Design Exercise 1: Wire Protocols

Ryan Jiang (ryanjiang@college), Evan Jiang (evanjiang@college)

February 14, 2025

1 Project Overview

For this design exercise, we built a simple, client-server chat application. The application allows users to send and receive text messages. There's a centralized server that will mediate the passing of messages. The application allows:

- Creating an account. The user supplies a unique (login) name. If there is already an account with that name, the user is prompted for the password. If the name is not being used, the user is prompted to supply a password. The password is not be passed as plaintext.
- Log in to an account. Using a login name and password, log into an account. An incorrect login or bad user name displays an error. A successful login displays the number of unread messages.
- List accounts, or a subset of accounts that fit a text wildcard pattern. If there are more accounts than can comfortably be displayed, we allow iterating through the accounts.
- Send a message to a recipient. If the recipient is logged in, deliver immediately; if not the message is stored until the recipient logs in and requests to see the message.
- Read messages. If there are undelivered messages, display those messages. The user can specify the number of messages they want delivered at any single time.
- Delete a message or set of messages. Once deleted messages are gone.
- Delete an account. We specify the semantics of deleting an account that contains unread messages.

The client offers a reasonable graphical interface. We also designed the wire protocol—what information is sent over the wire. Communication is done using sockets constructed between the client and server. It's possible to have multiple clients connected at any time. We built two implementations:

1. A custom wire protocol engineered based on class
2. One using JSON.

We measure the size of the information passed between the client and the server and wrote up a comparison in this engineering notebook, along with some remarks on what the difference makes to the efficiency and scalability of the service.

2 Engineering Log

2.1 [February 7] Initial Planning

The implementation of the chat system was divided into multiple phases to ensure a structured and modular development approach.

2.1.1 Phase 1: Connection Handling

The first phase involved establishing the fundamental client-server communication layer using sockets.

- **Multiple Connections**: The server was designed to handle multiple concurrent client connections using non-blocking sockets.
- **Read/Write Event Management**: Implemented a system to detect when data is received or when a client disconnects.
- **Graceful Disconnections**: Ensured proper cleanup of client sockets to prevent resource leaks.
- **Modular Architecture**: The code was structured to allow later phases (e.g., account management and messaging) to be integrated smoothly.

The key accomplishments of this phase include establishing a **basic socket communication** framework that enables the server to handle multiple client connections simultaneously. Proper mechanisms were implemented to ensure the **correct handling of client connections and disconnections**, preventing resource leaks and ensuring stability. Additionally, the server architecture was designed to be scalable, allowing for easy integration of future features such as authentication, messaging, and enhanced concurrency handling.

2.1.2 Phase 2: Account Management

The second phase focused on implementing user authentication and account management. Such features include

- **Account Creation**: Users could register with a unique username and password.
- **Secure Authentication**: Passwords were hashed before storage.
- **Login System**: Users had to enter correct credentials to access their accounts.
- **Account Deletion**: Implemented functionality to remove user accounts.

The key accomplishments of this phase include developing a robust **SQLite3 database schema** to securely store user credentials, ensuring data integrity and protection. Error handling mechanisms were implemented to manage **duplicate usernames and incorrect login attempts**, providing clear feedback to users and preventing unauthorized access. Additionally, comprehensive

testing was introduced to validate **account creation, login, and deletion** functionalities, ensuring system reliability and correctness. The high-level directory structure at this stage is set up below:

```
262DESIGN1/
   .vscode/
   client/
      client.py
   server/
      server.py
   tests/
   README.md
```

2.1.3 Phase 3: Messaging Functionality

The third phase focused on enabling users to send, receive, and delete messages.

- **Sending Messages**: Users could send messages to other users.
- **Message Queues**: Messages for offline users were stored/delivered when they logged in.
- **Message Deletion**: Users could delete specific messages from their inbox.

