ENGINEERING DISCUSSION:

Sockets:

Design:

Our server maintains a list of connections, a dictionary of clients, and a queue of messages. The list of connections is not the same as clients, since a single connection can login/logout as multiple different users. Connection list is handled when there is a connection interruption, which subsequently logs out the user if necessary. The dictionary of clients stores all usernames currently in use, with a value of 0 if the username is logged out, and a value of the connection address if logged in. In this way, we have a convenient way to check when sending messages if a user is logged in or not, and if so, we readily have the connection to send to; i.e. client sends to server sends to client.

For purposes of concurrency, we impose a lock object on the client dictionary and the message queue, eliminating race conditions when it comes to concurrent account creation, or sending a message to a user that's in the process of deleting their account. This ensures well defined behavior, and since python dictionaries are thread-safe, we can perform these operations with concurrent threads.

The message queue has a key per username, and a value of a list of undelivered messages. Whenever the user is logged out, incoming messages are stored in the queue, and delivered on demand once logged in.

The server side has a client logged in and logged out state that it keeps track of, and we impose that the server is in control of dictating when the client is in either state, with the wire protocol sending this information to the client in order for the client to adopt its state accordingly. Given this information that the client has, the client side handles issues of local improper message entries, such as unallowed requests in a given state or passing in over-limit strings. When it comes to nonlocal unallowed requests, such as creating an account that already exists or a concurrent login, the server must recognize this since it has global knowledge of all client states, and relay the error back to the client.

Type checking is not required in python for inputs since the terminal interprets any input as a string, so the comparisons that are conducted are string based.

All processing related to restrictions on message size or wire protocol tagging/metadata are done client side to ensure what the client sends to the server is in a finite set of options that are easily decipherable due to the wire protocol.

To reiterate what is below, our wire protocol takes advantage of the fact that there is a finite number of user interactions with a server, so we can add simple metadata to the beginning of

any string sent to the server to characterize its type. The only slightly nontrivial protocol is the send message feature, but metadata on recipient size makes the sent string decipherable.

gRPC:

Design:

For the grpc implementation, we rewrote the code in a way that is optimized for this specific protocol. The basic design is that whenever the client wishes to do anything, the client must send a request to the server with essential metadata, the server then executes the function server-side, and then returns the value back to the client. This allows users across languages to write code as if the server functions were local, and very easy to work with. We opted to have four basic functions that clients can call that are executed on the server. ServerSend, ChangeAccountState, ListAccounts, and ClientStream. The first three functions are straightforward in that the client passes some metadata and the server executes the function as intended (specific details are in the code). ClientStream is a bit more involved and is how we ensure live chat in a grpc system.

Similar to the socket implementation, we maintain a global client_dictionary (but also add a global message_dictionary) where the state of the server as a whole is maintained. Client dictionary maintains login state of all known users and message dictionary is a dictionary of undelivered messages for a given user. To ensure thread safety, we have a lock for every global item, and upon every read or write to that item, we acquire the specific lock until our reads or writes are complete for any given thread. We decided that this was the most straightforward way to avoid race conditions, and for the scope of this project, determined that the efficiency was acceptable.

We quickly ran into the challenge that grpc does not allow for continuous listening on the client side in the same way that sockets inherently does, and does not have a way to send messages to a specific client, since everything is dependent on a client calling the server. To solve this problem, we defined our functions as bidirectional streaming services, where neither the server nor the client has to wait to receive the complete message before processing. This meant that we could hack our system such that a client could, in a “listening thread” call a function in grpc that essentially never finishes executing until the client disconnects. This is the motivation behind ClientStream. Each client upon logging in calls ClientStream and passes their username, which then runs a while loop that continuously checks the message dictionary to see if any messages have been added to that user’s queue. If a message is found, ClientStream pops that message and then yields back to the client. To ensure thread safety, ClientStream pulses the locks for the respective objects, ensuring that other functions are able to update queues and account states in a reasonable amount of time. Whenever a user logs out or the connection is lost, ClientStream returns and closes that thread on the server.

The code for grpc is, in our opinion, harder to conceptually understand at first, and more complex to set up, but once established, is less complex than the socket implementation. Performance-wise, the grpc implementation is about the same as the socket implementation and may even be slightly faster. In terms of size, the buffers themselves are basically the same size and dependent on schema (single byte tags for sockets vs int tags for grpc), but rpc's require additional overhead that would not be present in sockets.

