When implementing the gRPC part of this assignment, the first design choice I made was the extent to which I would adapt and repurpose the code for the wire protocol — as opposed to writing out the client and server code from scratch. I decided that, given the timeline and work allocation, I would mainly write up the code for the gRPC part independently of the wire protocol; however, I made sure to jointly architect the programs with Mohammed, my partner, to ensure our code were functionally consistent.

One of the most major design choices that I had to make was how to approach the design of the service in the **.proto** file, and by extension, the design of the server and client code that relied on this service. There were two viable options for how to approach this: (1) I could define the service with a *single* generic **rpc**, which has all the operations built into it via operation codes, or (2) I could define the service with multiple **rpc**s, with one **rpc** per operation (similar to what was demonstrated in lecture).

```
service ChatApp {
   rpc PerformService(ServerRequest) returns (ServerReply) {}
}
```
Example 1: A single generic **rpc** responsible for all operations.

```
service ChatApp {
   rpc CreateAccount(CreateAccountRequest) returns (CreateAccountResponse) {}
   rpc ListUsers(ListUsersRequest) returns (ListUsersReply) {}
   rpc SendMessage(SendMessageRequest) returns (SendMessageResponse) {}
   # … etc
}
```
Example 2: Multiple **rpc**s, each responsible for an operation.

I ultimately decided to go for option (1). I found that it was closer to the mental model of what a protocol was. The operation codes, as represented by numbers, would mirror what a client would have to enter into the terminal via a menu. Moreover, it more closely mirrored the manual wire protocol from the first part. This would make it much easier to adapt some of the code from the first part when it came to writing the server and client code.

Another interesting design decision I had to make was how to represent the **ServerReply** message. At first, I considered having only a **reply**. However, I later decided to add a **success** value in addition to **reply**. I realized it was important to return some sort of indication of whether a request was successful: ‘success’ in this case is defined as some positive result, and ‘failure’ is defined as some negative result such as an error or an absence. This is because such a boolean value could be easily used to decide what steps to take, whereas an entire reply message would be very cumbersome to process!

Another major design decision that I faced was, on the server side, what data structures to use to represent the accounts, their online/offline status, and their respective inboxes. I decided to use two dictionaries, **usernames** and **inbox**, with the former mapping a username to their

online/offline status and the latter mapping a username to their inbox of undelivered messages. This was rather convenient as **usernames.keys()** would represent a list of existing users. I thought that dictionaries are efficient and flexible data structures, and are also relatively intuitive to iterate and index.

A design decision I made on the client side was how to prevent clients from sending inappropriate requests. For example, a client might try to log in or create account even though they're already logged into an account, or they might try to send a message even though they're not logged in yet. The way I solved this was by having a boolean variable **loggedIn** that would determine

Another major design decision I had to face was how to concurrently listen for incoming messages and process outgoing requests. What I did was, on the client side, I initiated a separate thread for listening using the **start_new_thread** function. We then used another operation code that represented this listening, so that the server could deliver an incoming message to the client as soon as one is detected in the **inbox** dictionary. This allowed for immediate delivery of messages, if user is logged in, as well as the solicitation of the user's typed request.

An interesting observation I noted, as a caveat of this design choice, is the awkward interface that presented itself in the terminal. Usually, in a chat app interface, there are two separate, disjoint interfaces: one for displaying incoming messages, and another for inputting messages akin to a text field. In a terminal, both of these interfaces are conflated. The design choice I made was to explicitly solicit a request from users with a message "Enter your request:". However, what was slightly difficult was to ensure that this message was printed at the appropriate time. For example, sometimes an incoming message would 'block' this message and it would then be unclear that a request was being solicited. The solution I did was I added an **if** statement with the appropriate condition to determine when to print this message.