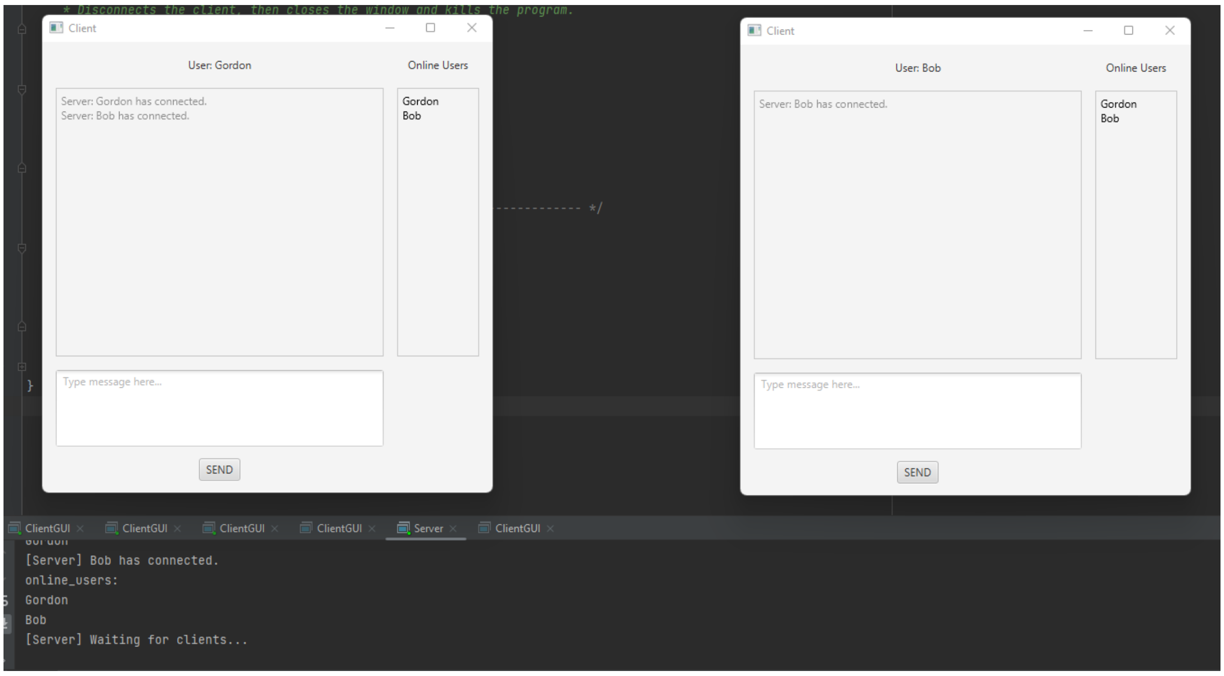
Username: John

Server: Bob has connected.

Gordon
Bob

```
Gordon
[Server] Bob has connected.
online_users:
Gordon
Bob
[Server] Waiting for clients...
```

Figure 6.2: Client "John" attempts to connect to the server.



```
* Disconnects the client, then closes the window and kills the program.
```

```
Server: Gordon has connected.
Server: Bob has connected.
```

