

# Design Exercise 1

## Engineering Notebook

Adarsh Hiremath and Andrew Palacci

February 22, 2023

### Introduction

In our design exercise, we implemented an end-to-end client-server chat application, first with our own wire protocol and later with gRPC. We built our chat application using Python with a simple tutorial and eventually built functionality on top of our minimal chat room application to fulfill the assignment specification. As currently implemented, our chat application supports the following:

- Account creation, login, deletion, and deactivation.
- Sending messages between accounts, even when some accounts are offline.
- Listing or filtering all users who have created accounts.

The source code for our chat application can be found [here](#). Throughout this notebook, Andrew and I intend to describe various design decisions made throughout the development of this project.

### Programming Language

Initially, we considered 3 programming languages to implement our client-server chat application in: Python, Java, and C++. While Professor Waldo recommended that we complete the project in Java, Andrew and I decided to use another language since we couldn't find an elegant solution for handling different versions of the Java Virtual Machine (JVM) during peer grading.

Ultimately, we decided to implement our design exercise in Python because of the extensive documentation for the socket module. While C++ was the clear choice for low-level optimizations, Andrew and I wanted to pick a language that would be most beneficial for learning. After some exploration on the internet, it became clear that tutorials for socket programming in Python were far more accessible to us than tutorials for socket programming in C++.

The final benefit to using Python instead of C++ was code reproducibility. Most if not all students have Python natively on their machines. Any student can run our program with a simple command

allowing them to mirror the dependencies listed in a requirements.txt file, regardless of their machine. Conversely, maintaining code reproducibility across machines for a C++ implementation would require creating precompiled binaries for students, which we found to be far more tedious.

### Server vs. Client

Before implementing our client or server, we spent some time considering what input processing we wanted to put on the server side versus the client side. As will be explained later in this writeup, our wire protocol handles information differently depending on the operation code provided before a client types in the delimiter. Our options were to either process the operations on the client side, rejecting invalid inputs before even pinging the server, or handling the operations on the server side.

We decided to process information sent by the client on the server side for a variety of reasons:

- A client side approach to input validation requires all clients to download updates if any operation specifications change. This results in versioning issues.
- Validation done on the client side results in code redundancy since every client has duplicate code specifying what inputs are allowed.
- Clients having access to the type of operations they can use to ping the server results in security issues.

### Wire Protocol

Here, we will specify the functionalities supported by our wire protocol as well as provide further explanation about how our protocol was implemented. Our wire protocol utilizes a delimiter system to distinguish between different requests from the user. Our delimiter is '|', so any requests made to the server must include the appropriate command followed by the delimiter followed by any relevant parameters. For example, sending a message to another user would look something like:

's|<recipientUsername>|<message>.' Below, we've included a complete list of the arguments accepted by our wire protocol.

- Create an account. Usage: c|<username>
- Log into an account. Usage: l|<username>
- Send message.  
Usage: s|<recipientUsername>|<message>
- List all users. Usage: u
- Filter accounts. Usage: f|<filterRegex>
- Delete account. Usage: d|<confirmUsername>
- Print a list of all the commands. Usage: h

Commands like logging out, listing all users, and deleting an account become easier to implement with a delimiter system. Without a delimiter system, a client sending "d," for example, would be ambiguous for the server because there is no way to distinguish between sending the character "d" to another client or deleting an account. A delimiter system solves this because the client can still send the character "d" to a user as long as they specify their desired operation is a message send.

We use Python's We use Lib/struct.py to encode and decode messages efficiently using UTF-8. The two encoding and decoding schemes we considered were ASCII and UTF-8. At first, we considered implementing our encode / decode system using ASCII, since the size of any individual ASCII character is exactly one byte. This would make implementing character limits relatively easy to implement, since our argument to *connection.recv()* would be the maximum number of characters we want a user to be able to send using our chat application. However, since ASCII only supports A-Z, 0-9, and dashes, we decided to use a UTF-8 encoding system to allow users to use punctuation when necessary. We decided on a limit of 4096 bytes, which allows for approximately 1000 characters to be sent over our socket in the worst case scenario.

After receiving our bytes over the web socket and decoding, we do a simple *split('|')* operation to distinguish between the delimiter and arguments sent by the client. We decided to deal with white space using a simple *strip()* operation on every element in the split version of the client's command. This way, any spaces before or after the delimiter would be irrelevant and the client can still make commands as long as they follow the outline described above.