Several anticipated problems were identified early in the development process. One major concern was the lack of a mechanism to **mark messages as "read"**, which could lead to confusion for users attempting to track their conversations. Additionally, the system initially lacked a **persistent storage solution**, making it necessary to implement an SQLite3 database to store messages reliably and ensure they remained accessible even after users logged out. Lastly, there was the issue of **blocking** vs **polling**, where we still want live updates but we still want the ability to only show a certain amount of unread messages per instruction specifications.

2.1.4 Phase 4: Message Read Tracking and Concurrency Fixes

To ensure that messages could be marked as read, we added a **boolean flag** in the database to track whether a message was read.

To deal with concurrency issues on message display, we implemented the following **heuristic of blocking** when:

- A conversation is initially created.
- When a user is not currently viewing a conversation.
- When a conversation exists and has unread messages.

In these cases, we treat our app similar to inbox-style programs like Gmail or Outlook. We implemented a refresh conversations button that enables the user to immediately check for updates. Otherwise, we **poll** every 2 seconds when:

- Both users are online and loaded in their conversation.
- A user has finished viewing all unread messages in a conversation.

The last bullet point necessitated a view more button, and we often ran into errors before implementing this. Without blocking prior to viewing all unread messages, we observed that any number of messages we tried viewing would display all unread messages, rather than just the number we selected.

2.1.5 Phase 5: Account Listing and Filtering

The penultimate phase introduced an account search feature:

- Users could list all accounts or filter them using a wildcard search.
- Implemented pagination for large account lists.

We also updated the system to allow **dynamic port allocation** through command line inputs instead of hardcoding values. However, we still made localhost the default, in case people wanted to test run it on their computer without command line flags.

2.1.6 Phase 6: GUI Integration

The final phase introduced a graphical interface for client interactions, which involves (1) building with `Tkinter` for simplicity (2) allowing users to interact via buttons instead of command-line inputs. Remaining issues for concluding the project include

- JSON-based implementation for messaging.
- Improving server logging (currently minimal).
- Handling edge cases for account deletion.

Several technical considerations were taken into account to ensure the system's reliability, security, and scalability. **Storage** was implemented using `SQLite3`, providing a lightweight yet persistent solution for managing user data and messages. To handle concurrency, the system utilized **sockets with Python's `select` module**, allowing multiple clients to communicate with the server simultaneously without blocking operations. **Security** was a key focus, ensuring that no plaintext passwords were stored or transmitted, with proper authentication mechanisms in place to protect user data via `hashlib`. Additionally, robust **error handling** was implemented to manage potential issues such as login failures, message delivery failures, and unexpected disconnections, improving system stability and user experience.

2.2 [February 9] Development Progress

2.2.1 Design Choices and Trade-offs

Design Choice 1: Message Deletion on Account Removal. When a user deletes their account, all of their messages—both read and unread—are permanently removed from all recipients. This ensures that no lingering messages exist from non-existent accounts, prevents potential data inconsistencies where a conversation may reference a deleted account, and aligns with expected user behavior—when an account is deleted, its presence should be fully removed.

Pros	Cons
Maintains database integrity by avoiding orphaned messages tied to non-existent accounts.	Users who previously received messages from the deleted account may lose important records.
Provides privacy for users who want to fully erase their history.	If a user deletes their account accidentally, their messages cannot be recovered.

Design Choice 2: Any one of the two parties can delete any message in their conversation. Once a message is sent, both the sender and the receiver have the option to delete it within their chat history. If deleted, the message will disappear from both perspectives. We assume a terms of service and online etiquette for now.

Pros	Cons
Legitimate users who make mistakes (e.g., sending a message to the wrong recipient) can retract messages.	High amount of trust placed in the hands of potentially malicious actors.
Saves additional storage space since messages don't need flags for appearance for some conversations but not others.	Users who accidentally have deleted a message cannot restore it from another perspective.

2.2.2 Feature Implementation

Several key features were implemented to enhance the functionality and usability of the system. Unit tests were developed to verify the correctness of **account creation**, **login**, **and deletion**, ensuring reliability in user authentication. To improve flexibility, **command-line options** were introduced, allowing users to specify server and client configurations dynamically. Additionally, a **basic GUI was built using Tkinter**, providing a more user-friendly interface compared to command-line interactions. Finally, **message retrieval with read tracking** was implemented, enabling users to distinguish between read and unread messages for better message management.

