Computer Networks CS513

Class Project: Client Server Chat

Submission by:
Josue Contreras

Date Submitted:
7/9/2020

Signature:

Abstract

This report describes the implementation of a simple chat group where users can ‘whisper’ or privately send a message to other connected users by selecting them in the client’s GUI. The chat group is developed in Java 8 using Java Socket Library and is based on a single server - multiple client model. The Java programming language was chosen to develop this chat program as it is simple to run under the Windows operating system. It is important to note that to run this chat implementation the latest release of Java 8 needs to be installed since lambda expressions are used to simplify the readability of the code. Additionally, all the libraries used in this project come packaged with the Java 8 release. A simple GUI implemented with the Java FX library allows users to interact on the client side and select other connected users to send private messages to. The design of this program is modular and special attention was taken to the client handler as it manages the actions sent and received in the input and output streams of each connected client without interrupting the services of the server. The report includes description of the tests and experiments conducted along with their respective test results. A link to the git hub repository is included as well as in the appendix.

Contents

Project Description

The projects main objective was to implement a chat application that handled multiple clients with one running server. The main feature that this chat application provides is a server that can handle multiple connection requests and disconnect requests, direct messaging/whispering between clients, and validation of client usernames. These requests must be handled properly as to not disrupt the running services at the server and client ends. This means that proper testing mitigates these errors as they are caught and handled properly beforehand.

In this chat program a single server handles all the requests from the clients. The multi-service solution implemented is multithreading the clients using a client handler class to direct the proper I/O streams to the right users/recipients as they are sent by the senders. When clients connect or disconnect the server prints out the action taken by the user. Furthermore, the server handle connections and disconnections without disruption of other services as IOExceptions are handled properly on the server side. This allows the server to keep running in the event that a I/O stream becomes disconnected. The only validation done on the server side is to check is the username selected by the client is a duplicate. In the case that it is a duplicate the server does not allows the client to connect and drops the request. The last major service from the server is the connected users list. This list is sent to the client when requested. In this implementation of this chat application the client sends a ‘heartbeat’ with this request and the server sends the connected list to the client/s.

For the client side of this chat program a GUI was designed to enable users to easily interact and navigate the application. This GUI is also event driven with button clicks that enable to send the message to other users that are selected form the connection list. The design of the client GUI will be explained in detail in the ‘Design’ section of this paper. The list of connected user displayed to the client is updated through the heart beat mentioned in the previous paragraph. The validation for duplicate names, as mentioned before, is not done on the client side. But the client side verifies that the username is not black, null, or just a number before sending it to the server for a connection request. This allows for the client to also contribute with a proper username. In the event that the server disconnects the client remains stable and displays in the messaging window that the server has disconnected. With the current implementation this chat application if the server reconnects the client must re-initialize the application to connect again.

The server-client model is well known and various implementations can be found online. This chat application was implemented referencing these sources that are listed in this paper. It is also important to mention that this application is unique as it handles requests differently then other implementations as it uses one thread to receive frames in the client side and another thread to request the connected client list. Since this is a simple chat application with the learning goal of providing insight into practical computer networks and the problems faces when implementing them it does not implement any network or message security. This is only a simple chat application that enables users to chat with each other.

Design

This section describes the design of the chat program system architecture and details the servers and clients design. The functional requirements for they various components of this system are annotated in each section and their implementations are described in the subsections.

System Architecture

Overview

The single server – multiple client model used for this chat application can be implemented in various ways. The implementation used in for this project uses a multithreading approach with a client handler on the sever side. This allows the server to handle consecutive connection requests form clients and handles their respective I/O streams. A manager was also implemented to processes the connected and disconnected clients that are sent to each client. A constants class was established to make communication throughout the program readable and consistent. On the client side a GUI was implemented that allows to send messages when a button event happens. This GUI is also capable of receiving messages and the list of connected clients. The figure below shows the chat application in full. There are three clients in connected to the server that can be seen running in the bottom terminal. The clients are able to successfully communicate with each other through direct messages or 'whispers.'

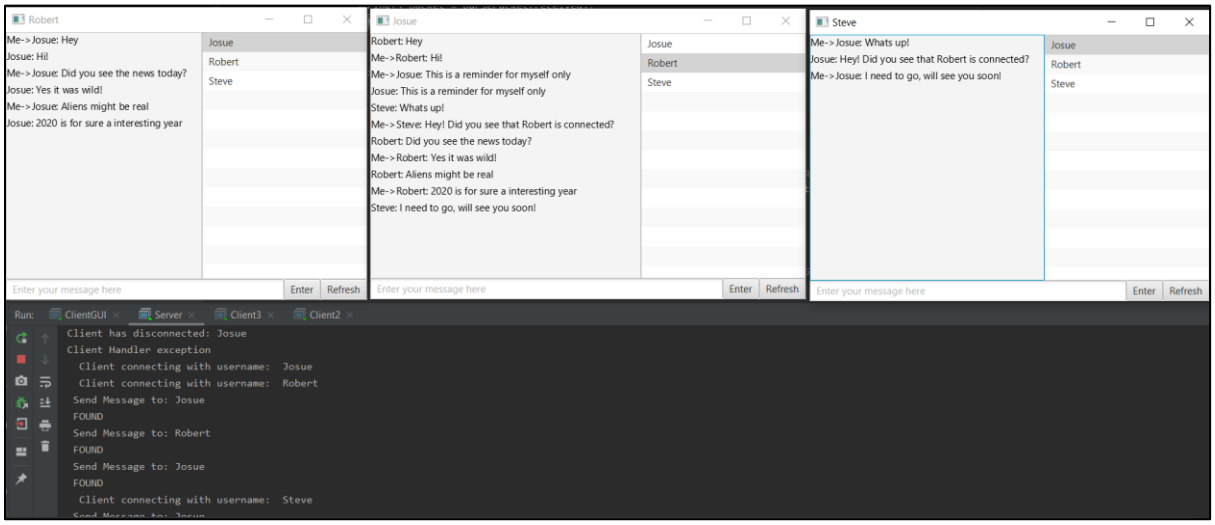
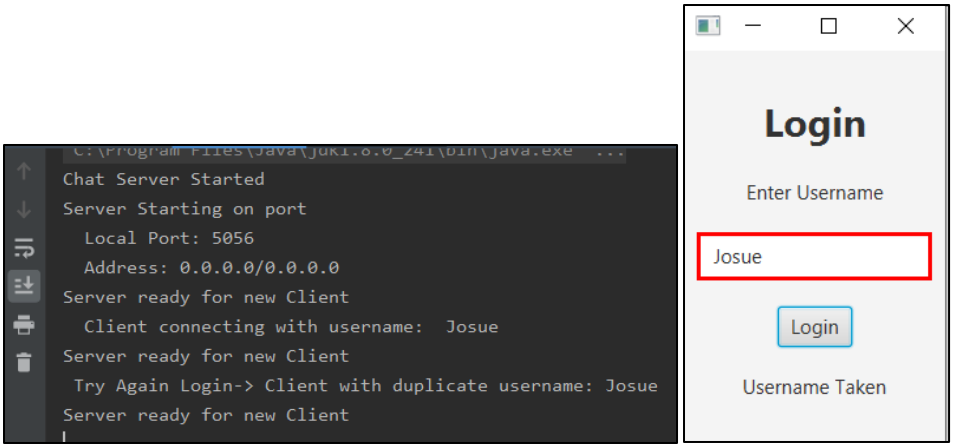


Figure #: Server (bottom) and multiple clients connected (Top)

Server

Server

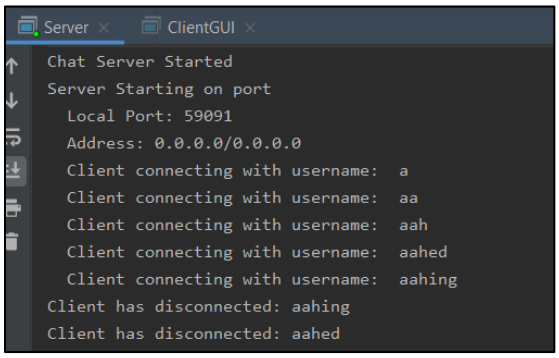
Upon initialization the server initializes its sever socket and start by listening for connection requests on that socket. In the event of a client request the server gets the clients I/O streams and reads the requested username that the client is picking. The server then proceeds to verify if a client that is connected has the same username and if it does the server rejects the connection by sending a duplicate constant and start listening for the next request as seen in the figure below. In the event that it is valid username then the server instantiates a client handler with the username client socket, and I/O streams. This client is also added to the client connected list that is managed by the connected user manager class. The final action by the server is it creates a thread for the client and starts it.



