262 Project 1 Dev Journal

Details on Testing:
We wrote unit tests for all of our utility functions shared across the socket and gRPC implementations (formatMessage, parseMessages, parseSearchResults, searchUsernames, isValidUsername, isValidQuery, isValidMessage) using the python `unittest` library testing a variety of failure and success conditions.

We currently test our server and client functions for both gRPC and sockets using manual end-to-end tests (ie. running the client and server code and testing inputs that represent a variety of failure cases which are all well-documented in our code and informed our code generation). We were further able to test the functionality of the socket and gRPC server connections and behavior itself. We did this by creating another thread which would run the server, and then in the main thread running the client and communicating with requests. We tried all operations across sockets and gRPC, which can be found in the `testSocketServer` and `testgRPCServer` functions.

Moreover, we also manually ran the server and client code on different laptops, testing for various edge cases. We list some below:

- A user tries to send a message, logout, or delete their account while not logged in.
- A user tries to register or login while logged in.
- A user tries to register with an already-existing username.
- A user tries to login with an already-logged in or unregistered username.
- A user deletes their account and then tries to log in again.
- A user sends a message to themselves (this is OK by our spec, we specified the code such that they will only see the message and no confirmation).
- A user sends a buffered message to someone else and deletes their account (the recipient should still show the message upon login, but should not be able to reply).
- A user logs out, logs back in, and tries to send messages.
- A user tries to register/login/search using a username/query over 50 characters/with non-alphanumeric characters or tries to send a message over 262 characters/with non-ASCII characters or newlines/separators “|”.
- Client-side checks for correctness of commands.
- Two users talk live to each other and one logs out (the other should have their messages become buffered)
- The client code ^Cs (the client should automatically send a logout, and the server should close the connection)
- The server code ^Cs (all clients should close their connections and exit)

There have been many discrepancies as to the nature and extent of unit tests. Prof. Waldo seems to expect a rigorous suite of fully implemented tests while the TFs have repeatedly said that manually testing code for correctness is sufficient (as long as it works). These posts are linked below. Ultimately, we opted to err on the side of caution and implement tests for the

server functionality; however, for a project of this scope, we felt that our manual testing of the client and server code (and hundreds of lines of accompanying documentation) are sufficient to prove correctness.

- https://edstem.org/us/courses/36544/discussion/2615012
- https://edstem.org/us/courses/36544/discussion/2620417
- https://edstem.org/us/courses/36544/discussion/2628212
- https://edstem.org/us/courses/36544/discussion/2629920

Spec + Wire Protocol:
Our spec was laid out in annotations in our code, as well as in the dev journal below (February 12 notes). Please see there for details.

Comparing gRPC and Sockets:
Aside from sharing utility functions and a common spec, we opted to implement the gRPC and socket code mostly separately. This gives us a bit more variety to compare the two techniques.

- **Code style and complexity**. We discuss this a bit in the code itself and further below in the journal, but at a high level, we found that compared to socket code, working with gRPC gave us higher levels of abstraction but also less granularity. With socket programming we had a higher degree of control over how to establish and maintain client-server communication. Specifically, we could manually define a wire protocol, giving us a lot of granularity over how much we actually needed to send and receive over sockets; in some cases, we only needed to send and read a 1-byte server status code to interpret the results of an operation. This, however, also came with more details to manually deal with, such as manually starting and ending connections, as well as, on the client side, our setup of running two threads to both send and receive continuously over a connection.

  gRPC provided a couple helpful abstractions that, in our opinion, actually made the code more intuitive to use in terms of implementation. While using auto-generated code was confusing at first, specifying how to transfer messages using protobufs gave us a straightforward and communicable way to define our protocol (and of course, protobufs also possess the cross-language compatibility benefits we didn't use). Writing the server code proceeded in a more API-like fashion, since each method was manually specified under the server class, and for clients we could handle communication through a call-response model. The ability of protobufs to specify that a call could receive a stream of results was conducive to handling instant message sending, and we did not need to start multiple threads for our server to handle multiple connections. Overall, gRPC lent itself to a more straightforward and intuitive style, but gave a lesser degree of control.