2.2.3 Testing Cases

To ensure the robustness and reliability of the system, various test cases were designed covering account management, messaging functionality, and edge-case scenarios.

2.2.4 Challenges and Fixes

- **Blocking on message reception**: Prevented real-time messaging.
  - **Fix**: Switched to non-blocking socket communication.
- **Large messages crashing server**: Needed to handle message size limits.
  - **Fix**: Added manual safeguards for messages over 256 characters.
- **Message deletion inconsistencies**: Messages were not being deleted properly from recipient's inbox.
  - **Fix**: Allowed batch deletion of messages.

2.3 [February 10] Debugging and Refinements

2.3.1 Challenges and Fixes

- **Incorrect merging of login and chat functionalities**.
  - **Fix**: Separated login logic from the chat UI.
- **Unread messages not displaying correctly**.
  - **Fix**: Added an option for users to specify how many unread messages they want to see.
- **View more functionality displaying all messages instead of unread ones**.
  - **Fix**: Implemented a counter system for tracking unread messages.

Final Confirmation: Checked with Professor Waldo on message storage behavior and got approval for the UI layout and conversation requirements relating to unread messages.

2.4 [February 11] Final Development Tasks and Analysis

The final set of tasks focused on enhancing efficiency and ensuring the system was fully functional. A **JSON-based protocol** was successfully implemented, providing an alternative to the custom protocol for message exchange. Additionally, **server logging** was refined to accurately track client connections and system events, improving debugging and monitoring capabilities. Previously, the server constantly was printing all statements of polling, which made reading the log difficult. Irrelevant updates were silenced while useful ones were implemented.

One remaining issue was that deleting and recreating an account with the same name did not clear old message IDs, leading to potential inconsistencies in message storage. To address this, the database logic needed to be modified to ensure that all associated messages were permanently removed when an account was deleted. This fix would prevent orphaned messages from being

linked to newly created accounts with the same username, maintaining data integrity within the system.

To run the app, we define the following commands. For our Custom Protocol,

- `python server/server.py --host [IP Address] --port 54400`
- `python client/client.py --host [IP Address] --port 54400`

For our JSON Protocol,

- `python server/server_json.py --host [IP Address] --port 54400`
- `python client/client_json.py --host [IP Address] --port 54400`

We also ran rudimentary scaling and efficiency analysis that we could show for Demo Day. We plan to augment this after the presentation.

2.5 [February 13] Integrating Feedback and Turn-In Procedure

See Section 3 for details on our testing framework and our performance analysis. We also integrated feedback from our peers such as adding a version number as an additional check. Lastly, we looked over comments to ensure the code was clean for turn-in.

2.6 [February 24] New Assignment - GRPC

The whole idea for the gRPC assignment was to maintain all chat functionality but this time implement it with gRPC instead of sockets. The process framework was as follows:

- Create a .proto file for chat services (e.g. account management, messages, etc...) with RPC methods
- Figure out formatting for messages and errors
- Run the .proto file to create `chat_pb2.py` and `chat_pb2_grpc.py`
- Augment our testing suite to include gRPC unit tests, as well as revamping our efficiency and scaling tests.

Something we immediately noticed when working was that we got a concurrency error when messaging. Previously, when we were working with sockets, this was fine. However, SQLite in Python doesn't allow concurrent usage of the same connection/cursor object from multiple threads at the same time. Our current gRPC server would make multiple worker threads but we were using a single, global sqlite3.Connection, which would get recursive cursor calls, which weren't allowed.

To change this, we decided to use a new SQLite connection per request in each RPC method, so that each gRPC call uses its own database connection and cursor, avoiding concurrency collisions. This may prove a problem later when testing scaling, but we thought it would be alright for the small scale usage of our chat app.