Server-side duplicate username verification

Client Handler

One of the functional requirements for the server is to handle multiple clients that communicate to each other. To accomplish this without disrupting the server a client handler class was implemented. The client handler itself stores all the components of a client upon connection. In this implementation the server does not come with a GUI, but all its processes can be seen in the terminal. The client handler in action can be seen below as connection and disconnect requests are handled between connection requests.



Server handling multiple client connection/disconnection requests

This client handler implements the runnable interface to be called in the server class for a thread. These components are the client's username, socket, I/O streams, and some additional variables. The run() function controls the clients communication with the server. It takes the packets sent by the clients and parses through the information they contain. The first character in the packet is the control ID that tells the server what type of information contained in the packet. This is then processed as a logout, direct message, or connected user list request. With this control ID the server takes the proper action on that packet it is sent to another user as a message or sent back to the client that requested it.

When a message is sent form a sender to receiver the client handler iterates over a linked list of client handler until it either finds the proper client and sends the message through its output stream or it doesn't and it responds back to the sender. In this implementation a linked list is used since this data structure allows dynamic allocation when adding elements of client handles to it. It is also one of the simplest data structures to use and since the premises of the project was not scalability is suffices the proper working if this application. Although the programs modularity could allow for more scalable data structure to be implemented.

User Manager

The server should also be able to keep track of the connected and disconnected users. This is another functional requirement of the system that is accomplished with a connected user manager class. This class not only enables the server to easily add or delete a connected client, but it also helps the server verify that a user does not have a duplicate name. It also provides a method that allows to send a string of characters to the client when requested. The figure below shows the client manager in action by sending the connected clients and disconnected clients.

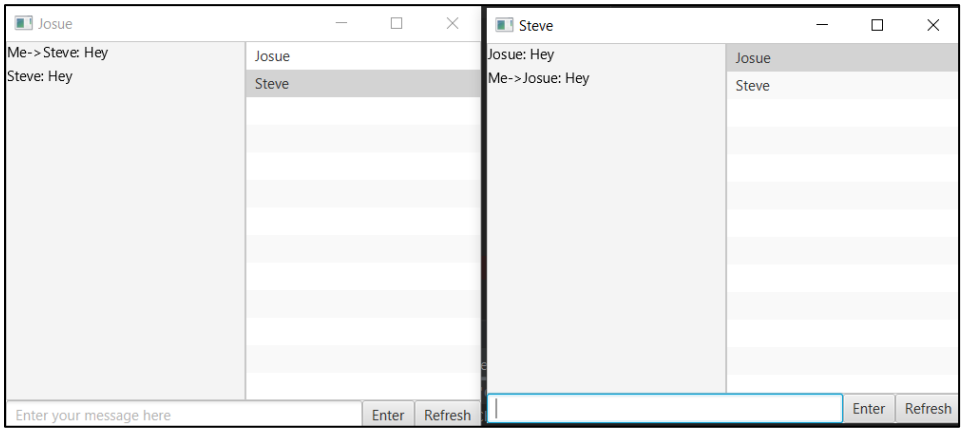
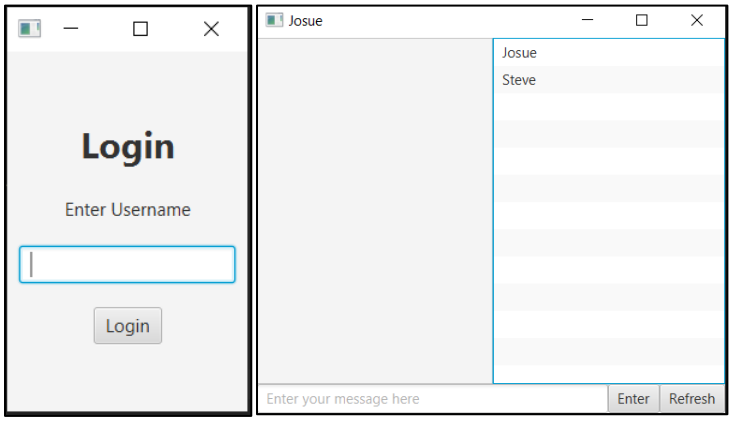


Figure #: Shows connection list over an interval of time or when refresh button is pressed

Client

Overview

In this implementation the client functional requirements are the following: displays online users, displays connection/disconnection actions, displays messages, can receive messages while typing, and client can disconnect without disrupting the server. To accomplish the GUI seen below was implemented in Java FX along side a client class that instantiated a client with its I/O streams and helper functions. This client implementation contains two threads with different priorities. The first thread receives the messages sent form the server to the client and process the packets. The second (lower priority) thread sends a heartbeat to the server given an interval of time. This heartbeat is a connected users list request that allows to dynamically update the list view of online users on the GUI.



Client login and message windows

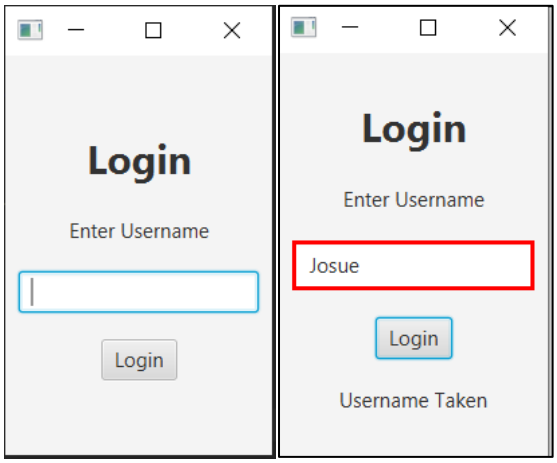
Threads

As mentioned before there are two threads for the client entity. The first one receives messages and processes them. It uses a similar approach to the how the server processes the packet by using the system constants as control ID to determine what type of packet was received. This is then processed as a message form a sender, a list of connected users, or a user that was requested was not found.

The second thread is a heartbeat and is set to lower priority. It is called a heartbeat because it is activates given a set interval time like a heart pumps blood. In this case the heartbeat is a request to the server for the connected client list. In this way the user connected list on the client side can be update dynamically without the user having to remember to refresh. The capability to refresh the list is possible as if the interval of the heartbeat is increased. Since this is a lower priority thread it is capable of being interrupted by the receiver thread which is the desirable effect as receiving a message is more important than a connected list.

GUI Application

To fulfill the display requirements on the client side a GUI was implemented in Java FX. This GUI consists of two windows: login and message windows. The login window is the first to show when the program is initialized as seen in the figure below. The window also shows various messages when an incorrect username is passed or the server is offline.



Client login window

The message window shows when the server accepts the username passed at the login screen. As seen in the figure below the message window is composed of a view of the direct messages and actions, a text field, enter and refresh buttons, and a connected list view. The direct messages in the window are denoted with a "->" that means that the message is only relayed to the one client and it is not broadcasted. Actions are also displayed in the same window as the messages but can be differentiated because they are bold. The connected user list on the right is dynamically updated as more users connect or disconnect. It is also possible to refresh this list using the refresh button. Finally, one of the functional requirements for the client was that is should be able to receive messages and actions while typing a message. This was easily accomplished by using this GUI with an event handler that listens for the push of the enter button.