Size:
For an example of sizes, we send "yo" from mike to yush. In the wire protocol, this takes 8 bytes: 2 for tags, 2 for "yo", and 4 for "yush". gRPC measures this as 16 bytes, so we can see the overhead difference; gRPC for convenience with the proto files ends up sending more overall so the client can remote carry out server methods, while the wire protocol is more streamlined as the code is adapted to the specific protocol.

Testing:

Basic functionality and edge cases were tested thoroughly via unit tests and manual tinkering. Thread safety was established by unit test 10, where multiple threads create accounts at the same time and also send messages in a synchronous manner, and thorough revision of global variable read/writes. To run tests (or to write new ones) see the usage section below.

Future Directions:

Security and scalability are major concerns with the current grpc server (no passwords, minimal server side authentication checking, fine grained locking). A user interface would also resolve the problem of terminal clutter, but due to the focus of this project being backend implementation, we elected to leave a cleaner user experience as a future implementation. Our implementation, to ensure thread safety, is synchronous, since we have timeouts for our locks. Our system should also work without these timeouts due to the checking of connection status, but further testing on this should be done before declaring an asynchronous system.

DOCUMENTATION:

Installation:

Clone this directory, make sure the latest version of python is installed, your firewall is disabled, and that grpc is installed; details are here for grpc installation. Try sudo conda install grpcio for Mac M1's if other installation methods do not work.

Socket Implementation

Wire Protocol:

We utilize a wire protocol that leverages the fact that there are only a few “types” of acceptable user interaction with the server: Create Account, Login, Logout, Delete Account, Send, Open Undelivered Messages, and List Accounts. Each of these interaction types are tagged with a number in the set [0, 1, 2, 3, 4, 5, 6] in that order on the client side, which is then sent to the server, which processes requests based on this tagging.

For Create Account and Login requests, such requests are associated with a username (limited to 50 characters) that is inputted on the client side, and the wire protocol attaches the tag (in big endian bytes) to the username string, and the server extracts the username that way. For Logout, Delete Account, or Open Undelivered Messages requests, these can only be performed once a user is logged in (create account automatically logs users into that username), so only the tagging is sent.

For Send requests, the user inputs (client side) the recipient username and the text data (limited to 280 characters). The client processes this information by first appending the send tag, calculating the character length of the recipient username, appending that as a tag (in big endian bytes), and then concatenating the recipient username string and the text data string. In this way, the server can extract the recipient username, and the remainder of the string is the data to be sent.

Finally, for List Accounts requests, the user is prompted with a regex expression to match with the dictionary of users, so the protocol sends the tag with the regex string for server processing.

Now, we want chat behavior to be such that Create Account and Logins can only occur when the client is logged off, Logout, Delete Account, Send, and Open Undelivered Messages requests can only occur once logged in, and List Accounts can be requested in either state. To this end, the server tags messages it sends to the client with 0 (logged off), 1 (logged in), 2 (either state) indicating the state of the client. When the client receives a message from the server, it changes state according to this tag, with the 2 tag inducing no change in state.

Usage:

To run the socket server, open a terminal in the main directory and enter the command “python socket_server.py {IP Address} {Port Number}” where {IP Address} is replaced with the machine IP address and {Port Number} is replaced with an open port number for clients to connect to. After this, you can run “python socket_client.py {IP Address} {Port Number}” either on a separate machine or on the current machine where the IP and port number match that of the server. Make sure to connect to the same port number that the server is listening on.

At any point that you are logged in, you can receive messages from other users who know your username. This means that you may get messages in your terminal while you are typing a command. In this case, simply continue typing as you were and hit enter when you wish to send your input to the server. This may look a little strange but is completely normal behavior and will not interfere with functionality. In general, the terminal will print “success” if a desired action was executed by the server and will print an error message if something went wrong in execution.

Once the client is booted up, you must create an account or log in before sending or receiving any messages. To do this, simply enter into the terminal “Create Account” or “Login” and then enter the desired username when prompted (client ensures less than 50 characters for the limit). If that username is taken, the server will respond with an error status and the user will have to enter the desired prompt again. If that username is not found, an error status will be returned. Moreover, you cannot log in to a username if another client thread is logged into that username.