2.7 [February 25] Working on the assignment

After implementing all the features, for testing we noticed it was very difficult to track the number of bytes sent and received, as gRPC runs on HTTP/2. So, we decided to approximate in our testing by approximating:

- The size of each protobuf message before it's sent (i.e. request.SerializeToString() length).
- The size of each protobuf message after it's received (i.e. response.SerializeToString() length).

This gets us the raw amount of protobuf bytes going across.

We also added unit testing through the same procedure for the gRPC. Lastly, we made sure all the ports were standardized.

For running the app, we now do:

- `python -m my_grpc.server_grpc --host [IP Address] --port [Port Number]`
- `python -m my_grpc.client_grpc`

However, the other 2 protocols still run the same.

For testing procedures, scaling tests actually opened too many file descriptors when we ran them locally, so we had to run the command `ulimit -n 4096` to avoid errors. Then:

- Efficiency Testing: `python -m tests.scaling_tests --protocol grpc`
- Scaling Testing: `python -m tests.scaling_tests --protocol grpc`
- Unit Testing: `python tests/run_tests_grpc.py`

We make sure that we're running the server on a different terminal beforehand. We've updated the results section below by augmenting the tables and adding new subsections in section 3. In addition, we answer the questions about RPC per the assignment in section 4.

3 Testing and Performance Analysis

3.1 Testing Framework

The testing framework verifies account management, messaging, retrieval, live message handling, and deletion. A subset of test accounts is reserved for consistency, ensuring cleanup after execution.

We wrote an automated test suite that includes 30 unit tests to check functional logic across these categories, and both our custom and JSON protocols are 30/30. The test suite was entirely self-contained. We have specific test user accounts with usernames that real clients would be unlikely to pick, and the cases handle the creation and deletion of accounts gracefully. In addition, the

test cases are all consecutive, meaning we were able to test potential errors that would come up throughout the process.

Below is the full process for all of our unit tests; every test is run sequentially in order, meaning the results of subsequent tests depend on the previous tests accurately working as well.

Account Management Tests

- Create TestUser1 account [expect success]
- Create TestUser1 [expect error: should say account already exists]
- Log into TestUser2 [expect error: should say user doesnt exist]
- Create TestUser2 [expect success]
- Log into TestUser2 with password “hello” [expect error: incorrect pass]
- Log into TestUser1 properly [expect success]
- Log into TestUser2 properly [expect success]
- Log out of TestUser2 [expect success]

Offline Messaging Tests

- TestUser1 sends TestUser2 “1” [expect 1 to be delivered]
- TestUser1 sends TestUser2 an empty string [expect nothing to be received]
- TestUser1 sends TestUser2 a SQL injection attack under 256 chars [expect the string specifically to be delivered not the SQL command to activate]
- TestUser1 deletes the previous message it just sent (the SQL injection attack) [expect successful deletion, this test was added later for test suite flow]
- TestUser1 sends TestUser2 a string of length 257 [expect error message too long]
- TestUser1 sends TestUser2 “2” [expect 2 to be delivered]
- TestUser1 sends TestUser2 “3” [expect 3 to be delivered]

Reading Checks

- Log into TestUser2 [expect success]
- TestUser2 checks the number of unread messages they have [expect total = 3]

- TestUser2 tries opening conversation with TestUser1 [expect the popup box asking how many messages they would like to read]
- TestUser2 selects to read 2 messages [expect to read 1 and then 2]
- TestUser2 checks the number of unread messages they have [expect total = 1]
- TestUser2 selects view more [expect to see popup box]
- TestUser2 asks to see 5 messages [expect error: too many messages]
- TestUser2 asks to see 'a' messages [expect error: value must be an integer]
- TestUser2 selects to read 1 message [expect to read 3]

Live Messaging Tests

- TestUser2 sends a message to TestUser1 saying "hello" [expect success]
- TestUser1 checks messages with TestUser2 [expect instant opening because both accounts are online]

Deletion Tests

- Delete TestUser2 account [expect success]
- Check TestUser1 messages [expect no available conversation with TestUser2]
- Create TestUser2 account [expect success]
- Check chat history between TestUser1 and TestUser2 [expect no available chat history]
- Delete TestUser2 account [expect success]
- Delete TestUser1 account [expect success]

In the future, potential improvements include implementing a short delay (e.g., 30 seconds) for message deletion to prevent accidental removals, enhancing error messages to provide clearer feedback, and considering rate limiting to prevent excessive message sending and potential spam.

