When implementing the socket-based chat app, one of the first design choices I had to make is what information I would store on each user and how would this information be stored. Initially, for each user, I stored their username, the list of messages that were sent to them (both messages “immediately” delivered and to be placed on a queue) and whether they were online or offline. I did the approach of storing messages to be immediately delivered because it didn’t seem right to store each user’s client connection as I initially planned my code that a user can join from different devices and thus, could have multiple client connections. However, dealing with the scenario of a single user having multiple client connections quickly became very complex and I pivoted to an approach where I only stored the list of messages that were to be delivered later (once the recipient was online) but I now also store the client connections so I can immediately deliver the messages. This proved to be simpler for me to implement and makes my code more understandable. With this approach, I think in the future, I could make it handle multiple client connections by storing a list of all the current active client connections and sending messages to each of those.

Another decision I made was what data structures should I use to store for each user, their username, messages, online/offline status and client connection. I ultimately decided on using a single large data structure for all the users instead of multiple data structures, each that stored some data because it would be tedious and error prone to have to update multiple data structures each time a change happened (e.g. the user status changed from online to offline, a user account has been deleted, etc.).

Once I decided to use a single large data structure, I ultimately decided on a dictionary with the username as the key which makes sense since usernames should be unique and then the value as a list that stored their messages, online/offline, and client connection. Since this is a lot of different information I am storing in a list as the value of a dictionary, I hardcoded the indices this information corresponds to in my list to make my code less error-prone.
extent to which I would adapt and repurpose the code for the wire protocol.

Another crucial design decision I made was that I initially had 1 thread on my client that handled both receiving and sending messages to/from the server. However, this caused issues when the client was receiving messages immediately sent from another user because the receiving client would only receive the message after they typed some kind of request, but then the output of that request would not occur until they typed the next request since I only had 1 thread on the client. Thus, this led to an issue where the user who received immediate messages would type a request but then the output of the request would occur after they typed their next request and the chain would continue with the output of a request being displayed after their next request causing an off-by-one error.

I now have two threads in my client, one that handles receiving messages from the server and another that handles sending messages to the server and this resolved my “off-by-one error” described in the previous paragraph. However, it did add a bit of complexity since I close both threads when I end a client connection. Moreover, when a message is immediately delivered, it does screw up the UI component on the terminal since a message that is immediately delivered will be immediately printed on the terminal regardless of what was currently being displayed on the terminal.