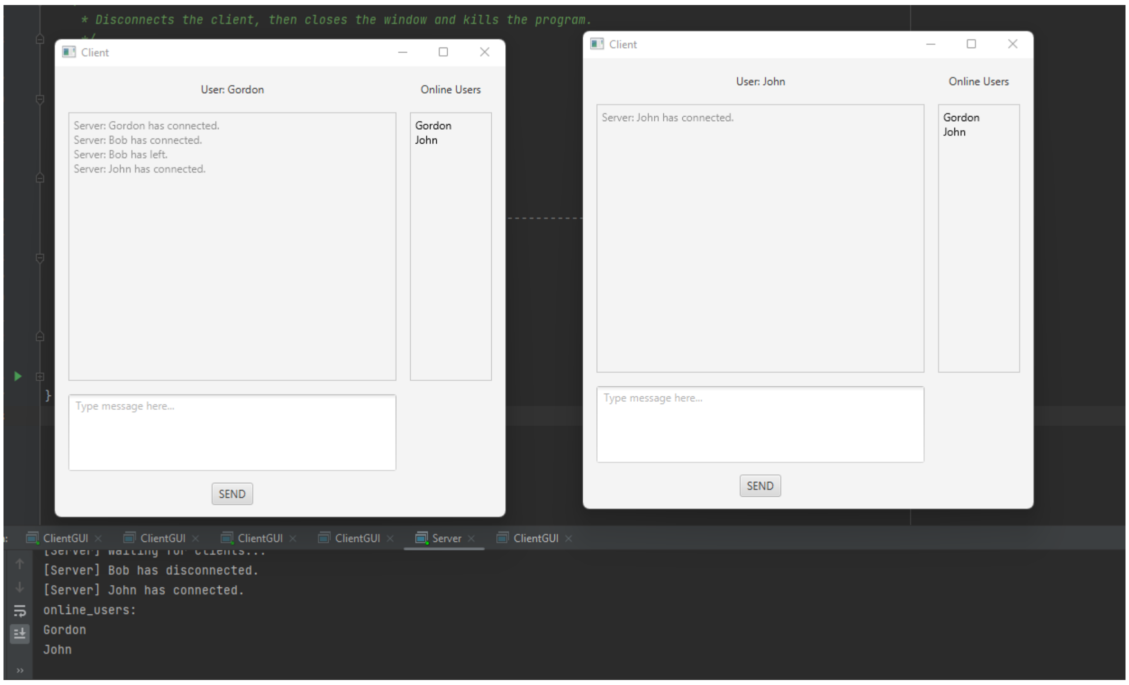
```
Gordon
Bob
```

```
Server: Bob has connected.
```

```
Gordon
Bob
```

```
[Server] Bob has connected.
online_users:
Gordon
Bob
[Server] Waiting for clients...
```

Figure 6.3: Client "John" is unable to connect to the server.



```
* Disconnects the client, then closes the window and kills the program.
```

```
Server: Gordon has connected.
Server: Bob has connected.
Server: Bob has left.
Server: John has connected.
```

```
Gordon
John
```

```
Server: John has connected.
```

```
Gordon
John
```

```
[Server] Bob has disconnected.
[Server] John has connected.
online_users:
Gordon
John
```

Figure 6.4: Once client "Bob" disconnects from the server, client "John" connects to the server

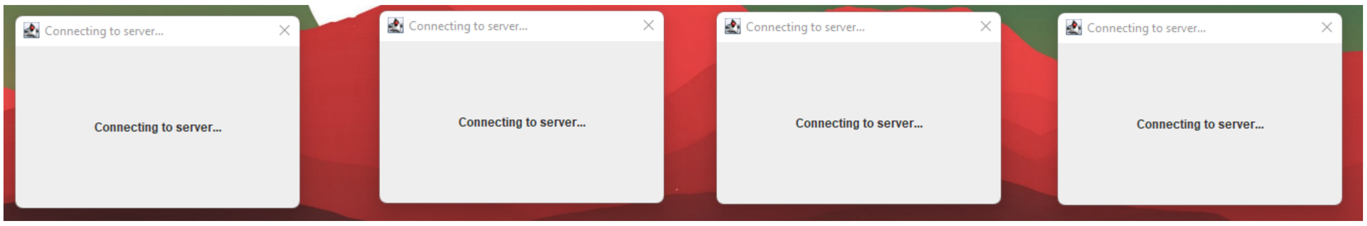


Figure 6.5: Multiple clients attempting to connect to server before server is started.

