How to run our chat server

1. First open up the server in the terminal using the following command:

javac Server.java && java Server <port>

We did the testing on port numbers between 4000 and 9000 so feel free to use one of those.

2. Next, open a client using the following command:

javac Client.java && java Client <host> <port>

The host can be your local device ip address which you can find by typing ifconfig in the terminal and using the value shown in inet at lo0.

3. You can open multiple clients in various terminal windows to test communication between the clients.

We already have four users created in the system – Alice, Sadie, Robbie, and Charlie. Alice, Sadie, and Charlie all have unread messages. You can use those for testing if you would like.

User Actions Supported by Our Server

Whenever a client connection is opened, the server sends the following message after a welcome:

`Please type LOGIN if you already have an account or CREATE to make a new one. You can also enter QUIT to quit the program"`

1. When a user writes LOGIN, they are prompted to enter a username. An error message is displayed if the username is not recognized by the server. A welcome message is displayed on successful login.
2. When a user writes CREATE, they are prompted to enter a unique username. An error message is displayed if the username already exists. A welcome message is displayed on successful creation of the account. Note the user is logged in now and is active.
3. When a user writes QUIT, the socket closes and the user can no longer interact with the client server chat system. Note, there is a small bug here due to a threading issue that we unfortunately could not find a solution for in the given timeframe. The functionality still works, but the server sends one additional message that is output to the client.

Once the user is logged in, our server supports further actions from the user. These are listed below:

4. USERS: Typing USERS prints the list of all usernames stored in the server backend.
5. DELETE: Typing deletes the account of the current user and closes the socket. However, if the user has unread messages, the system alerts them and gives an option to not delete the account. In this case they would then have to still use the command to view their unread messages. Note there is a tiny bug here similar to that of QUIT, where there are issues with the threading, so an exception is printed. However, the backend functionality still works as expected.
6. HELP: Typing HELP provides users the list of all instructions that they can perform.
7. UNREAD: Typing UNREAD checks to see if the user has unread messages and either alerts them that they do not or prints all of the unread messages on separate lines. The unread messages are then deleted from the "database" (i.e. text file storing them).
8. @username <message>: Finally, our chat application allows users to send a message to a specific username using this syntax. If the recipient user is active, the message is delivered to the recipient. If the recipient user is not active, then the message is added to the list of unread messages for the recipient user on the server backend and they user is alerted that they have unread messages whenever they next log in.

Wire Protocol

We thought a lot about the different options here – passing strings, passing objects, creating our own bastardized version of JSON, etc. Jim suggested that because we used Java, it would be more difficult to write a protocol because Java uses types. However, he also told us that using just strings was admitting defeat, so we settled on an alternate method.

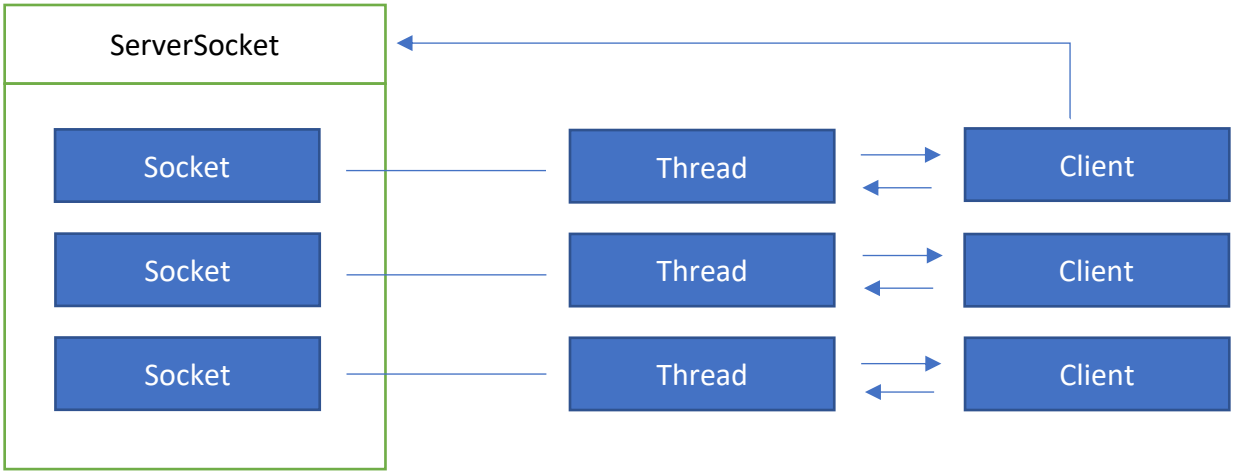
Our wire protocol is a simple byte-based protocol. The protocol allots one byte to store the data type (in our case always strings), 4 bytes to store an integer of the data length, and N bytes to store the data value. Strings are encoded based on the protocol in the Client class then sent over DataInput and DataOutput streams as bytes, then decoded based on the protocol by the ServerThread assigned to the client. The Protocol class includes the code for the encoding and decoding. All strings sent across the wire use the protocol, including every command sent by the user and every response sent from the server.