3.2 Efficiency Analysis

3.2.1 Procedure

The efficiency evaluation follows a structured approach designed to measure two key performance aspects: **execution time** and **data overhead**. By performing the same sequence of operations using both the custom wire protocol and JSON-based protocol, we can directly compare their impact on system efficiency. There are three components:

(1) Standardized Workflow for Fair Comparison: The test script executes an identical sequence of actions for both protocols:

1. Create two user accounts.
2. Send 20 messages from one user to another.
3. Read all 20 messages at once.
4. Delete both accounts.

This ensures that any performance differences are due to the protocol itself and not external factors like system load.

(2) Measuring Execution Time: The test **starts a timer** at the beginning and stops it after all actions are completed. This helps quantify how much the protocol affects overall system responsiveness.

(3) Tracking Data Overhead: Throughout messaging, the test records the **total number of bytes sent and received**. For the custom wire protocol, the raw message string length is measured. For the JSON-based protocol, the size of the JSON-encoded messages is measured. This allows evaluation of network efficiency, since higher data usage can lead to increased bandwidth costs and latency.

The following table summarizes the efficiency test results:

3.2.2 Results

Protocol	Elapsed Time (s)	Total Bytes Sent	Total Bytes Received
Custom Protocol	25.396	2267 bytes	1070 bytes
JSON Protocol	25.310	4150 bytes	2339 bytes
GRPC Protocol	0.030	2220 bytes	1088 bytes

Table 1: Comparison of Custom, JSON, and GRPC Protocol Efficiency

3.2.3 Initial Custom and JSON interpretations

From Table 1 above, we observe that execution time is nearly identical (approximately 25.3 seconds), suggesting that serialization and parsing do not contribute significantly to processing delays. Most execution time is likely consumed by network communication and database operations rather than protocol-specific processing.

We also see that data overhead is substantially higher for JSON. The JSON protocol sent 4150 bytes, almost 83% more than the custom protocol's 2267 bytes. Similarly, the JSON protocol received 2339 bytes, which is 2.2 times larger than the custom protocol's 1070 bytes. This confirms that JSON introduces significant additional data overhead due to:

- Structural metadata (e.g., curly braces, field names like `"message"`).
- String escaping and encoding requirements.

Therefore, for large-scale deployments, the custom protocol may be preferable due to its lower bandwidth consumption and reduced network congestion, particularly in high-traffic environments where every byte matters. However, if flexibility and ease of debugging are more important considerations, JSON may still be a viable choice despite its increased data overhead. Ultimately, the decision depends on whether bandwidth efficiency or development convenience is the primary concern.

3.2.4 gRPC interpretations

Per the table, we can see that gRPC is orders of magnitude faster than the other two protocols, while still only sending and receiving about the same number of bytes as our custom protocol. There are a few reasons why this might be the case we can think about:

- Serialization - gRPC uses protocol buffers, which is more compact than JSON and our custom protocol doesn't serialize
- Network Efficiency - gRPC has built-in compression, connection pooling, and multiplexing

3.3 Scaling Analysis

3.3.1 Procedure

The scaling test evaluates how both the custom and JSON protocols handle concurrent messaging under increased load. The steps are as follows:

1. Start a timer.
2. Create k user accounts, where $k = [10, 20, 30, 40]$.
3. Each account sends a "hello" message to every other account in parallel.
4. Delete all accounts.
5. Stop the timer and collect metrics for maximum message latency, throughput (messages processed per second), CPU, memory, and network I/O usage, and total bytes sent and received.