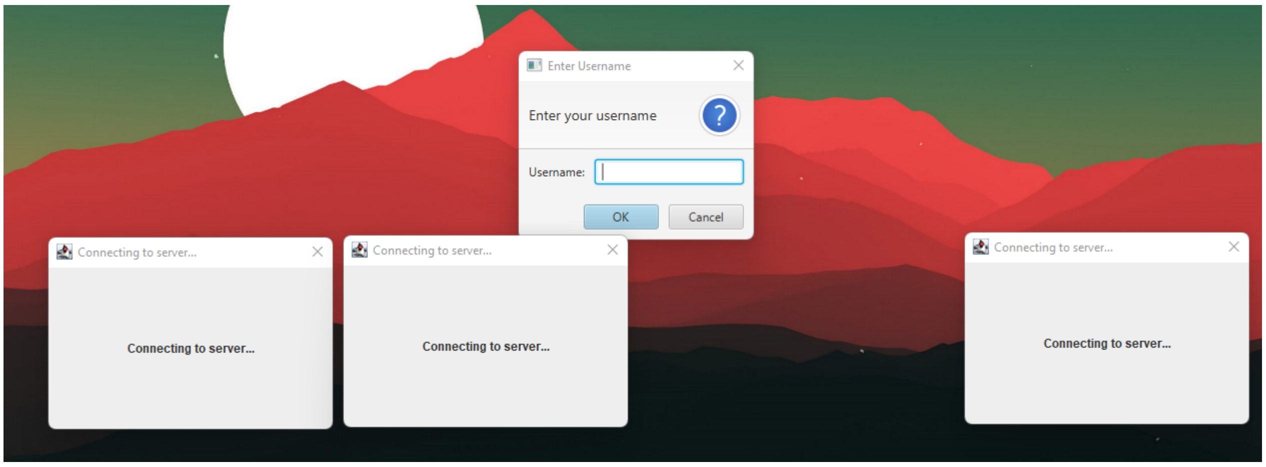


Figure 6.6: When the server is started, clients connect one by one.

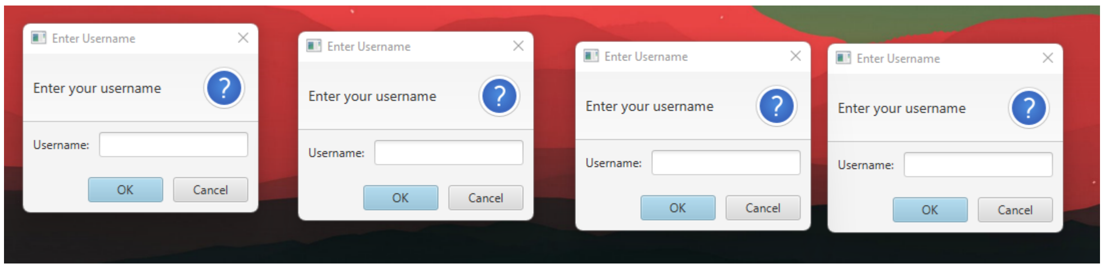
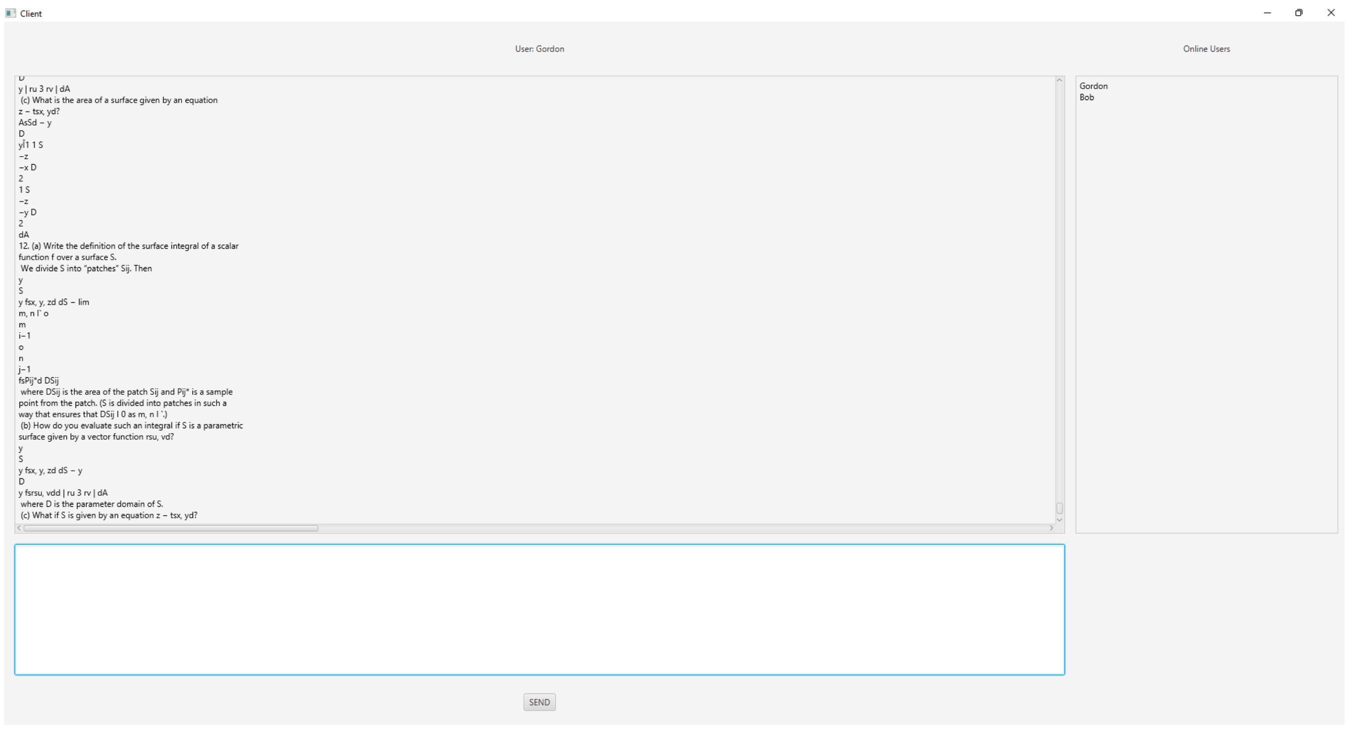