- **Performance.** gRPC ran significantly slower than sockets as determined by timing. We would think this is caused by its higher level of complexity in dealing with stubs and protobufs, as well as its use of HTTP compared to more low-level socket connections. However, both implementations were still adequately fast as to appear virtually instantaneous from the client end, as we would desire from a messaging app.

We specifically timed a few basic operations with both implementations in Python. First we tested how long it took to set up the server and client and establish a connection; then how long it took to run the suite of unit tests that tested all the operations, both successful and unsuccessful (register, login, search, send, logout, delete); then how long it took to process 100 successful register requests. (Note the times also include the overhead with printing to the terminal.)

- To initialize, sockets took 0.0022 seconds and gRPC took 0.0189 seconds.
- To run the tests of basic operations, sockets took 0.0024 seconds and gRPC took 0.0187 seconds.
- To run 100 registers, sockets took 0.0175 seconds and gRPC took 0.0712 seconds.

So the time used by gRPC was a few factors (not quite an order of magnitude) of the time taken by sockets. But both ran satisfactorily fast.

- **Message size.** As mentioned above, the granularity of sockets gave us exact control as to how much we needed to read over sockets. Combined with ASCII encoding, this gave rather lightweight sizes for the messages we actually needed to send. Particularly, server responses of status codes required only the reading of 1 byte. Individual usernames required transmitting no more than 50 bytes, and individual messages required transmitting no more than 262 bytes (not accounting for separators like newlines and "|"). Overall, this kept everything very lightweight.

  It appears that gRPC also keeps the message sizes they pass small. Message objects have a ByteSize() method which returns the number of bytes in the wire encoding, and the results given when these methods are called also give small sizes, on the order of 100 bytes. Message sizes were not quite as small as those passed through sockets, but we expect that they are still much more compact relative to, say, JSON. One final remark is that the Message objects in Python themselves always took up 80 bytes, even for messages with a status code, as determined by the sys.getsizeof() functions. We assume this is due to internal implementation-specific state in protobufs that is abstracted away from us.

Initial thoughts February 5

- We're going to try implementing everything in Python for the time being. This is for a few reasons:
  - We both have more background experience in Python compared to, for example, Java or C++. Debugging and iterating on code would be relatively faster because of this.
  - Python, on the whole, has a faster development cycle and simple syntax. Of course this comes at the cost of performance, control, and detail—and using, say, C++ would create a "better" program by many opinions—but for a project of relatively simple scale as a socket implementation this is OK by our standards.

- Also because of the limited scope of this design project (given its position as an introduction to the concepts in this course), we will not focus on performance or rigor.
    - Particularly, we won't think about locking issues.
- We will probably look at sending/receiving messages over the command line. But this could be extended to a Flask app hopefully without too much difficulty.
- Because Python is interpreted we can simply run the files using the python program, no need to compile. We will watch out for library dependencies.

Designing February 7

- Specification:
    - **Client**: any computer running the client code.
    - **User**: an entity who can send and receive messages. The difference between client and user is that we could imagine a user registering on one computer; stopping the program; and then going to another user and trying to "log in".
        - We assume a user will not be logged in on more than one client
    - **Server**: (one) computer running the server code.
        - The server will store buffered messages. As soon as a message is sent, it is deleted.
    - Functions:
        - **register**: a new user running the client code specifies a unique username to register on the server. The server checks that the username is unique. If so, the user is registered and is *logged in*. If not, the user is prompted to try again.
            - We'll assume that there will be no scenarios where someone concurrently sends a message to a user who is just registering, thereby creating a race condition.
            - The user should have a **maximum character length** (to make encoding messages easier).
        - (*new*) **login**: an existing user running the client code enters their username. The server checks that the username is registered and if so, is *logged in*. If not, the user is prompted to first register an account.
            - Undelivered messages will be delivered to the user immediately
        - **list**: anyone running client code, logged in or not, may prompt the server to list accounts, with an optional text wildcard. The server returns the list of accounts.
        - **send**: a logged in user specifies a receiver username and a message to send; the server checks for the validity of the receiver username. If the receiver is not a registered user, return an error message. If the user is online, deliver immediately. Otherwise, the message is stored server-side in a buffer.
            - Error message could say, `the recipient does not exist, or may have deleted their account`.
        - **Delete**: logged in user running client code asks to delete their account. The server deletes the user, removes their name from the registry and

