

Engineering Notebook

Kamryn Ohly + Grace Li

Link to Live Notebook (with videos!):

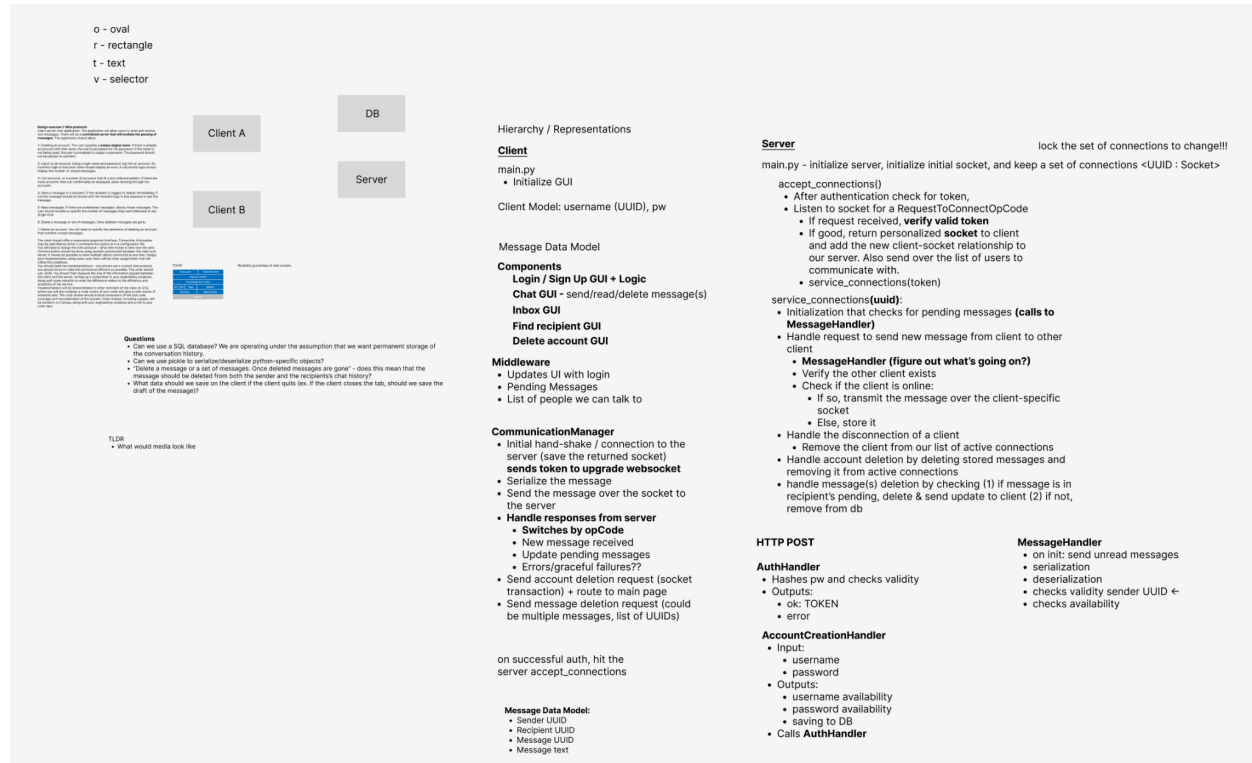
<https://docs.google.com/document/d/1c2UGPOdWTSNv5wOQQ20KYTk31JbHIEwcHpAHkqhWU58/edit?usp=sharing>

Demo Video: <https://youtu.be/jjelV9xjemw>

Introduction

Thank you for taking the time to test and try our messaging application. We thoroughly enjoyed our time creating this project and designing its architecture. In this journal, we have organized our thoughts throughout the assignment into a few main categories: architectural design, technical implementation, challenges & refactors, testing, and final thoughts. We will walk you through some of our initial approaches & ideas, our heartbreak when these approaches led to dead-ends, and how we overcame them. We will also outline our customized wire protocol and the rationale behind why we made the choices that we did. We'll start by talking about our architectural design and approach.

Architectural Design



The day that this assignment came out, we dove into organizing the architecture. On a [chaotic Figma document](#) (shown above), we brainstormed approaches that would satisfy the requirements. In particular, we wanted to ensure that before we started coding that we had a blueprint that would set us up for success. We divided into the Client and the Server, and we mapped out the flow of information, including whether the Client would hold onto data (such as pending messages) or if the Server would remain the “source of truth.”

In our first iteration, we started designing the **authentication** between the client and server first. We decided that upon launching the server that it would open a socket that any client could connect to after the client was authenticated. We originally pictured that the client would “register” and “login” to the server via HTTP calls, then the connection would be upgraded into a socket. Our initial perspective was that this approach would be more secure, as sockets allow for more unrestricted back-and-forth communication as opposed to HTTP requests and posts. However, after further conversation with each other & paying close attention during lecture, we learned that using HTTP would be a much heavier approach and would require unnecessary increases in complexity. All of which could introduce a greater security risk than the reward. For these reasons, we decided to authenticate the user via an individual socket between a client and server.

