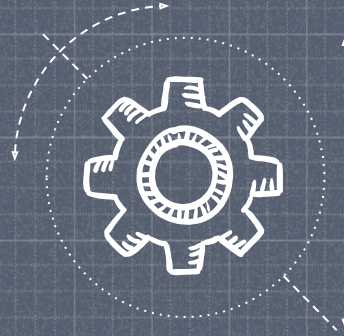




# CS262 Engineering Notebook

Pratyush Mallick | Anika Lakhani





# Programming Assignment 1



## Monday, February 6th: Specifications

### 1. Users can create, log into, and delete accounts

- a. What happens if a user tries to delete an account before their message has delivered?
- b. If a user isn't currently logged in, their message should be queued; deliver their message as soon as they log in
- c. Confused by "deliver undelivered messages to a particular user"
- d. Could use a SQL database for this, but based on the scope of the assignment, we decided that it makes more sense to use a dictionary to store account usernames and passwords (limitation: passwords are not secure this way because they are stored on an accessible txt file!)



## Monday, February 6th: Specifications (cont.)

### 2. Build a wire protocol

- a. One file that defines the functionality of a server
- b. One file that defines the functionality of a client
- c. Need to build a client, server that use these functionalities
- d. Support multiple clients; don't need to support multiple servers
- e. Write a README with detailed instructions on how to use these files as server and client wanting to chat

### 3. gRPC implementation

- a. Use gRPC to instead build the wire protocol, then compare over various factors: 1) complexity, 2) performance, 3) size of buffers, etc.



## Tuesday, February 7th: Preliminary Research

- [Message – python-can 4.0.0 documentation](#)
- [Documentation | gRPC](#)
- [Welcome to gRPC Python's documentation!](#)
- [Can someone explain what a wire-level protocol is? - Stack Overflow](#)
- [About: Wire protocol](#)
- [WebSocket: Simultaneous Bi-Directional Client-Server Communication | by Gabbie Piraino | Medium](#)
- [Socket in Computer Network - GeeksforGeeks](#)



Wednesday, February 8th: C++ Implementation Attempt

We first thought to implement with C++ because we were both fairly familiar with the language and wanted to use its range of packages.

After looking into a Python implementation, we realized that Python was far more efficient, elegant, and easy to code in (even though C++ compiles faster and is generally more comprehensive.)

The next slide shows the common pseudocode between our C++ and Python implementations:



## Wednesday, February 8th: Pseudocode Behind the Approach (CLIENT-SIDE)

1. Establish the host and port
2. Create the socket and connect
3. Create a message that stores the client's input
4. Until the message reads 'bye', signaling the end of the conversation, use the socket to send the message to the server. Be able to receive and print the server's response. Then, take in a new message.
5. Repeat step 4 as needed
6. Close the program



## Wednesday, February 8th: Pseudocode to Code Ideas (CLIENT-SIDE)

1. Establish the host and port
2. Create the socket and connect
3. Create a message that stores the client's input
4. while (message != "bye")
  - a. Use the socket to send the message to the server
  - b. Receive and print the server's response
  - c. Take in a new message
5. Close the program



## Wednesday, February 8th: Pseudocode Behind the Approach (SERVER-SIDE)

1. Establish the host and port
2. Create the socket, bind the host and port
3. Listen for and connect to new connections
4. Receive and decode the client's message; return an error if this is undesired data
5. Create a message that stores the server's input
6. Encode and send this message to the client
7. As long as you're receiving data, keep doing steps 4 through 6
8. Close the program



## Wednesday, February 8th: Pseudocode to Code Ideas (SERVER-SIDE)

1. Establish the host and port
2. Create the socket, bind the host and port
3. Listen for and connect to new connections
4. while True:
  - a. Receive and decode the client's message;  
error message + break if undesired data
  - b. Create a message that stores the server's  
input
  - c. Encode and send this message to the  
client
5. Close the program



Thursday, February 10th: Testing it Out

We tested the C++ code we generated from the pseudocode on one computer. We realized that, using this design, we created a requirement for a certain order in our communication: unless exactly one message is being sent alternately from client to server (A->B, B->A, repeat), our code does not work.

We would like to create an approach that does not restrict the messaging patterns of server and client. At this point in time, we also had not switched over to using Python.



Thursday, February 10th: Testing it Out (cont.)

We were also unsure of how to handle putting both client and server on one host. For now, we designed code in which the client and server are on the same computer.

We had yet to test this with two different computers and also felt very lost as to how to implement gRPC. For instance, it took a while to figure out how to auto-generate the Python code via the terminal after writing our proto file.



## Monday, February 13th: Buggy Python Implementation

We finally switched over to Python and saw immediate improvements. For instance, because of Python's functionality that can support `".connect,"` for example, our earlier steps in the pseudocode became much simpler:

- Creating a client socket:
  - `client_socket = socket.socket()`
- Connecting the client socket:
  - `client_socket.connect((host, port))`
- Taking input:
  - `message = input(" -> ")`



## Tuesday, February 14th: Remaining Todo List

1. Fix the Python implementation of the wire protocol
2. Create code that supports specification pertaining to accounts
3. Start on gRPC implementation
4. Upkeep design notebook
5. Compare the two approaches; record findings in design notebook



Wednesday, February 15th: Implementing Login User Flows

We created functionality for users to register and log in. We record the username and hashed password in a txt file to be referenced when a user is trying to log in again. We still need to figure out account deletion as well as link this login flow to our client.py.

