## Instructions:

Design exercise 1: Wire protocols

For this design exercise, you will be building a simple, client-server chat application. The application will allow users to send and receive text messages. There will be a centralized server that will mediate the passing of messages. The application should allow:

1: Creating an account. The user supplies a unique (login) name. If there is already an account with that name, the user is prompted for the password. If the name is not being used, the user is prompted to supply a password. The password should not be as plaintext.

2: Log in to an account. Using a login name and password, log into an account. An incorrect login or bad user name should display an error. A successful login should display the number of unread messages.

3: List accounts, or a subset of accounts that fit a text wildcard pattern. If there are more accounts than can comfortably be displayed, allow iterating through the accounts.

4. Send a message to a recipient. If the recipient is logged in, deliver immediately; if not the message should be stored until the recipient logs in and requests to see the message.

5: Read messages. If there are undelivered messages, display those messages. The user should be able to specify the number of messages they want delivered at any single time.

6. Delete a message or set of messages. Once deleted messages are gone.

7. Delete an account. You will need to specify the semantics of deleting an account that contains unread messages.

The client should offer a reasonable graphical interface. Connection information may be specified as either a command-line option or in a configuration file.

You will need to design the wire protocol— what information is sent over the wire. Communication should be done using sockets constructed between the client and server. It should be possible to have multiple clients connected at any time. Design your implementation using some care; there will be other assignments that will utilize this codebase.

You should build two implementations— one should use a custom wire protocol; you should strive to make this protocol as efficient as possible. The other should use JSON. You should then measure the size of the information passed between the client and the server, writing up a comparison in your engineering notebook, along with some remarks on what the difference makes to the efficiency and scalability of the service.

Implementations will be demonstrated to other members of the class on 2/12, where you will also undergo a code review of your code and give a code review of someone else. The code review should include evaluations of the test code coverage and documentation of the system. Code reviews, including a grade, will be turned in on Canvas, along with your engineering notebook and a link to your code repo.