Design and Development Process and Decisions

We started out by creating a flow diagram for our anticipated user flow and class diagram for the classes we anticipated needing and their respective functionality. Those diagrams are included at the end of this document, but do not reflect the final state of our program, as we added additional classes for objects and threading.

For development, we started by creating a server that could connect to a single client, which was pretty straightforward. We then expanded on that to introduce threading into the server so it could support multiple clients. The ServerSocket allows for connections to the server, and

then each time a new client connects, a new socket is created for that client and a thread is created to connect to that socket. The thread handles most of the communication between the client and server, and minimizes the risk that a client cannot perform actions because the server is busy. So essentially the process looks something like this:



Getting this working was almost simpler than actually understanding how it all connects. We used the Oracle documentation and an example they have about writing a KnockKnockServer, which helped a lot. We also discovered the linux command nc for testing, which was a life saver. This command allowed us to initially focus on just the Server and ServerThread classes then create the Client class only after ensuring those two classes worked.

After getting multiple clients and threads working from a single server, we focused on adding in the login and account creation capabilities. Because this server is not always running (as one for a real chat app would be), we chose to save the usernames to a text file so that they were saved across different runs of the server. Obviously, this is a nonsecure method of storage, but for our purposes that is fine. Additionally, we chose to only require a username, and no password, to create or login to an account. Again, this is insecure and unrealistic, but adequate for our purposes.

One of the drawbacks of using a text file to store usernames is that we had to then update that file when new users were added or when accounts were deleted. Adding new users was simple – the file could be opened in append mode and a new line added in. However, deletion added some complexity. We considered two different options for doing this: 1) wait until the server is closed and rewrite the usernames file then with all of the usernames stored locally 2) rewrite the file using a temp file at the time of account deletion. Ultimately, we chose to go with option 2 – even though it is not the ideal solution (an ideal solution would be a database), we determined it was better to have the file update to the truth immediately instead of depending on a clean exit from the server. We also could have done this without creating a temp file by just rewriting the text file immediately but wanted to preserve the original in case of error.

Once the account functionality was finished, we moved on to the messaging. To keep track of online users for instant message delivery, we maintained a list of active ServerThread instances and stored the associated username with each thread. Then when a user sent a message, we could look through the active threads to see if the recipient was online and send the message immediately if so. Adding in this functionality was pretty straightforward. In addition, handling messages for recipients who don't exist was also simple.

Handling Unread Messages

The next major step was to determine how to deal with messages sent when the recipient was not online. There were a number of decisions to make in this space, including:

- Should the sender be alerted that the recipient is offline?
- How should the unsent messages be stored?
- Should a recipient be alerted that she has unread messages? If so, when and how?

Ultimately we made the following decisions for unread messages:

- Users could access unread messages via a text input similar to the other actions in the app. This would print out each of the unread messages on their own line for the user to read.
- Users would be alerted that they have unread messages both at login and at account deletion if they have any unread messages at those points, but the user would still have to use the command to read them.
- A sender of a message would not be alerted that the recipient is offline. Although this would be helpful in a chat situation so that the user does not expect an immediate response, because of the looping we use to handle inputs, it would create additional complexity in ensuring that we don't repeatedly tell the user that the recipient is offline.
- Unread messages would be stored in a text file, similar to the usernames, so that they would persist across multiple runs of the server. They would be loaded by the server at runtime and removed from the file upon sending. New unread messages would be added to the file.
- In the text file, each unread message is stored as a single line with each field (sender, recipient, and message) separated by the string "-|::|-" because this string is highly unlikely to show up in a message.