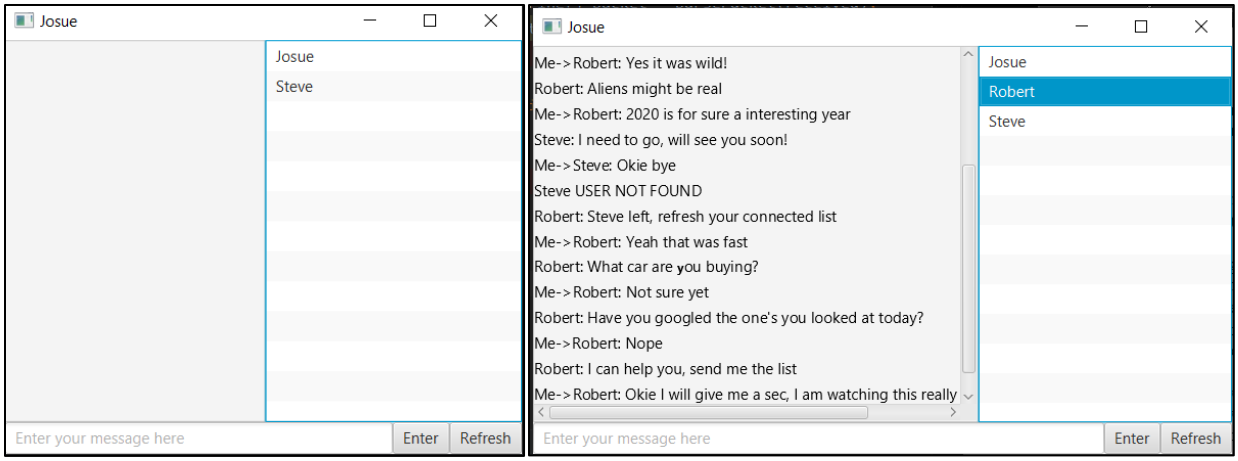


Figure #: Client message window

Testing and Validation

Testing of a client server model system allows to prevent errors that may affect the functionality of the program in fatal ways. This section contains the tests conducted on the units of code and the system as a whole. When it comes to testing the functionality of such system, the server and client model, we must consider the unit testing on components and system testing on I/O streams of the client and server. Now since the client possess a GUI the functionality was tested based on a manual event-based verification approach that follows the functional requirements of the client. For the sever two tests where performed, functional test and performance tests. The functional test performed on the server allowed to verify the I/O streams and the proper response to various requests. The performance tests used where destructive; this means that the test allowed to record the systems average limits before failure. This is important to know in a system like this to mitigate the risk of system reaching such events like described later in this section.

Unit Testing

User Manager and Client

Unit tests allowed to test the induvial components in this object-oriented program. Two types of Junit tests where preformed to test the system. The first set of Junit tests where developed to be tested on a inactive server. The classes that these tests pertain to are the connected user manager and client classes. The second type of Junit tests where for a server that was active. This pertained more to the client and I/O streams form the server. These tests where periodically preformed to make sure changes to the client or manager did not affect the expected outputs. The figures below show these two Junit files running successfully. The Junit test where simple in this case as testing the server proved to be a challenge, hence more manual I/O based verification was used to test the active server as a whole as seen in the next sections.

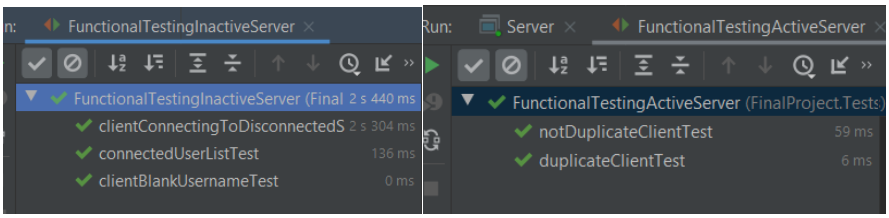


Figure #: Junit Tests for client-server system

Login Username Verification Window

The login window for the client GUI was tested on its own since verification happens on the client side and on the server side. Therefore, a more manual approach to test was taken. This approach outlined all the possible inputs and GUI respective response to duplicate names, invalid username (number or no input), or no server connected as seen in the figure below.

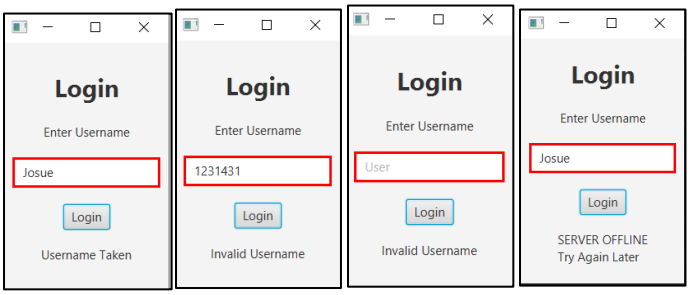


Figure #: Server and client username GUI verification unit test

System Testing

I/O Stream One Client and One Server

As mentioned above a more manual testing approach was taken to validate the correct response of the server to the client. In this section test are conducted as a whole system since the output to the client given a response is monitored. For the first test the I/O streams of one client and one server are monitored. One of the tests preformed was communication to a disconnected user as seen in the figure below. Two random names where added to the list view and the server was able to determine/display that this user was not connected to it. It then responded to the client with a "user not found" action.

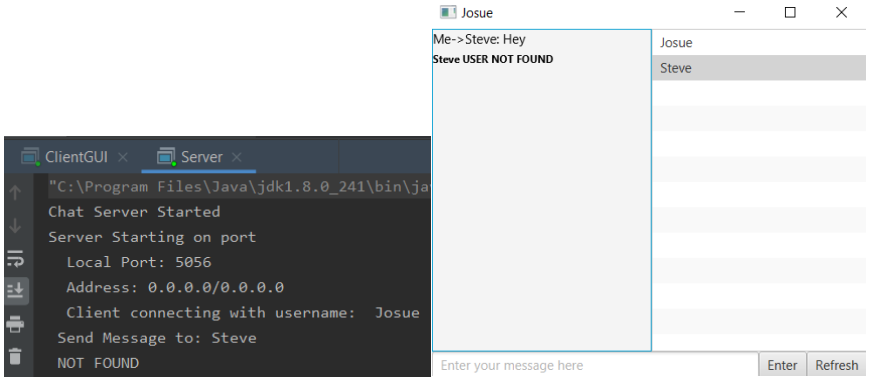


Figure #: Send to unconnected user

I/O Stream Multiple Clients and One Server

For these next tests multiple clients connected to the server. The figure below shows the direct messages or 'whispers' between two users. Note that a message can also be sent to sender. In this implementation a direct message can is denoted with the arrow "->" form the sender's side. At the receivers end the message appears with only the name.

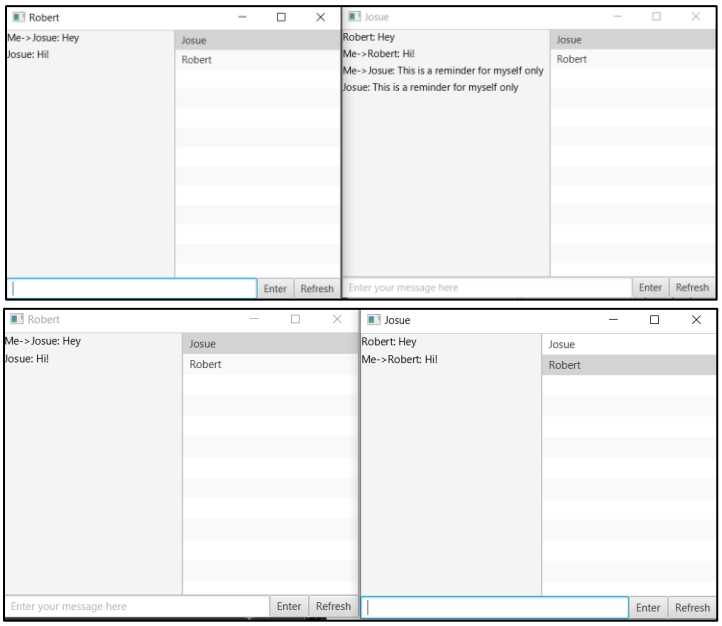


Figure #: Send direct message ('whisper') to user, send to self, receive messages

In this second test the of multiple clients connected to one server direct messages can be seen more clearly between three users, notice the "->" form sender to receiver. This test also proves that the connected user list updates as more users are connected to the server. Finally, these system tests also demonstrate the proper packet structure that is passed from sender to server to receiver. This is a result of the system constants.