logs them out. The server keeps the user's buffered messages (so clients who login will still get the messages, but can't message back)
        - **Logout**: close the connection.
        - There is no function to receive messages. Receiving messages is either done instantaneously through the send function, or received from the server's buffer upon login.
- Questions:
    - What happens to client messages after they are viewed? are they just deleted or are they saved on the client?
        - Client could keep a list of the messages they're receiving in each session which is continuously appended to. But we will choose not to save this to disk; if the client stops and restarts they will have a clean slate of messages (save queued messages to be delivered to them by the server).
    - How to store messages (transfer buffers) on the server?
        - Info needed: sender, recipient, message
        - For storing queued messages, format like a dictionary; key is recipient username, the value is a queue of (timestamp, sender username, message) tuples
        - See below for discussion of saving to (server's) disk
    - How to send messages over the wire?
        - Encode as plaintext and then use socket communication
        - **Impose a character limit so that message lengths are capped**
        - Let's wait until some office hours to gain clarity on how best to send socket messages.
    - What happens to queued messages if the server goes down?
        - Adding a dataset to store messages would probably be overkill
        - We will choose to store messages in the form of dictionaries server-side.
        - Pickling (or even just saving to a text file) to the server's local storage would help if the program is stopped and then restarted. *However, since this storage is local this assumes that the same computer will be chosen to act as the server every time.*
        - Saving to disk after every server communication (to minimize the chances of buffer messages being lost) would be really inefficient, though
    - How do we efficiently read through and deliver messages from file stored on disk when the server starts up again?
        - We could maybe have one file

Setup February 8
Links to keep in handy:
https://realpython.com/python-sockets/
http://jakevdp.github.io/blog/2014/05/05/introduction-to-the-python-buffer-protocol/
https://www.digitalocean.com/community/tutorials/python-socket-programming-server-client

- By Prof. Waldo's notes we'll assume both computers are on the same wifi network. (https://edstem.org/us/courses/36544/discussion/2522323)
- Tutorials usually just consider a setting where both the client and server code are run on the same computer, hence they use HOST 127.0.0.1. Here is a simple way to allow for inter-computer communication:
    - Let's just hardcode our port in both the client and server files as, say, 22067.
    - First run the server code, which will **print** the current host name (via socket.gethostname())
    - Then to run the client code, we'll accept as a command line argument (argv) the host name.
    - Or, we could just leave the host as an empty string, "". But that seems less wise.
- Charu will look at socket tutorials and set up some skeleton code. Eric will look at gRPC and similarly set up some skeleton code.
- Comments on gRPC:
    - Again we'll yield to the fact that this is a design exercise and shirk some best practices. We should use uuids for users, have secure channels, use a logger, etc. etc. … We'll not consider these desiderata for this assignment.
    - Generating from protos: `python -m grpc_tools.protoc -I ../protos --python_out=. --pyi_out=. --grpc_python_out=. ../protos/messageservice.proto`
- Also, it's almost certain that we'll be naming some stuff semantically suboptimally, but that's OK because this is an exercise.
- gRPC code will be handled through separate files.
- Socket and gRPC basics set up! We can communicate between our two computers.