Figure 6.7: All clients have connected to the server and await username input.



```
D
y | ru 3 rv | dA
 (c) What is the area of a surface given by an equation
z − tsx, yd?
AsSd − y
D
yÎ1 1 S
−z
−x D
2
1 S
−z
−y D
2
dA
12. (a) Write the definition of the surface integral of a scalar
function f over a surface S.
 We divide S into "patches" Sij. Then
y
S
y fsx, y, zd dS − lim
m, n l` o
m
i−1
o
n
j−1
fsPij*d DSij
 where DSij is the area of the patch Sij and Pij* is a sample
point from the patch. (S is divided into patches in such a
way that ensures that DSij l 0 as m, n l `.)
 (b) How do you evaluate such an integral if S is a parametric
surface given by a vector function rsu, vd?
y
S
y fsx, y, zd dS − y
D
y fsrsu, vdd | ru 3 rv | dA
 where D is the parameter domain of S.
 (c) What if S is given by an equation z − tsx, yd?
```

Figure 6.8: The final word limit test using Calculus Eighth Edition by James Steward.

7

ISSUES

- The user interface uses JavaFX to create the GUI for the client. In setting up the project on multiple contributors' computers, the problem could arise where the different computers are be running different versions of Java, which would cause the client GUI to crash. The biggest issue was that JavaFX only supplies files for the newest version of Java, meaning that a computer using an older version of the Java SDK could not run the GUI. This issue was fixed by installing IntelliJ which handles the JavaFX and SDK installations.
- When a `ClientHandler` thread is created it must be stored in the `ConcurrentHashMap` using the username of the client as the key. However, since the username of the client connecting is only obtained after the execution of the thread, the thread could not be stored until it was executed. Waiting for the client to send the username would cause the server to block. This was solved by passing the `ConcurrentHashMap` to the thread when it is created, and once the thread is executed and has a unique client username, it adds itself to the `ConcurrentHashMap`.

8

SIGNIFICANT DATA STRUCTURES

8.1 ARRAY BLOCKING QUEUE

- Part of the Java Standard API's `util.concurrent` package, the `ArrayBlockingQueue` is an array based queue data structure that is thread safe.
- If an attempt is made to take an item off the queue while it is empty, it blocks until an item has been enqueued. It also takes care of race conditions when items are enqueued.
- The properties of the `ArrayBlockingQueue` made it easy to manage incoming and outgoing messages in different threads, allowing us to focus on the main logic of the application, and not on preventing race conditions.

8.2 CONCURRENT HASHMAP

- The `ConcurrentHashMap` is a thread safe version of the Java `HashMap`. The `ConcurrentHashMap` only allows one thread to edit the `HashMap` at a time, while allowing multiple threads to read from the `HashMap` all at once.
- Using a `ConcurrentHashMap` to store user threads allowed us to store the threads with out any risk of a race condition or data corruption.

9

DESIGN

Due to the inclusion of both the CocurrentHashMap and the ArrayBlockingQueue, mentioned in section 8, we did not have to worry about thread created errors, such as a race condition. This allowed us to write our code without the use for any blocking systems, as they were built into the data structures that were used.

10

COMPILATION AND EXECUTION

It is assumed that the project will be run on Linux from a Bash shell.

10.1 DEPENDENCIES

In order to compile and run the server and client the following dependencies must be installed and available on the `PATH`:

1. OpenJDK 18.0.2
2. OpenJFX 18.0.2.u2-1
3. Gradle 7.5

10.2 COMPILATION AND EXECUTION

10.2.1 Server

The server can be compiled and executed by following the next steps:

1. From the project root directory `cd` into the `server` directory.
2. Execute the script `run.sh`.

10.2.2 Client

From another terminal window, follow the same steps for the client, but in step 1 `cd` into the `client` directory, and, when executing `run.sh`, supply as argument the IP address or hostname of the machine or network the server is running on.

11

LIBRARIES

- JavaFX
- Java Standard API