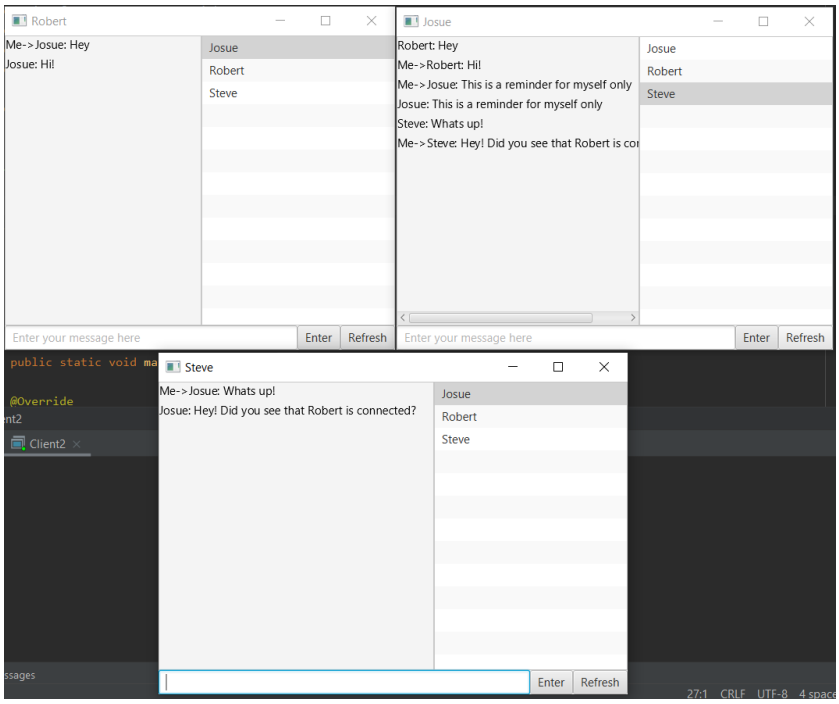


Figure #: Properly see all messages, refresh on connected users.

Performance Testing

Testing Multiple Clients until Failure

The first performance test designed was to determine the maximum number of clients that could be connected before the client-server system would start to fail. The idea behind this test is to run a method that connects N number of clients to the server. Then the last client would try to connect and communicate with one of the clients connected. If successful, the N number of clients would increase on the next run until the system failed to support the last client connected of its complete functionality. For each run the client connected list would be refreshed to see if the proper number of clients are displayed.

Once the system started to fail the test will start to decrease and increase the number of clients based on their success or failure. Once it finds the intersection of failure and success this test determines the max number of clients that can be connected and still properly communicate with each other. In this case as seen in the Figure below the max number of connected users is 6473 clients (it says 6472 clients below, but a 1 was connected after to test the system).

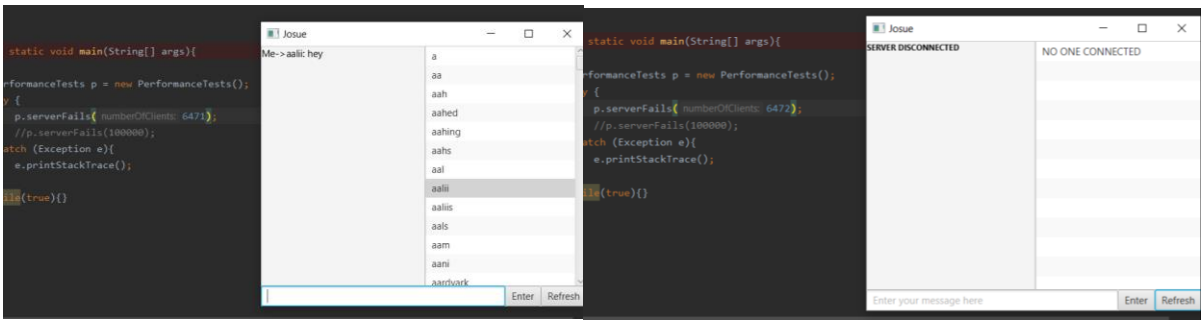


Figure #: Multiple Clients Bottleneck (6471 Clients on left, 6472 Clients on right)

This error in the system might be the result of threads created for such a vast number of clients. Another theory for such event is the length of the client list that is passed as a string from the server to the client when requested. The max length that a java string can have is 2147483647 the output string when the system fails is 59063. The most probable cause might be a combination of the heartbeat thread that requests the connected list of users and the length of the string. A better approach to send the connected list and maybe mitigate this error is split the list in sections instead as a whole. This might allow the program to increase its users connected list without disconnecting them when the 6472 connected clients is reached. Another solution would instead of using the heartbeat to request for the connected client list us the refresh button. This way the user would have to request manually the connected list when they need to and would use less processing on the server side to handle these consecutive requests. This test proved useful as it determines the max characteristics/capabilities this system can handle. More of this test can be found in the Appendix of this paper under "Additional Performance Tests."

Consecutive Client Connection Requests until Failure

The second performance test for the server was designed to determine the server's max capability to accept consecutive client connection requests. For this test 100,000 clients tried to connect sequentially to the server. This test was run 10 times to get the average failure point of the server. Like expected the server never reached 100,000 clients but an average of 14596.4 clients before the Server cannot accept connection requests. The results for the 10 tests are found in the Appendix under "Consecutive Client Connection Test Results."

One interesting result when testing appeared by connecting 100 clients sequentially and disconnecting all of them at the same time. The server could not handle all this disconnects to the point that it had to terminate. The exception stack trace pointed to the user manager as it seemed to be trying to delete all the disconnects to fast. This is important since a better data structures than a linked list can be allocated to manage the connected users. Since the linked list must be traversed form beginning to end this means that when such a massive event of user requests to disconnect happens it cannot keep up with it and a different method in the manger could de implemented.

Unsuccessful Run Time Performance Test Implementation

Overall the testing of such system proved to be a challenge when it came to performance testing. One of the tests that was designed and implemented but did not work in the end determined the runtime performance of the system from sender to receiver. During the implementation of such test multithreading became a big challenge to test this type this characteristic of the system. The test was designed to connect a incremental amount of clients to the system for a set number or runs. The means that each run would start with one client for example and the next would add another, so the system would have two and so on

as the run number increased. Then to test the performance two clients would be picked and a message would be sent from client to the other. The time it took for the other client receive would be recorded and, in this way, the average runtime for each client in the system in that run would be taken. This would ideally yield results of the bottle necks of the system and the number of clients that the system could handle before seeing latency between messages. Now when implementing this system, the nature of the threads for the clients yielded no results as the message would arrive in various times that where not clocked by the change in time. The approach described above allowed to determine this bottleneck but did not take into consideration the messaging delay that would be seen if used by multiple users at once.

Future Development

The main current data structure for this implementation is a Linked List. For future implementation a better data structures that is more scalable should be integrated. This would allow for more users to connect and would help the servers performance. It could also allow to expand the capabilities of how connected user are managed.

Looking into a more complete chat application a user login that requires a password and allows to save the conversations previously conducted could be implemented. This would allow the system to keep track on users and reconnect them if the server fails. This would also mean that more work would be done in the backend as a database would have to be created to store this information

The current implementation does not consider security. In practice security is important and a secure socket could be a further improvement with some king of cryptography. This capability would make users more likely to use a system like this that keeps their personal information private.

From the GUI perspective on the client side a window change could be implemented for each conversation with the various users. This would replace the arrow that denotes the direct message in the current implementation. Furthermore, this would also allow for the creation of group chats that could be easily done by selecting multiple people at the same time.

Conclusion

Overall this project was successful as it implemented all the functional requirements specified. It also accomplished its objective to provide more insight into practical computer networks and the problems faces when implementing them. The test results on this implementation of the server-client model proved that failures are not always fatal and can be overcome. This is true in practice as not all failures can be caught, but they can be mitigated. This was also the main reason that the performance test where developed for this system as this allowed to characterize its systems limits. Hence in the improbable event that this implementation should be applied to a system used real people it could be managed correctly. More development must be done to this system as it has proven to not be able to withhold a larger number of users. Furthermore, more testing could be done to such system that is beyond this classes course material.

References

Chailloux, Emmanuel, et al. Client-Server, caml.inria.fr/pub/docs/oreilly-book/html/book-ora187.html.

"Client Handler." The Berkeley Telemonitoring Project, 2020, telemonitoring.berkeley.edu/tutorials-index/servers/client-handler/.