Coding Feb 12

- **REMEMBER TO ANNOTATE CODE, WRITE A README, AND ADD ERROR CHECKING.**
- The TLDR is:
    - We are trying to use threads for everything right now: for input/communication in the client code, and handling multiple connections in the server code.
    - We went to office hours today and watched a quick demo; it seems to be the case that multithreading was generally encouraged by the TFs. So we are working off the base code provided in section today.
- Here's an updated spec for our wire protocol + functionality. It's easy just to pass everything as strings, but to at least preserve some shred of dignity, we're going to *preface* our string-encoded messages with an integer code which determines the operation used or response status. Specifically:
    - Each call only requires **one sendall()**, from client to server and vice versa.
    - The general form of data transfer will be: client sends the operation code followed by just strings separated by special characters, e.g. "|". Server does the same, plus a status code.
    - Reading will be determined by the lengths of messages, whereas sending details are conveniently covered by sendall(). We will assume that we will have few

messages, i.e. we won't be reading more than thousands of bytes in a single socket call. The code will take up one byte. For requests sent by clients, this is an integer corresponding to the type of operation desired. For responses sent by the server, this is an integer corresponding to an error code. Error codes across operations will **be disjoint, except for unknown error** (127). (No way we'll have more than 256 errors.)
    - We're using ASCII encoding for now. **For usernames, which we force to be alphanumeric, this is OK because all encodings are 1 byte per character** (so max length of username encoding is 50 bytes).
    - Errors can be returned by a server check, or client-side checks.

Operation code (for client requests) / **Status code** (for server responses) int8	**Content** String (ASCII byte encoding, string format dependent on operation) **E.g.** header (int16 length of results) + string-encoded results

- **Client state:** username (if user is logged in)
- **Server state:** list of registered usernames; map of users to their sockets (for communication); message buffer (map of usernames to lists of senders and messages in encoded form)
- **Register:** code 1
    - If the user is already logged in, this method is not allowed.
    - Client prompts the user to enter a new username.
        - Client side checks: maximum length of username is 50 (characters); username should only contain alphanumerical characters; username should not be blank.
        - If these checks don't pass, don't communicate with server and prompt user to try again.
        - Usernames must be unique and we're using them as uuids (boo).
    - **Wire:** Client sends 1 followed by string encoding of username; server can just read 51 bytes
    - **Wire:** Server sends status code; client can just read 1 byte
        - 1: Registration OK
        - 2: Username already exists
        - 127: Unknown error
    - Client notifies of successful registration with username and **prompts user to login** with their new username after registration. (The user is not automatically logged in.)
    - **Server changes state:** the new username is noted.
- **Login:** code 2
    - If the user is already logged in, this method is not allowed.
    - Client prompts the user to log in using their username.

        - Client side checks: maximum length of username is 50 (characters); username should only contain alphanumerical characters; username should not be blank. If these checks don't pass, don't communicate with server and prompt user to try again.
        - Users can **only be logged in on one device at a time.**
    - **Wire:** Client sends 2 followed by string encoding of username; server can just read 51 bytes
    - Server performs checks and retrieves unread messages
        - Check for error conditions below
        - Index into map to return unread messages; encode as strings (see below)
    - **Wire:** Server sends status code, followed by length of all unread messages and all unread messages; client dynamically reads (see below)
        - 8: Login OK, no unread messages
        - 9: Login OK, there are unread messages (delivered)
        - 10: User is not in system (hasn't registered)
        - 11: User is already logged in
        - 127: Unknown error
        - A 2-byte integer is stored before all unread messages indicating the total length of the encoded string (see below)
        - Unread messages are encoded as strings of messages, separated by newline characters; each message is of the form "sender username|message".
    - Client parses buffer messages and displays them along with total unread message count
        - First read 1 byte for status code
        - If there are unread messages (status code 9), read the next two bytes, and then read the next number of bytes instructed by that value to get string of messages
    - **Client changes state:** they keep track of the logged-in user, identified by the username.
    - **Server changes state:** the client socket is added to the username-socket map. The buffer may release unread messages.
- **Search:** code 3
    - If the user is not logged in, this method is not allowed.
    - Client prompts the user for query
        - Client side checks: maximum length of query is 50 (characters); query should contain only alphanumerical characters and *
        - Wildcard search interprets * as zero or more of ANY character. An empty string is interpreted as a query for all users.
    - **Wire:** Client sends 2 followed by string encoding of wildcard; server can just read 51 bytes
    - Server checks for usernames that match wildcard