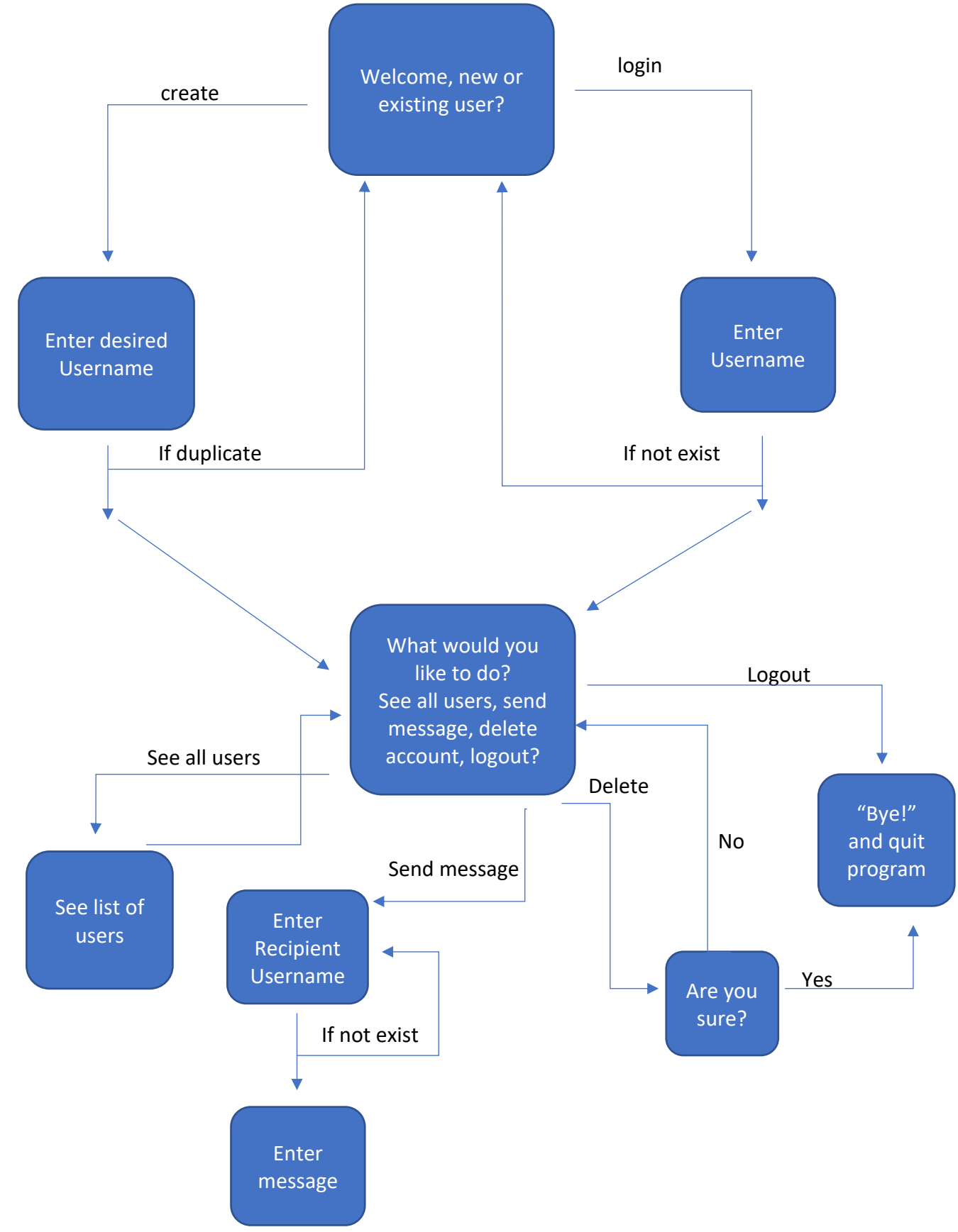
Client Code Threading

At this point, we had been testing using the nc command so that we could focus on just the Server and ServerThread functionality. We thought that adding in the Client code would be trivial, but found that it was actually trickier than we anticipated. Our initial inclination was to use loops in the Client class to handle the input from the user and output from the server, but could not figure out the correct loop logic to handle both streams at the same time. We

encountered a lot of interesting and undesirable scenarios when coding this way, which we determined to be timing issues because only one part of the loop could be executed at once.

We then decided that we had to use threading for the Client, so added a thread for the client to handle all of the messages coming in from the Server. We planned to add a second thread to handle the output stream to the server as well, but found that the program worked (or mostly worked minus a few tiny bugs that might have been fixed with the addition of a second thread, but ultimately we had to turn in the assignment).

User Experience Flow



Client

```
sendUsername()

// upload to server
// p2 add timestamp
sendMessageToServer()

getUsernameList()

// probably call this periodically
getMessage()

// something here to pass actions to
server

handleInputAction()
if (add)
```

Server (need a thread for each client), send success for everything

```
addUser()
        Call checkUsername() to make sure no
    duplicates
        Add new username to file
        Add username to list of online users

handleLogin()
        Call checkUsername() to make sure
    username is in db
        Add username to list of online users
        Call checkUnsentMessages()

checkUsername()
        Check if username is in db
        Return true/false

checkUserOnline()
        Check if username is logged in

// message from client to send to another user
receiveMessage(username, message)
        Call checkUser() to make sure username
    exists
        Store message somewhere - queue
        Call checkUserOnline()
        Call sendMessage() if online

sendMessageToClient()
        Call checkIsOnline() to see if user is online
        Send message
        Remove from queue

deleteUser()
        Call checkUnsentMessages()
        Remove username from logged in users
        Remove user from file

handleLogout() // also do if socket is disconnected
        Remove username from logged in users
        Close connection

sendAllUsernames()

checkUnsentMessages() // recipients
```