Gupta, Yogindernath Gupta. "Client Server Testing." Software Testing Genius, 8 Sept. 2018, www.softwaretestinggenius.com/client-server-testing/.

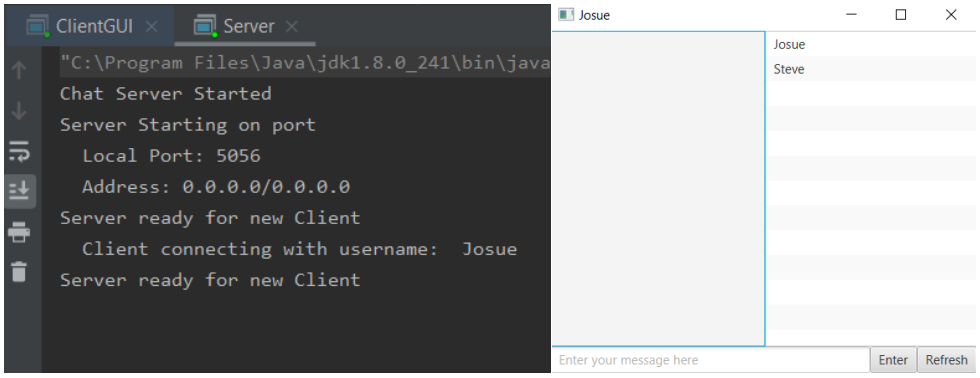
Mahrsee, Rishabh. "Introducing Threads in Socket Programming in Java." GeeksforGeeks, 8 Feb. 2018, www.geeksforgeeks.org/introducing-threads-socket-programming-java/.

Ray. "Java Socket Programming Examples." Javanetexamples, cs.lmu.edu/~ray/notes/javanetexamples/.

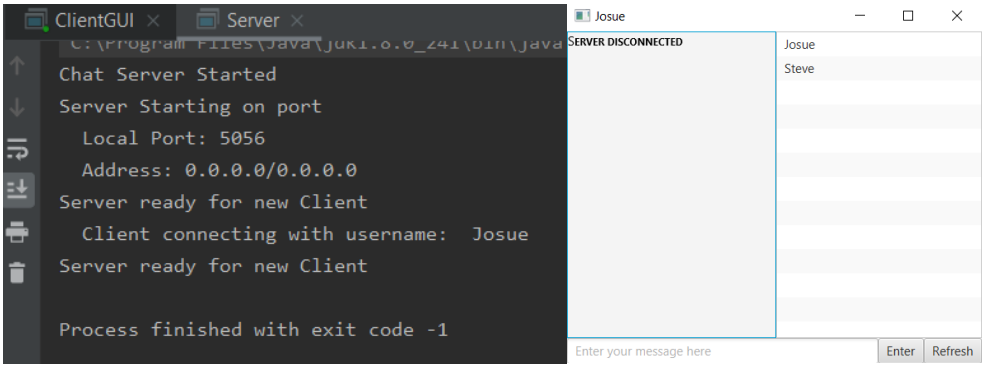
Appendices

Additional Functional Tests

Client connected to server and server goes down



Sever connected to one client



Server failed while connected client

Unexpected Client Connection After Client Connection Request

Testing the client connection with no username input threw an unexpected exception that was caught in the testing phase of this project.

```
// -> Create a ClientHandler Object
try {
    clientUsername = input.readUTF();
}catch (IOException e){
    System.out.println("Client Connection Failed");
    clientConnectionS = false;
}
```

Additional Performance Tests

Consecutive Client Connection Test Results

```
437 Total Clients Connected 14500
438 Connection Successful: 14500.0
439 Failed Connects: 0.0
440 Total Clients Connected 14600
441 Connection Successful: 14600.0
442 Failed Connects: 0.0
443 Total Clients Connected 14700
444 Connection Successful: 14634.0
445 Failed Connects: 66.0
446 Total Clients Connected 14800
447 Connection Successful: 14635.0
448 Failed Connects: 165.0
449 Total Clients Connected 14900
450 Connection Successful: 14635.0
451 Failed Connects: 265.0
```

```
437 Total Clients Connected 14500
438 Connection Successful: 14500.0
439 Failed Connects: 0.0
440 Total Clients Connected 14600
441 Connection Successful: 14600.0
442 Failed Connects: 0.0
443 Total Clients Connected 14700
444 Connection Successful: 14634.0
445 Failed Connects: 66.0
446 Total Clients Connected 14800
447 Connection Successful: 14634.0
448 Failed Connects: 166.0
449 Total Clients Connected 14900
450 Connection Successful: 14634.0
451 Failed Connects: 266.0
```

```
437 Total Clients Connected 14500
438 Connection Successful: 14500.0
439 Failed Connects: 0.0
440 Total Clients Connected 14600
441 Connection Successful: 14600.0
442 Failed Connects: 0.0
443 Total Clients Connected 14700
444 Connection Successful: 14628.0
445 Failed Connects: 72.0
446 Total Clients Connected 14800
447 Connection Successful: 14628.0
448 Failed Connects: 172.0
```

```
435 Connection Successful: 14400.0
436 Failed Connects: 0.0
437 Total Clients Connected 14500
438 Connection Successful: 14500.0
439 Failed Connects: 0.0
440 Total Clients Connected 14600
441 Connection Successful: 14564.0
442 Failed Connects: 36.0
443 Total Clients Connected 14700
444 Connection Successful: 14564.0
445 Failed Connects: 136.0
```

```
440 Total Clients Connected 14600
441 Connection Successful: 14600.0
442 Failed Connects: 0.0
443 Total Clients Connected 14700
444 Connection Successful: 14609.0
445 Failed Connects: 91.0
446 Total Clients Connected 14800
447 Connection Successful: 14609.0
448 Failed Connects: 191.0
```

```
440 Total Clients Connected 14600
441 Connection Successful: 14600.0
442 Failed Connects: 0.0
443 Total Clients Connected 14700
444 Connection Successful: 14618.0
445 Failed Connects: 82.0
446 Total Clients Connected 14800
447 Connection Successful: 14618.0
448 Failed Connects: 182.0
```

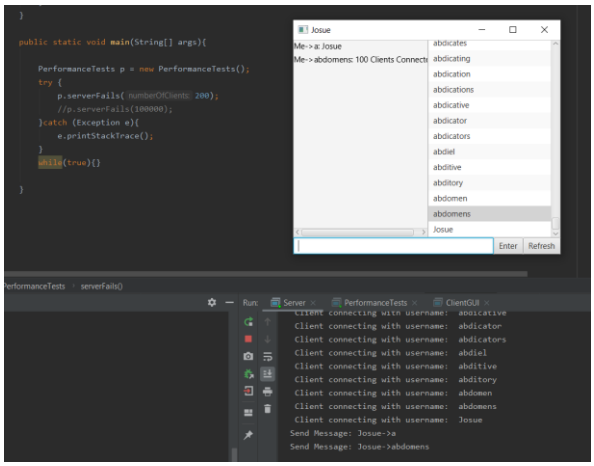
```
440 Total Clients Connected 14600
441 Connection Successful: 14600.0
442 Failed Connects: 0.0
443 Total Clients Connected 14700
444 Connection Successful: 14619.0
445 Failed Connects: 81.0
446 Total Clients Connected 14800
447 Connection Successful: 14619.0
448 Failed Connects: 181.0
```

```
434 Total Clients Connected 14400
435 Connection Successful: 14400.0
436 Failed Connects: 0.0
437 Total Clients Connected 14500
438 Connection Successful: 14420.0
439 Failed Connects: 80.0
440 Total Clients Connected 14600
441 Connection Successful: 14420.0
442 Failed Connects: 180.0
```