        - To be lazy, we can just pass this into regex, adding ^ and $ to ensure an exact match of the query string while replacing * as .*, and search the server's list of registered usernames.
    - **Wire:** Server sends status code, followed by number of found usernames and all found usernames; client reads dynamically (see below)
        - 16: Search OK
        - 17: Search retrieved no results
        - 127: Unknown error
        - A 2-byte integer is stored before all unread messages indicating the total length of the encoded usernames (see below)
        - Usernames are encoded as strings separated by |, e.g. "user1|user2"
    - Client parses the server message, dynamically seeing how much to read in the manner of login (and unread messages), and displays number of found usernames/results
- **Send:** code 4
    - If the user is not logged in, this method is not allowed.
    - Client prompts the user to specify the recipient username, then the message
        - Client side recipient checks: recipient username should not exceed 50 characters and should only contain alphanumerical characters; username should not be blank; user can't message themselves. If these checks don't pass, don't communicate with server, don't prompt for message text, and prompt user to try again.
        - Client side message text checks: message should not exceed 262 characters.
    - **Wire:** Client sends 3 followed by encoding of message
        - Message is specifically encoded as "sender_username|receiver_username|message".
        - Server can read 4096 bytes
    - Server handles the message.
        - If the other user is logged in, immediately deliver message. Otherwise, **change state** by adding the message to the buffer.
    - **Wire for the client instantly receiving the message:**
        - 32: Incoming instant message
        - Data is sent as code 32, then sender username, then message. After parsing code 32, the client can read 4096 bytes and parse.
    - **Wire back to the sender:** Server sends status code, client can just read 1 byte
        - 24: OK, instantly sent
        - 25: OK, message buffered
        - 26: Other user does not exist, or may have deleted their account
        - 27: Send failed (other user's socket may have broken)
        - 127: Unknown error
- **Logout:** code 5
    - If the user is not logged in, this method is not allowed.
    - **Wire:** Client sends 4 followed by username; server can read 51 bytes

        - Client-side checks: username matches the actual logged-in username
    - Server identifies the username and updates its state
    - **Wire:** Server returns status code, client can just read 1 byte
        - Logout OK 40
        - Unknown error 127
    - **Client changes state:** No active username; system is logged out.
    - **Server changes state:** Username removed from username-socket map (so marked as logged out)
- **Delete:** code 6
    - If the user is not logged in, this method is not allowed.
    - Client prompts the user to retype their username to confirm
        - Client-side checks: username matches the actual logged-in username
    - **Wire:** Client sends 5 followed by username; server can read 51 bytes
    - Server identifies the username and updates its state
    - **Wire:** Server returns status code, client can just read one byte
        - Delete OK 48
        - Unknown error 127
    - **Client changes state:** No active username; system is logged out.
        - Logout logic followed
    - **Server changes state:** Username removed from username-socket map (so marked as logged out); username removed from list of usernames; all messages where user is the sender are purged from the buffer (**or**: sender username changed to “deleted user”)
- **On unexpected server disconnect:** Client, which is still listening, receives 0 bytes. Let’s have them logout?
- **On unexpected client disconnect:** Same thing on server side. The server should change state by updating its user-socket map?
    - For both, select seems like a way to mitigate this.
    - Try-catch should also do the trick…
- Client should vet for invalid commands; if there are checks that can be done client-side, they should. Reduce unnecessary communication. Client and server should also appropriately handle the other code not being run, analogous to disconnect scenarios above.

gRPC Feb 14

- OK, we basically implemented the socket spec using the above; the detail helped us hash things out
    - Threads on client and server side, following the TF tutorial
    - Finalized spec:
        - Server: Creates a new thread for each client connection, maintaining a map of logged-in users to socket connections. In each thread, over the lifetime of the connection we read from the connection, process the logic, and return a status code plus results if applicable.