## Brainstorm:

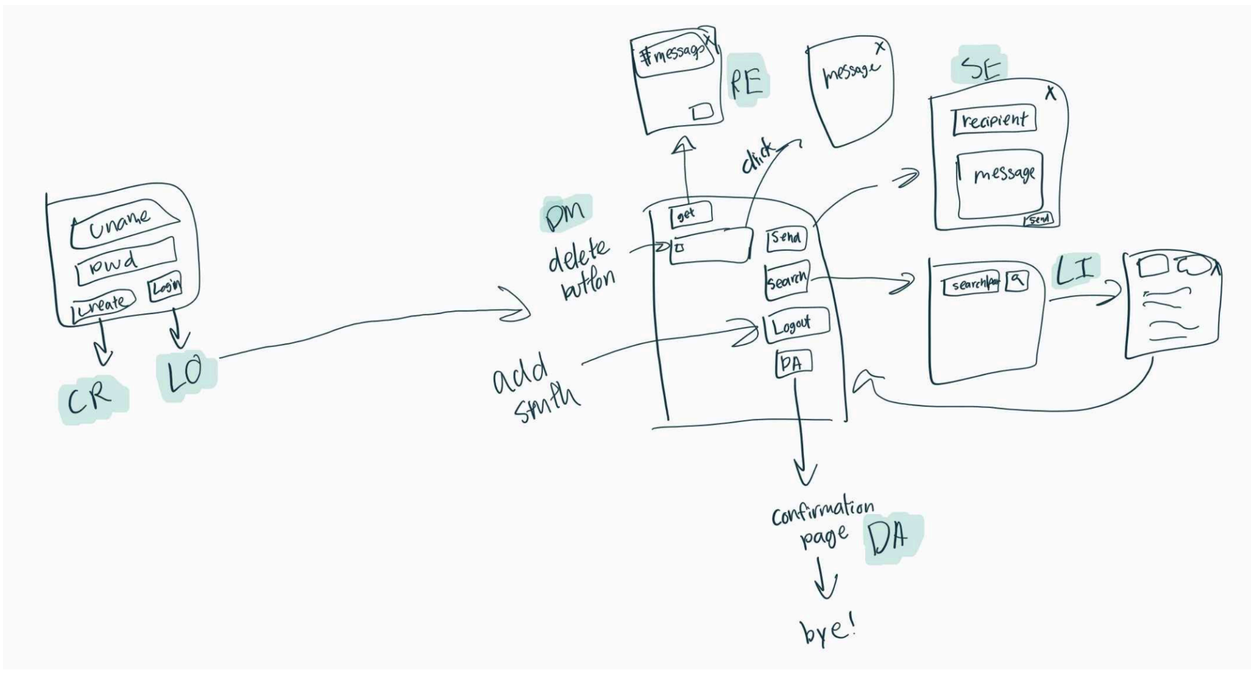
Wire protocol

- First few characters of every message are numbers that determine the number of bytes in the rest of the message (including the 2 character type determinator
- After the numbers, the first two characters of the "message" determine what action the message is trying to do
    - CR - Create account
        - Can't be empty/null - check on client side
        - NO SPACES allowed in username/password - check on client side
        - So format is
        - CR[username][space][password (hashed)]
        - Enforce unique usernames
        - Returns to client: CRT on success, ER[something] on failure
    - LI - Login
        - LI[username][space][password (hashed)]
        - Returns: LIT on success, ER[something] on failure
    - LO - log out
        - LO[username]
        - Returns LOT on success, "cannot" fail (unless the connection fails or something)
    - LA - list account NOTE: because we do filtering on client side we simply return a list of all accounts
        - LA
        - Returns to client: LAT[username][space][username][space]...
    - SE - send message
        - SE[from username][space][to username][space][date][space][message]
        - Returns to client: SET, "cannot fail"
        - If the to username is logged in, server will also send a message to that user's socket

                - This message has form SEL[number (messageID)] [space] [fromUser][space][timestamp][space][# of characters][message]
    - RE - read message
        - RE[number] (maybe empty, in which case we list all unread messages, OR invalid, in which case we return an empty string)
        - Returns: RET[# of messages read total][space][number (messageID)] [space] [fromUser]  [space] [time] [space] [# of chars][message]..[space] [number (messageID)] [space]..
        - May need to add other stuff like timestamp at the end or something using a similar format
        - Text entry for number of messages to read
    - DM - delete messages (that you have received)
        - DM[username][space][messageId]
        - Returns: DMT on successful deletion, ER[something] on error
    - DA - delete account
        - DA[username or uuid]
        - Note: UI should also kick this user out/automatically close
        - Returns: DAT, "cannot" fail unless the connection fails or something
- Return format of errors:
    - ER[error message]
- Change parsing method based on those first 2 characters
- Password needs to be hashed before sending
    - Hash password before sending the LO request

JSON:
- TBD, probably just sending parts of the accounts data structure that can be seen below

Features:
- Email-style, each message gets its own screen/box/whatever
- Messages from the same user show up as separate messages, because we don't feel like implementing email threads
- Deleting an account = all messages associated with that account are deleted too
    - Unread messages "fail to send"? Or they just disappear
- For the MVP0, we won't show the user the messages they sent out
    - Can add this later maybe
- For MVP0, we will just use the username is the user's name
    - Can give display name later
- For MVP0, when an account is deleted, their messages to other people will remain, but any messages sent to them will disappear



Data Structures:

- Dictionary of lists w/ username as key
- ~~Somehow put a lock on this data structure~~

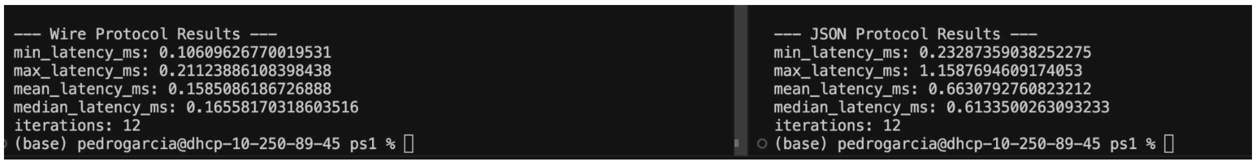
```
{
        username  : {
        loggedIn: True/False
        accountInfo :
                {
                        Username: User name,
                        Password : hashed password
                },
        messageHistory: [
                        {
                                Sender: “value”,
                                Timestamp : “value”,
                                Message : “value”,
                                messageID: “value”,
                                delivered: true/false
                        },
                        {
                                Sender: “value”,
                                Timestamp : “value”,
                                        Message : “value”,
                                        messageID: “value”,
                                        delivered: true/false
                              }
                              ]
          },
}
```

Return true

```
{
          Success :true or false,
          Error : empty/{}
}
```

Success true

Experiments:



```
--- Wire Protocol Results ---
min_latency_ms: 0.10609626770019531
max_latency_ms: 0.21123886108398438
mean_latency_ms: 0.1585086186726888
median_latency_ms: 0.16558170318603516
iterations: 12
(base) pedrogarcia@dhcp-10-250-89-45 ps1 %
```

```
--- JSON Protocol Results ---
min_latency_ms: 0.23287359038252275
max_latency_ms: 1.1587694609174053
mean_latency_ms: 0.6630792760823212
median_latency_ms: 0.6133500263093233
iterations: 12
(base) pedrogarcia@dhcp-10-250-89-45 ps1 %
```

Looking at our experiments, the wire protocol consistently showed **lower latency** than JSON. These experiments included various procedures defined such as creating an account, logging in, searching for a user, and messaging various users. ON 12 iterations of this setup The Wire Protocol exhibited a mean latency of approximately 0.157 ms and a median latency of around 0.156 ms, with individual measurements ranging from 0.106 ms to 0.211 ms. In contrast, the JSON Protocol showed a mean latency of about 0.467 ms and a median latency of roughly 0.453 ms, spanning 0.238 ms to 0.667 ms. Because JSON involves additional serialization and deserialization, it tends to be slightly slower. The wire protocol’s more compact format reduces processing time, resulting in faster average message round-trips.