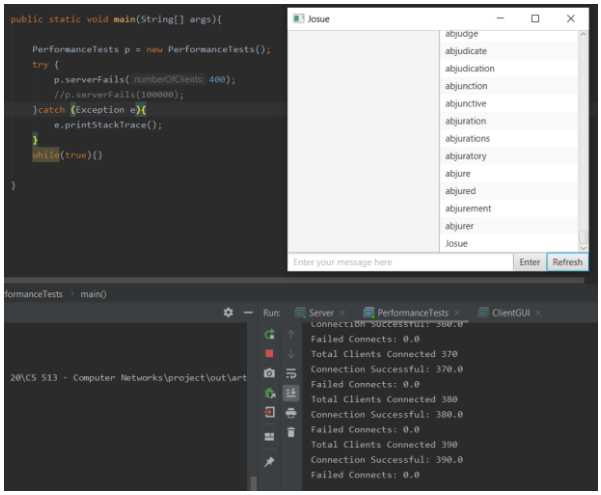
```
440 Total Clients Connected 14600
441 Connection Successful: 14600.0
442 Failed Connects: 0.0
443 Total Clients Connected 14700
444 Connection Successful: 14616.0
445 Failed Connects: 84.0
446 Total Clients Connected 14800
447 Connection Successful: 14616.0
448 Failed Connects: 184.0
```

```
440 Total Clients Connected 14600
441 Connection Successful: 14600.0
442 Failed Connects: 0.0
443 Total Clients Connected 14700
444 Connection Successful: 14622.0
445 Failed Connects: 78.0
446 Total Clients Connected 14800
447 Connection Successful: 14622.0
448 Failed Connects: 178.0
```

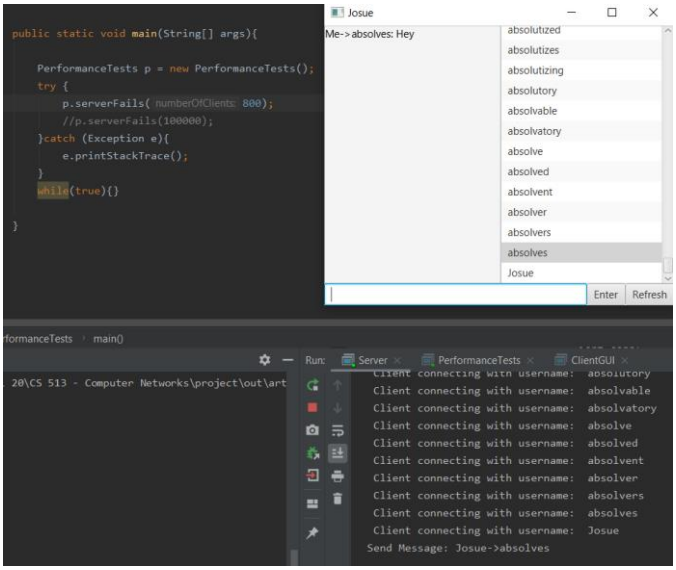
Testing with Multiple Clients until Failure