We are a little stuck as to how to create an "inbox" for queued messages that we can show the user once they log in. We also do not know what to do if a user logs off before their message can send.



## Figuring out the Delete Functionality

We first tried to create an option from the main menu for a user to delete their account but then realized this would be a security issue if a user can delete an account before logging in. We played around and initially put the code into `client.py` because `client.py` would only be called from `login.py` once the user is logged in.

We then copied the contents of `client.py` into `login.py`, but we realized that having this content in one file could create dependency issues.

We also wanted users to be able to delete their account at any time as long as they are signed in, so we decided to add the Delete functionality by detecting when a user types `"/delete account"` in to the terminal. After a confirmation prompt, the account information will be deleted from the `.txt` file.

Deletion was tricky to figure out- we realized that there isn't a great way to simply remove a specific line from a `.txt` file in Python. So, we used a temp file to create the updated accounts information list, then renamed the accounts information list to the name of the temp file.

We also decided not to auto-delete an account after multiple failed login attempts and instead just direct the user to create a new account. We tried to put ourselves into the user's shoes to design the most helpful user flow.



## Friday, February 17th: Starting Proto

gRPC was initially a nightmare to figure out. Getting familiar with package installation and M1 Mac problem solving was a big barrier to entry for getting started on Part 2. After researching multiple client/server gRPC approaches, we saw many different and complicated ways to display a chat stream. It took a couple days for us to understand the technical concepts enough to realize that the chat proto line is actually very simple: to send a message, you take in a message stream and output a message stream (the messages can even be of the same struct!).

- `rpc Chatter (stream Reply) returns (stream Reply);`

We worked on first building an MVP, then adding extra functionality and complexity. The first version of our gRPC was a chat with no accounts and no login that kept crashing.

It was difficult to figure out what logic went in what file- did a given function belong in `client.py` or in the gRPC code, for instance? We tried adding functions and modifying the auto-generated code to no avail, eventually realizing that we didn't need to touch those two files and instead would need robust client and server Python files.

Because so many files share common names and functions, we had a lot of inconsistency based errors in our initial gRPC code where we didn't update an aspect of the code in each relevant file or were using inconsistent naming conventions across files. Taking a step back, simplifying, and first focusing on the MVP was really helpful- we started over around three times and became quicker at building up the basic functionality from scratch.

We wanted to explore other uses for gRPC beyond bidirectional communication, so on Saturday and Sunday, we worked on creating Unary, Server-Side, and Client-Side implementations.



Friday, February 17th: Trustable, Working Part 1

We also finally connected functionality between `client.py` and `login.py`, also testing and refining our Delete function, to get a nice, acceptable version of Part 1!

We added functions to display a user list as well as search for a user using the first letter of their username.

If we have time, we'd love to continue making our code more efficient, add more comments and README documentation, and possibly introduce fun colors into the CLI interface to improve the user experience.



## Friday, February 17th: Remaining Todo List

1. Allow for multiple clients in Part 1  
(want to switch to more of a node implementation where the server is merely the connector between two separate clients)
2. Figure out database implementation for 'inbox' (message queueing) functionality
3. Create and use unit tests for our code
4. Compare the two approaches; record findings in design notebook
5. Write README, commenting, general cleanup



## Brainstorming for Inbox Database Implementation

- Switch over to SQL so we can use joins and eloquently store a larger amount of data
- Create a 'message array' for each user that is printed out when a user logs in
- Create a txt file for each user in which messages to that user are tagged with the receiver username and appended to the text file → delete new messages from txt file when receiver username is logged in (they've been read) → print out txt file when a user logs in on their next session because only their unread messages will remain
- Use console logs to record messages and print them if necessary



Saturday, February 18th: We added a gRPC Easter Egg..  
**try to find it!**



## Saturday, February 18th: Global User Functionality

1. Consolidated `client.py` and `login.py` in Part 1 into just a `client.py` file; fixed any possible dependency issues that could come with this change
2. Redesigned Part 1 using a 'node' approach in which two independent clients can log in with username/password and message each other
  - a. This new design will help us because, previously, the server wasn't logged in as an account while chatting with the client.
  - b. Now that we have accounts for each chatter, we can assign a receiver to each message so that we know where to send queued messages
3. Created threads so conversations can happen independently



Saturday, February 18th: Nice-to-Haves for Part 1

1. Ability to still send messages if the other user you were chatting with logs off
2. A prettier CLI look and feel
3. A working message queue system
4. More messaging freedom than the one-for-one chat system we have (where each user trades off sending one message at a time, in an alternating order)
5. Functionality if a user wants to change their password or forgets it



## Sunday, February 19th: Auth System

1. Improved reimplementation of `server.py` (originally changed to handle multiple clients and support threads)
2. Implemented functionality for `/logout`; created a system to tell whether an existing user were authorized
3. Reimplemented `Delete()` to account for changes made to `server.py`
4. Implemented inboxes and wrote unittests
5. Wrote a robust and detailed `README.md` with instructions for users



## COMPARING PART1 and PART2 APPROACHES

### PART1

- Message response time is noticeably slower
- Coding experience was a lot more conventional-played to our existing experience
- Buffer size was larger
- More room to be creative with the client/server setup (for instance, we switched from a traditional model to a 'node' model)

### PART2

- Very big learning curve at first that honestly made this method less efficient
- However, once we understood how to implement functionality, our bonus three functions were very quick to make
- Buffer size was smaller because of tight packing of protocol buffers
- Code is less complex, but also less familiar to the average coder