To keep our wire protocol simple and minimize code repetition, we decided to abstract away the implementations using helper functions for each operation. For example, this is how our code is structured for creating an account, logging in, and listing users. The code for the other operations follows a similar structure. Each operation code corresponds to a different helper function and each helper function

behaves differently depending on the specification of the operation.

```
if op_code == 'c':
    msg = create_account(msg_list,
                        connection)
elif op_code == 'l':
    msg = login(msg_list, connection)
elif op_code == 'u':
    msg = list_accounts()
```

Additionally, implementing our wire protocol using helper functions allowed us to easily make our server side send information to the user or print information in the console when applicable. In this case, our helper functions return information to print on the server and actually send information to the appropriate client within the function body. Here is a snippet from our login method with a helper function *verify\_dupes(connection)*:

```
def login(msg_list, connection):
    """Check that the user is not already
    logged in, log in to a particular
    user, and deliver unreceived messages
    if applicable."""

    if len(msg_list) != 2:
        msg = (
            colored(
                "\nInvalid arguments!
                Usage: l|<username>\n",
                "red"))
        return msg

    check_duplicate =
        verify_dupes(connection)

    if check_duplicate == True:
        msg = (colored("\nPlease log out
        first!\n",
        "red"))
        return msg
```

In our *login()* function, we have a helper function called *verify\_dupes()* which iterates through the connection objects of all the logged in users and compares them to the connection object of the user making a request. This identifies whether or not the user attempting to log in is already logged into another account. Similar helper functions are implemented for every operation in our wire protocol and greatly improve the efficiency of our code.

## Account Handling

To implement our account handling schema, we debated between using a database or simple Python dictionaries and lists. After consulting Professor Waldo, we decided to use a simple dictionary and list implementation since we assumed that there would be no Byzantine failures causing the server to unexpectedly shut down.

We have three main data structures that we use throughout our chat application.

- *pending\_messages*: a dictionary with username as key and pending messages as values.
- *accounts*: a list of account names.
- *conn\_refs*: a dictionary with usernames as keys and socket connection references as values.
- *logged\_in*: a list of users that are currently logged in.

We decided to use a single dictionary for pending messages to handle the scenario where a user is offline. Storing all messages sent and received in a dictionary would be a waste of space because our server only cares if the user has seen the message intended for them. When a user goes online, our program checks whether there are any pending messages to be sent to them and sends messages accordingly. We have a single list for account names in order to implement the listing users portion of the assignment. Our connection references dictionary stores all the users along with their corresponding connection references to manage communication between multiple clients. We also store the references to the socket connection in order for the server to send messages over the right connection. Finally, our logged in list stores a list of all the users that are currently logged to handle validation.

## Original Wire Protocol vs. gRPC

When creating the gRPC re-implementation of our original wire protocol, we tried to keep the code as similar as possible. As such, we created three server-side data structures that serve as analogues to each socket-based alternative: *messages*, *accounts*, and *live\_users*. *Messages* is particularly different from the previous version's *pending\_messages*, because although it's once again stored as a key-list pair, this time clients can listen to their keys on *messages* to find out immediately when they get a message. Thus we implemented messaging entirely via the *messages* dictionary and associated streams, instead of needing 2 different methods for online vs. offline messaging. Additionally, *live\_users* in the gRPC case serves a stronger purpose: keeping track of which users are currently logged in.

However, most of what surprised us were the stark differences between gRPC and the socket method, in terms of client vs. server side programming, general code complexity, and error tolerance. First off, the near-entirety of our socketed code was on the server side, for reasons previously explained. However, for gRPC, adding significant bulk to the client side was unavoidable, because gRPC has strict definitions for

the input and output types of each Remote Procedure Call. Thus we had to perform some error handling and packaging before serializing data to send to the server, which made us decide to split tasks relatively *evenly* between client and server side for gRPC. More specifically, we thought it made sense to validate the *format* of an input on the client side (how would it be transformed to fit a given message type otherwise?), and furthermore decide how to serialize the input. This still left the crucial task of validating an input's *content* to make sure it's useful, and we left this to the server — we figured that if the format of the data looks correct, the server should be responsible for checking it against other data it currently stores as well. Some other significant design decisions we made for the gRPC version are:

- Replacing UUIDs with unique usernames. We were previously storing a mapping between UUIDs and usernames, and not guaranteeing that
- Adding a username confirmation for deletion, so that people couldn't accidentally hit d and delete all of their information.
- Using regexes for filter strings, since the syntax is openly available and very well documented (as opposed to a less coherent, custom formula that we made ourselves). This allows everyone using the app to be on the same page.

As we began working on the gRPC version, we initially were surprised by the newness of gRPC and its very structured wire protocol. We also believed our code creation was more complex for gRPC, since we had to create an additional .proto file, compile it, and align strictly with the compiled code. This felt trickier and more nuanced than our original app, which relied almost entirely on server-side handling and employed a very simple string-based wire protocol. However, when we continued to work on both versions of the application, adding functionality and making the two semantically equivalent, we began realizing the power of gRPC. The gRPC version was far simpler to modify, due to the very structured nature of the data being transmitted, alongside powerful features like metadata in the "context" parameter of remote procedure calls. This also made gRPC easier to debug, since we knew the types of RPC inputs and outputs, and had a much clearer view of our system's constraints.

Furthermore, we noticed that the gRPC version was much more resilient than the socketed version of the chat app. Killing the server, or at least causing it to throw an error, was much more straightforward using a malicious prompt in the socketed app, whereas with gRPC no such effect would occur on the server side — if anything, there would be an

error on the client side that would force clients to reconnect, which is far more desired than any sort of error on the server side.

Thus, overall, we felt that gRPC was harder to get attuned to, but led to significant rewards in terms of robustness and maintainability. Shifting to performance, we also believe that.