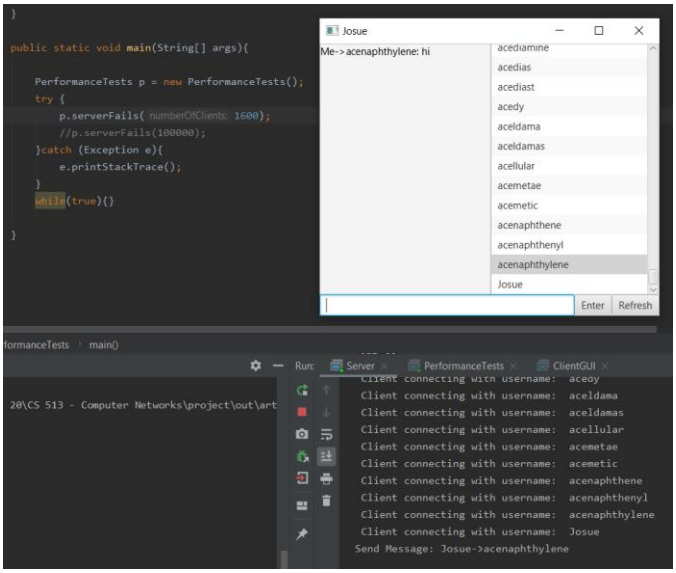
200 Clients connected system successes



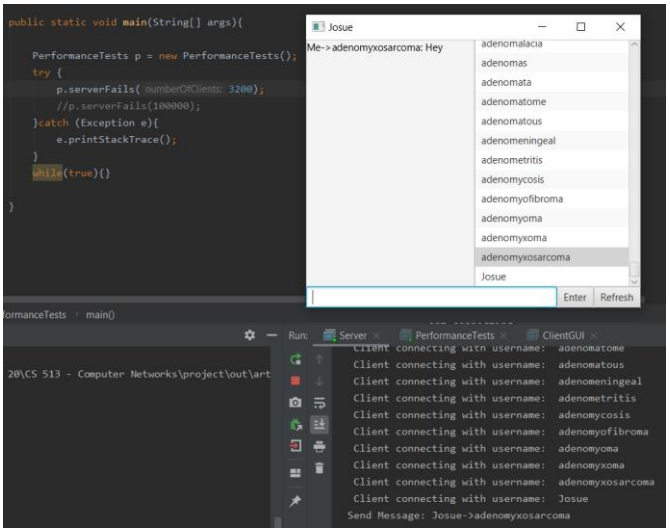
400 Clients connected system successes



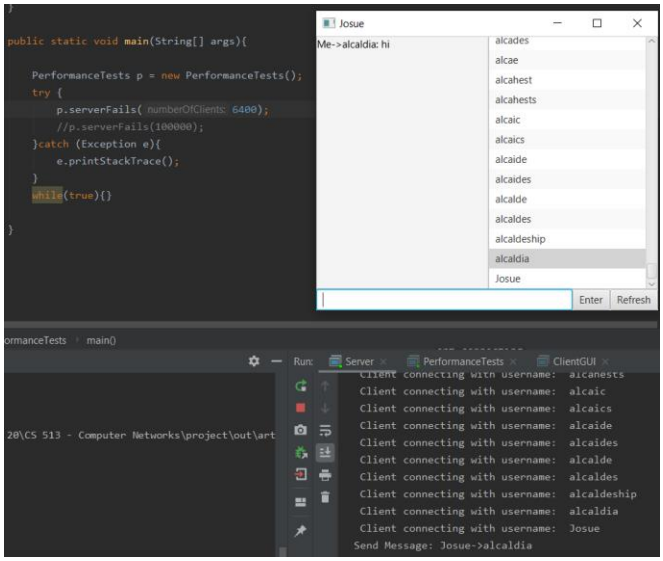
800 Clients connected system successes



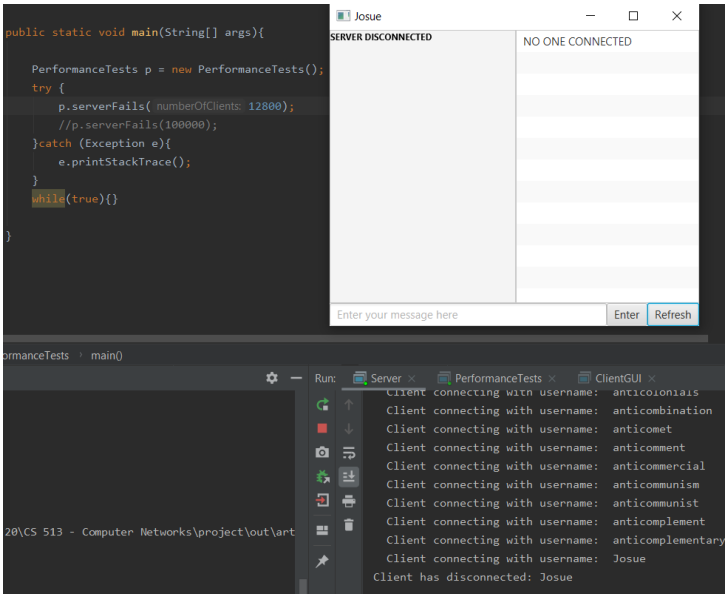
1600 Clients connected system successes



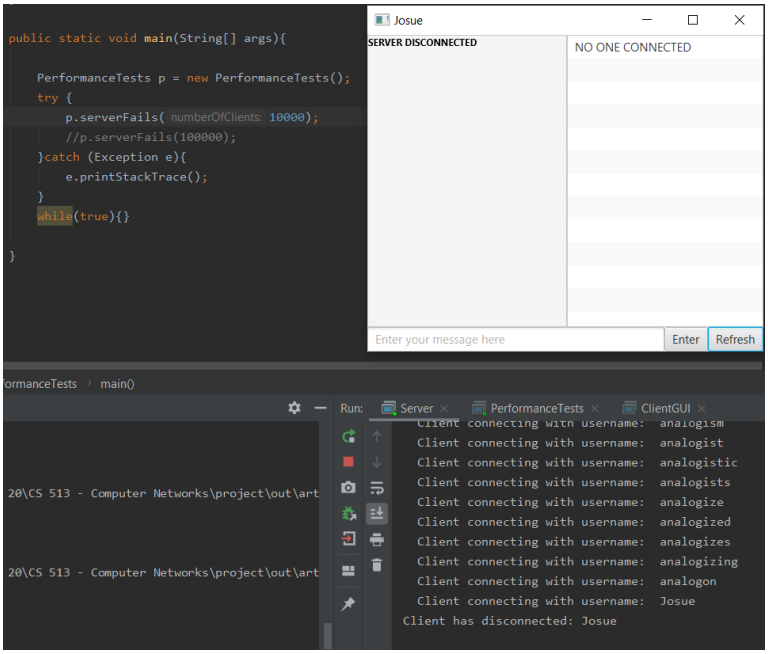
3200 Clients connected system successes



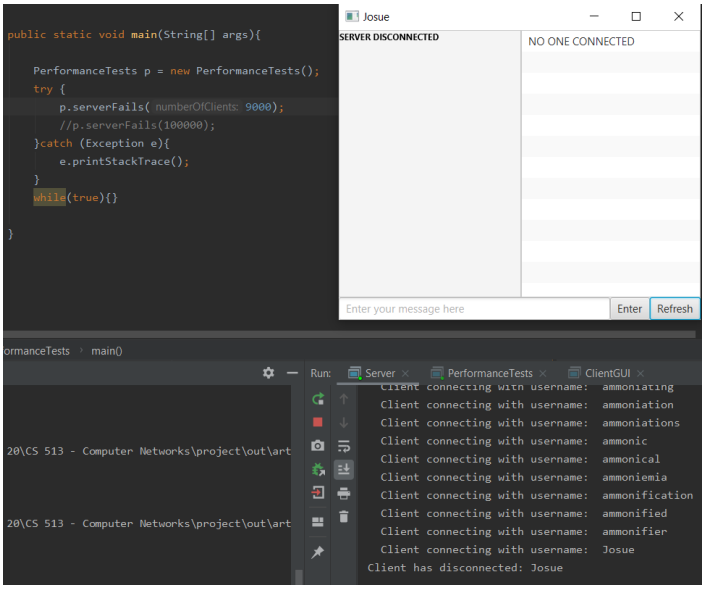
6400 Clients connected system successes



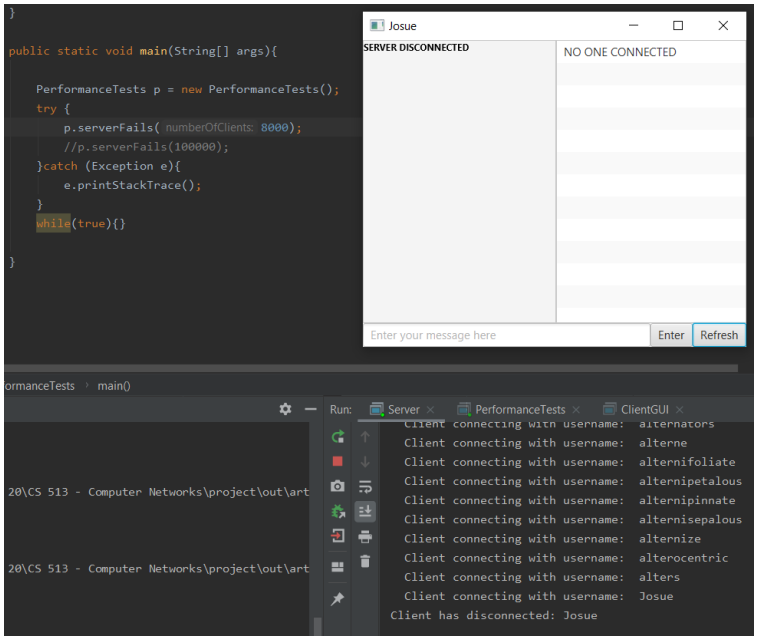
12800 Clients connected system failure



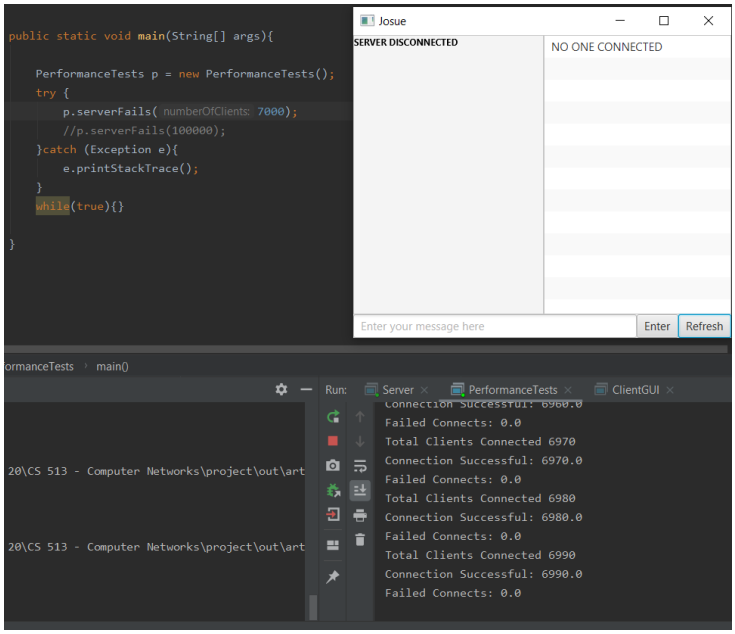
10000 Clients connected system failure



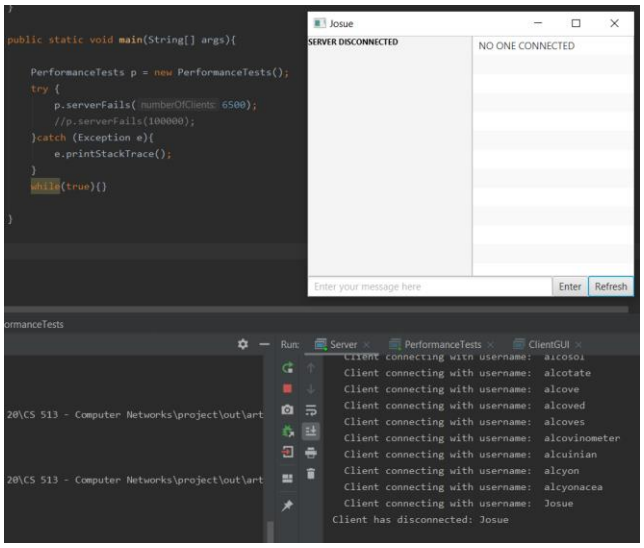