        - Client: Creates two threads sharing a socket connection. One continuously reads from the socket to wait for server responses, parsing and displaying results on the terminal. The other continually reads input and sends it over the socket.
        - Note that the terminal could become messy if we're reading inputs and things get printed at the same time—whatever.
- Moving to gRPC allows us to stop worrying about the encoding specifics of the wire protocol by just transferring everything via protos. This allows us to perform some abstraction from the specifics defined above. Here are some changes:
    - No more operation codes for client messages; they can just call the corresponding RPC endpoints. Implementing the gRPC server is less low-level and more like writing a simple API.
    - Analogous to a map of users to sockets, to indicate users who are logged in we'll maintain a map of users to Queues (see below)
    - Again this is a design exercise so we don't concern ourselves with the nuances of gRPC, such as contexts, gRPC status codes, etc.
- Dealing with sending messages instantly:
    - Because we don't have a mapping of users to sockets and data is transferred through server calls, the way we send messages instantly is worked a bit differently: in addition to a buffer, for real-time messaging we maintain a map from users to Queues
    - When a user logs in, they will immediately "Subscribe" to a stream of messages, wherein the server reads from the Queue and yields incoming messages as they arrive.
    - Queues are closed (e.g. upon logouts) by passing in a special EOF string.
    - On the client-side, we don't want to block while waiting for a stream of messages from the server. So we can simply pass that functionality off to a child thread.
    - Since we want to be listening to messages immediately after login, we do this subscription step automatically in the login functionality.
- The high level spec for gRPC is:
    - All function calls are defined in the protobuf.
    - Server: simply implements the protobuf with the logic, which is analogous to that in the socket code. One difference is the addition of a Subscribe function, which delivers new messages as they get sent. Queues are used for each logged-in user to facilitate message delivery.
    - Client: runs most of the logic in a loop where user input is elicited, processed, sent as a gRPC call, and results are processed. To facilitate listening for messages live, a separate thread is created after logging in to listen from the server's Subscribe function.

Error Handling and Testing Feb 16

- Working on error handling is where the inferiority of Python really kicks into play. Figuring out runtime exceptions is annoying.

- To keep things simple, whenever we find a socket or gRPC error, or whenever an interrupt is detected (e.g. ^C), we will shut down on the client and attempt to log out first.
    - Under both implementations, we can implement this as a logout call when handling a KeyboardInterrupt.
- On the server side, if an error is detected we will close down the client connection.
- In child threads, we will call os._exit() to quit the entire program. This is a little ugly (because apparently it doesn't wait to properly terminate everything, etc.) but sufficient for our purposes.
- The majority of our error handling is just try-excepts:
    - We will try-except whenever we encounter a socket call. Importantly, for both the client and server, we can start by trying to read the 1-byte operation/status code. If the channel has closed, then 0 or None will be returned and we can handle that disconnect right then and there.
    - We can also just wrap the overall functions which handle socket and gRPC logic in try-excepts. Any errors encountered during the function runtime will be bubbled up to that except.
- We added a little bit more functionality to our spec:
    - Graceful error handling: if a client disconnects, they will first automatically log out (see above).
    - The client can choose to manually quit the program instead of just ^Cing by typing "quit", "bye", or "disconnect". This corresponds to a new operation code which can be handled entirely by the client code.

Finishing Up Feb 20

- We are using advantage of the grace period before demo day to make some belated final steps:
    - Annotating everything with comments that we think are pretty exhaustive;
    - Squashing bugs that pop up here and there.