The client would connect to the server’s initial socket, which would accept the connection and pass back a socket that would allow the client to then complete a handshake to authenticate itself. In particular, we decided that we would create a SQLite database to keep track of authenticated users, and that a user could register with a username, password, and an email. We planned to use both hashing and a checksum to verify and authenticate user info.

Next, we focused on designing how to handle the communication between the server and client at an abstract level. We knew that the client would need to be able to initiate actions (such as sending a message, deleting one, or setting a limit on the number of messages that they could receive). We also knew that the server would need to be able to handle accepting requests and creating responses to these requests. We decided that we would create a system of “events” or operations that could be denoted by an operation code (“op_code” or “opcode” in our codebase).

We came up with ideas for what information and functionalities the Server would have, as opposed to the Client.

One major decision that we made earlier was that the primary source of truth would be held server-side. Specifically, when a client sends a message to an unavailable client, that message is stored server-side so that the client does not have the responsibility to stay on line to guarantee message delivery. Additionally, given the broad nature of the assignment, we made a few implementation decisions that we judged to stay true to the nature of the assignment:

When messages are deleted, we decided to only delete messages from the sender’s side, instead of from both sender and receiver’s conversation history. We made this decision because we reflected on

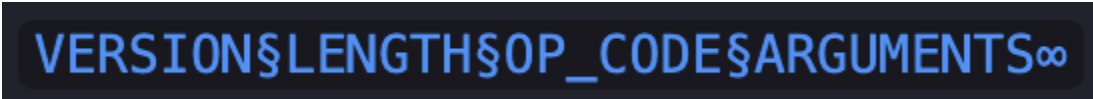
similar systems, such as email, and noticed that deleting an email only removes it for the person performing the “delete” action, and not for both parties.

Design of Custom Wire Protocol

One of the major parts of our project that changed and adapted most frequently was our custom wire protocol. We struggled to tell what would be the best, lightweight approach without resorting to too simple designs. We decided to use a delimiter and the below structure, but it definitely was not our first option. Originally, we did not utilize an ending delimiter - only learning our lesson after our messages presented in the UI began to look scrambled and nonsensical!

From the start, we knew that we needed to implement versioning & length to ensure reliability and scalability over time, especially if we made improvements that required backward compatibility one day. Thus, we lead our protocol with versioning information followed by the length of the rest of the message. For more technical information, see below:

Our protocol uses a structured string format where messages take the form:



`VERSION$LENGTH$OP_CODE$ARGUMENTS∞`

Here, the delimiter “\$” segregates different parts of each message, and a trailing “∞” signals the end of a message to ensure proper separation in the data stream. This design was chosen to keep the protocol lightweight and fast, enabling rapid parsing and minimal overhead in serialization. Additionally, the explicit length field helps in reading the correct number of bytes from the socket, further contributing to the robustness of our message transmission. An extension of this protocol would be to include a server and client side checksum computed by the contents of the message and the length to ensure the message was not tampered with during communication. While we were unable to complete this technically right away, this was part of our original brainstorming and we plan to implement it as our codebase continues to grow and improve.

To speak to more technical details about our protocol implementation, we decided to create a client-side `ServerRequest` class and a synonymous server-side `SerializationManager` class to handle the serializing and deserializing of our protocols. The client’s file would parse messages from the server into an easy-to-use format for the client, as well as format messages from the client back to the server. Then, the server’s class could parse the client’s messages as well as format responses. By placing all of this logic within these two files, it became incredibly convenient to make changes to our protocol without touching large parts of the codebase. Unfortunately, this was not the original approach that we took, and it was a major refactor to make this happen! We discuss more below, but this also led into our integration of JSON.

Approach to integrating JSON

To integrate JSON, we knew that there needed to be a convenient way to switch between protocols without requiring massive duplications of code. At first, we worked to create entirely separate handler functions and classes, except this felt wasteful and incredibly inefficient to debug. We knew that we needed to avoid hundreds of “if-else” statements, so we decided to use our above approach to serializing in one place to have our original serialization function parse both our customized protocol and JSON into the same data structure. When you run our code, you can decide whether to use JSON or our custom protocol, and the serializer will then behave accordingly. By parsing into a convenient common structure at the very beginning and serializing into the different structures at the end, you can find minimal code duplication. This also paves the way for additional protocols to be added with ease, as it is only necessary to change a small handful of functions.

```
import json

def serialize_to_str(version, op_code, arguments, isJSON=False):
    """
    Serializes the message into a string using either the custom delimiter-based protocol
    or JSON format.
    """
    if not isJSON:
        # Build the operation-specific part by concatenating the opcode and arguments.
        operation_specific = op_code if not arguments else op_code + "$" + "$".join(str(arg) for arg in arguments)
        length = len(operation_specific)
        # Return a formatted string with a trailing marker "∞"
        return f"{version}${length}${operation_specific}∞"
    else:
        # JSON protocol mode: build a dictionary object and serialize it
        message = {
            "version": version,
            "length": len(op_code) + len(arguments), # Simplified length calculation for JSON
            "opcode": op_code,
            "arguments": arguments
        }
        return json.dumps(message)
```

This snippet (from our server’s serialization manager) demonstrates how we build a message for both our custom protocol and our JSON mode. In custom mode, we join the opcode and any arguments using the delimiter \$, calculate the length of this operation-specific section, and then append a unique trailing marker (∞). In JSON mode, we simply pack the same information into a JSON object.