9000 Clients connected system failure



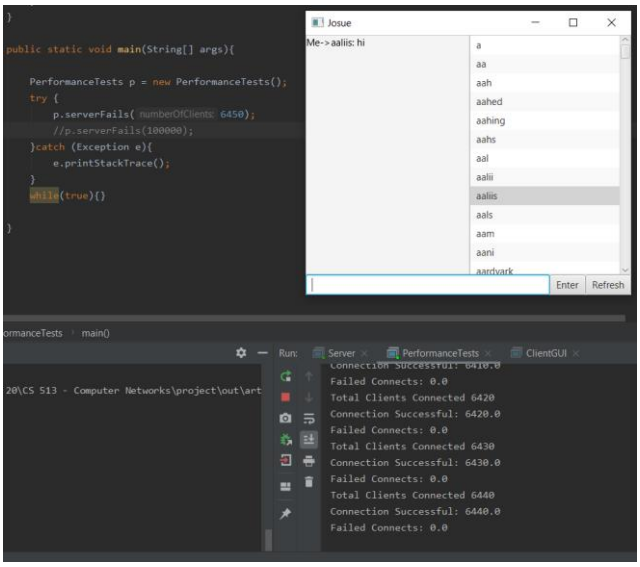
8000 Clients connected system failure



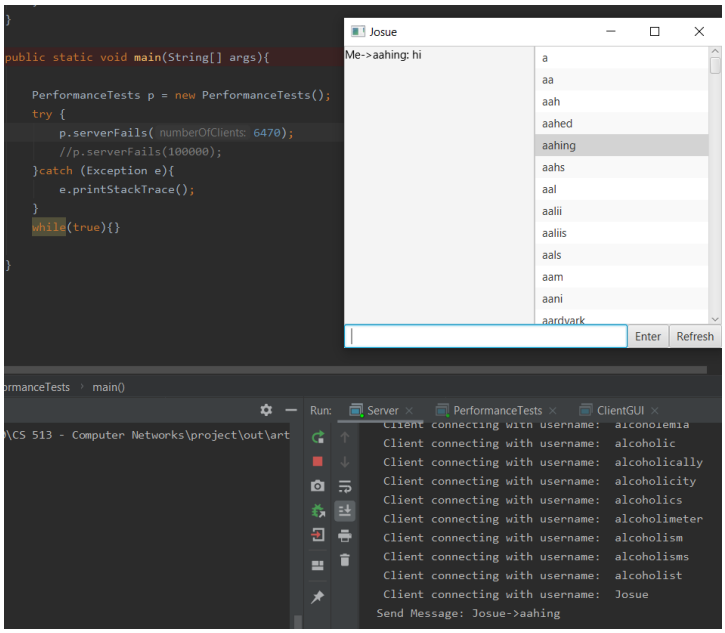
7000 Clients connected system failure



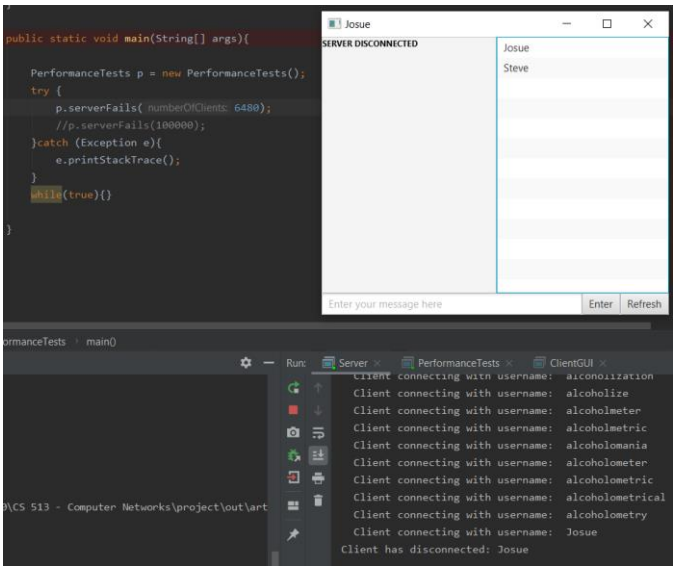
6500 Clients connected system failure



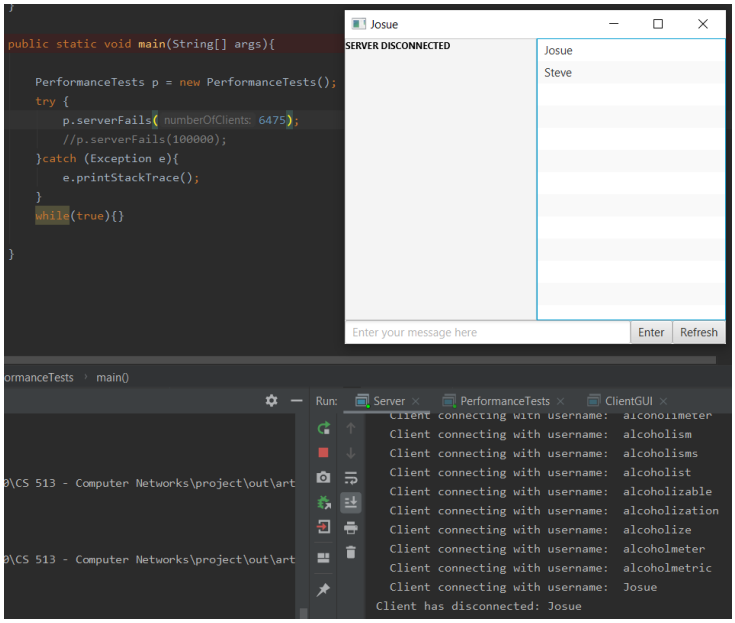
6450 Clients connected system success



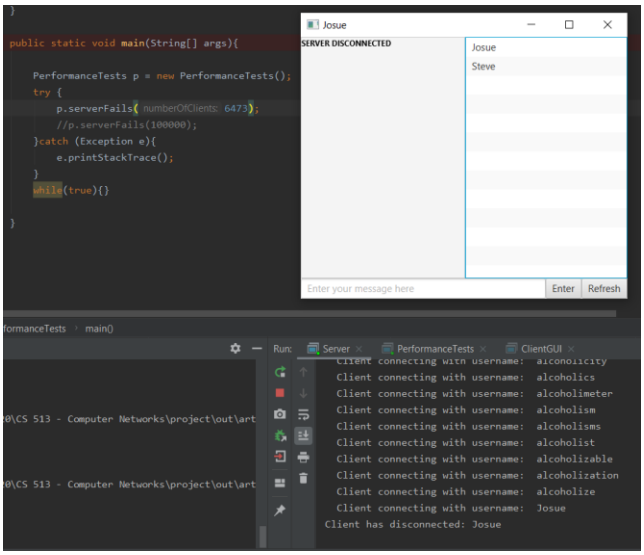
6470 Clients connected system success



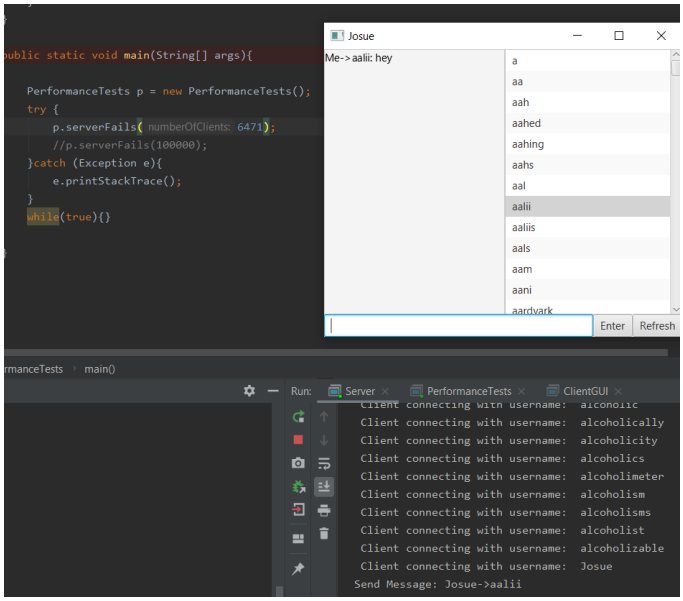
6480 Clients connected system failure



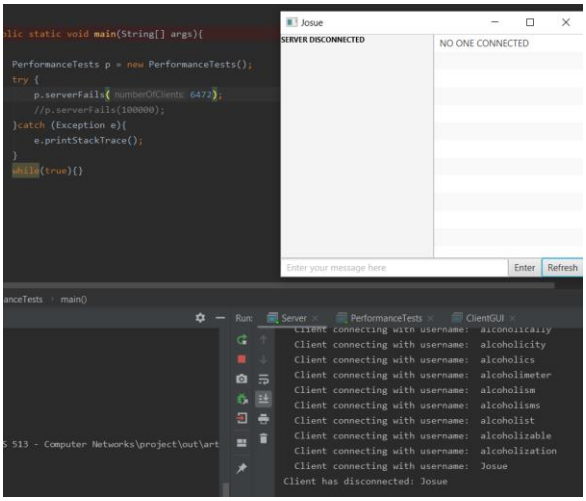
6475 Clients connected system failure



6473 Clients connected system failure



6471 Clients connected system success



6472 Clients connected system failure