If you want to list accounts, you can do so at any time, whether logged in or not. Simply enter List Accounts into the terminal and when prompted, enter the string to match after “search users:” and then enter the maximum number of matches that you would like the server to return. The server will then return the matches in a string and output to the terminal. Our list accounts function follows regex rules for python, which you can read more about here.

To send a message, type Send into the terminal and then when prompted, enter the desired receiver’s username and then the message you wish to send. Messages are delivered in real time to users that are logged in and message queues to users that are not logged in are delivered as soon as the user logs in. **A success message when sending a text to another user does not necessarily mean that the user received the message.**

Indeed, messages sent to users that are logged out are stored in a dictionary queue. Upon logging in, a user may check their undelivered messages by requesting Open Undelivered Messages in the terminal.

You must be logged in to logout or delete an account, but at any point, if you would like to delete your account, type Delete Account into the terminal, and if you would like to logout, type Logout. Keyboard interrupts are accounted for and will automatically log out any user that was logged in at that terminal.

Running Test Cases:

Switch to the socket_tests folder in the repository, and start a test_server by python test_server.py {IP Address} {Port Number}. Once the test_server is running, you can run tests 002-010 by python test00x.py {IP Address} {Port Number} for x in [2, 10], which roughly test basic functionality and edge cases like double create accounts, concurrent logins, sending texts to nonexistent users, race conditions, etc. test011.py is a proof of load management by multiple concurrent threads.

gRPC:

Note no wire protocol is used here due to the convenient nature of gRPC protocols. All buffers and services are defined in wire.proto.

Usage:

To run the grpc server, open a terminal in the directory “grpc” and enter the command “python3 grpc_server.py {IP Address} {Port Number}” where {IP Address} is replaced with the machine IP address and {Port Number} is replaced with an open port number for clients to connect to. If successful, you should see a message “Starting server on {IP_addr}. Listening…”. After this, you can run “python3 grpc_client.py {IP Address} {Port Number}” either on a separate machine or on the current machine where the IP and port number match that of the server. Make sure to connect to the same port number that the server is listening on.

At any point that you are logged in, you can receive messages from other users who know your username. This means that you may get messages in your terminal while you are typing a command. In this case, simply continue typing as you were and hit enter when you wish to send your input to the server. This may look a little strange but is completely normal behavior and will not interfere with functionality. In general, the terminal will print “success” if a desired action was executed by the server and will print an error message if something went wrong in execution.

Once the client is booted up, you must create an account or log in before sending or receiving any messages. To do this, simply enter into the terminal “Create Account” or “Login” and then enter the desired username when prompted. If that username is taken, the server will respond with an error status and the user will have to enter the desired prompt again. If that username is not found, an error status will be returned.

If you want to list accounts, you can do so at any time, whether logged in or not. Simply enter List Accounts into the terminal and when prompted, enter the string to match after “search users:” and then enter the maximum number of matches that you would like the server to return. The server will then return the matches in a string and output to the terminal. Our list accounts function follows regex rules for python, which you can read more about here.

To send a message, type Send into the terminal and then when prompted, enter the desired receiver’s username and then the message you wish to send. Messages are delivered in real time to users that are logged in and message queues to users that are not logged in are delivered as soon as the user logs in. **A success message when sending a text to another user does not necessarily mean that the user received the message.** This only means that the message was received by the server and will be delivered once the intended receiver logs

in. **Note this is distinct from the Socket implementation; here, no prompt is required to automatically deliver queued messages.**

You must be logged in to logout or delete an account, but at any point, if you would like to delete your account, type Delete Account into the terminal, and if you would like to logout, type Logout. Keyboard interrupts are accounted for and will automatically log out any user that was logged in at that terminal.

Have fun!

Running Test Cases:

Start an instance of the server the same way that you would above, by running python3 grpc_server.py {IP Address} {Port Number} in the grpc subdirectory. Then, switch to the grpc_tests folder in the repository, you can run tests 001-009 by python test00x.py {IP Address} {Port Number} for x in [1, 9], which test basic functionality and edge cases like double create accounts, concurrent logins, sending texts to nonexistent users, etc. test010.py is a proof of load management by multiple concurrent threads and thread safety on simultaneous “create accounts.”