Functionality across computers

We ensured that our implementation successfully functioned across computers. To make this possible, we passed the server IP address as a command-line argument to the client IP and Port setup configuration. We are not hardcoding these elements as we are using command-line arguments.

Pending Messages

Users are able to save the number of pending messages they'd like to receive by altering their settings. This number saves to the database and will be remembered by the server upon user logout and login. We chose this implementation because this ensures the user has full control over the number of notifications they will receive.

Threading, Abstracting to Handlers, and Password Hashing

We employed multi-threading to simultaneous client connections on the server side, safely delegating each client interaction to a dedicated thread or handler so that the main application remains responsive. We abstract complex operations into separate handler functions or classes—this includes serializing and deserializing messages as well as processing client events. For security, we use SHA-256 hashing (via Python's hashlib) to store and verify passwords securely without ever saving them in plaintext.

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept_connection(server_sock):
    client_sock, addr = server_sock.accept() # Accept a new connection
    print(f"Accepted connection from {addr}")
    client_sock.setblocking(False)
    sel.register(client_sock, selectors.EVENT_READ, data=None)

def read_from_socket(sock):
    try:
        data = sock.recv(1024) # Read up to 1024 bytes from the client
        if data:
            # Process data: look for our message ending marker "\n" and handle the buffer accordingly
            messages = process_buffer(data)
            return messages
        else:
            sel.unregister(sock)
            sock.close()
            return None
    except Exception as e:
        print(f"Error reading from socket: {e}")

server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.bind(('0.0.0.0', 5001))
server_sock.listen()
server_sock.setblocking(False)
sel.register(server_sock, selectors.EVENT_READ, data=None)
```

Robust handling of multiple client connections is essential for our server. In this snippet, we set up a non-blocking socket server using Python's selectors module. This approach enables us to read from

the output buffer correctly and ensures that message boundaries—determined by our ending character—are honored.

Challenges & Refactors

One significant challenge was managing the output buffer combined with socket errors. Initially, our buffers would occasionally contain residual data from incomplete messages, leading to parsing errors. We tackled this issue by integrating selectors to help clear only the part of the buffer that corresponds to a complete message. This required the addition of a unique ending character, “∞”, which reliably indicates where one message ends.

We also ran into another buffer problem which was devastating and very time-consuming. We found that as we sent data over the socket that it would progressively hold onto “old” data, which would lead to long term issues. While we believed that we were clearing the `data.outb` in every circumstance, we neglected to recall that when Client A sends a message to Client B that we must clear both buffers. This error caught our attention for nearly 6 hours before being resolved, teaching us another critical lesson: **add error logging early and often**. At the time, we did not have enough logging to support efficient debugging, and after adding them, every subsequent error was far easier!

Another key refactor was the centralization of our serialization and deserialization logic. Rather than handling these concerns at multiple points in our code, we refactored everything into a single point of contact. This was a tedious process that required major alterations across our codebase, but it was a great undertaking as it ultimately not only reduced complexity but also allowed us to easily interchange our custom protocol with JSON when needed, thereby making the system more scalable and easier to debug.

Testing

In order to guarantee the reliability of our code, we wrote dedicated unit tests along the way with Pytest. We separated our test files based on functionality—one for the server's serialization manager and another for the client's server request handling. This separation allowed us to focus on testing the nuances of each module individually. We designed our tests to cover both our custom string-based protocol and our JSON-based protocol. For instance, we ensured that every aspect of our message format—from version and length to op-code and arguments—was verified by comparing expected outputs to actual results. We also made sure to account for edge cases like empty argument lists, special characters, and malformed data. To simulate network data, we used mocks so that our tests could run reliably without needing actual socket connections. Additionally, we tested our code between machines and guaranteed functionality. Each test includes a clear description of the scenario it addresses, mirroring the detailed style we used in our engineering notebook. This approach not only helped us catch issues early on but also ensured that our serialization and deserialization processes were robust and aligned with our design specifications.

Styling and Clarity

Throughout our codebase, we took extra care to include comprehensive docstrings for every function. These docstrings explain not only what each function does, but also provide context on how it fits within the overall system. For example, when detailing a serialization method, we describe the expected input parameters, the structure of the output message, and any nuances in the calculation of message lengths.

Conclusion

Overall, building this messaging platform not only gave us a hands-on look into what it takes to maintain and synchronize (slightly) distributed systems, but also introduced us to the fundamental concepts of logging, CAP-theorem tradeoff, a first-principles protocol approach, and real-world unit-testing. We found the experience not only deeply educational, but also surprisingly delightful. We look forward to building upon this foundation in problem sets to come!

P.S. AI Disclosure

We used ChatGPT to help us think of and write test cases, create the tkinter UI, draft the password hashing, and improve the structure of our documentation. We found it especially helpful to learn the tkinter protocol, as neither of us had worked with Tk GUI before.