3.3.2 Results

Metric	10 accounts			20 accounts		
	Custom	JSON	GRPC	Custom	JSON	GRPC
Elapsed Time (s)	21.579	21.572	0.111	43.650	44.051	0.214
Throughput (messages/sec)	2.085	2.086	404.069	4.353	4.313	886.848
Maximum Message Latency (s)	1.231	1.193	0.075	2.784	3.418	0.152
Total Bytes Sent	6231	10611	5826	22671	39031	21361
Total Bytes Received	1705	5980	1825	829	4048	6730

Table 2: Scaling Test Results (10 and 20 accounts): Custom vs. JSON vs. GRPC

Metric	30 accounts			40 accounts		
	Custom	JSON	GRPC	Custom	JSON	GRPC
Elapsed Time (s)	64.932	65.961	0.374	86.014	87.079	0.665
Throughput (messages/sec)	6.699	6.595	1162.195	9.068	8.957	1172.181
Maximum Message Latency (s)	3.781	5.060	0.298	4.702	5.944	0.563
Total Bytes Sent	30747	52855	46596	32459	55527	81531
Total Bytes Received	1374	5704	14745	1629	7544	25860

Table 3: Scaling Test Results (30 and 40 accounts): Custom vs. JSON vs. GRPC

3.3.3 Initial Custom and JSON interpretations

From the Tables above, we observe that

- **Execution Time and Throughput**: Both protocols completed in approximately the same times with nearly identical throughput, indicating that the choice of protocol does not significantly affect processing speed. Comparing the results, both also tended to scale linearly with the number of accounts, even though the number of messages being sent was quadratically increasing.
- **Latency**: The custom protocol tended to be around the same at lower levels, but as the number of accounts scaled up, the difference between the custom and the JSON widened noticeably.
- **Data Overhead**: The custom protocol consistently was much smaller than the JSON protocol, which is indicative of the higher network overhead due to structural metadata and encoding because of JSON.

In conclusion, the custom protocol is more network-efficient, minimizing bandwidth usage, which makes it a preferable choice for high-volume messaging applications where reducing data overhead is crucial. Although the difference in latency observed in this test is small, JSON's higher overhead could lead to noticeable delays in real-time applications that require rapid message exchanges. In terms of trade-offs, JSON provides better readability, debugging, and extensibility, making it easier to integrate new features. However, the custom protocol offers better efficiency and scalability, making it the more suitable option for large-scale deployments where performance and bandwidth conservation are priorities.

Both protocols perform similarly in execution time and throughput. However, the custom protocol is significantly more efficient in data transmission, reducing network congestion and improving scalability. JSON remains useful for flexibility but is less optimal for large-scale, bandwidth-sensitive applications.

3.3.4 gRPC interpretations

Once again, gRPC blows the other two protocols out of the water, with orders of magnitude lower elapsed time and orders of magnitude higher throughput. The maximum message latency is also much lower, but only by around one order of magnitude. Lastly, we can observe that the total number of bytes sent and received both tend to be way higher and scale way quicker with respect to the number of accounts. However, we expected this might occur, since we mentioned previously in our daily entries that our design choice to make it so every request got its own new SQLite connection in each RPC method and that each gRPC call gets its own database connection and cursor. We can kind of see this creeping up on us quickly.

Once again though, we can see that the origin of these results likely lies in a few areas:

- Serialization - gRPC uses protocol buffers, which makes encoding and decoding much faster and explains the time based statistics
- Multiplexing - gRPC uses HTTP/2 which has multiplexing, which allows many signals to go at once and stops blocking delays

4 gRPC Analysis

The assignment asks us several questions about gRPC. We will address these here. We utilized gRPC and define message structures with protocol buffers and communication over HTTP/2

4.1 Simplifications and Difficulties

Pros:

- gRPC automatically implements serialization for us, so we don't have to deal with manual parsing like JSON does
- The .proto file lets us automatically generate client and server stubs in many languages beyond just python if we wanted to, and all we had to do was run a command line argument

- We no longer had to deal with connection management since that is done automatically via persistent HTTP/2, so we don't care about manual socket reconnections/disconnections

Cons:

- gRPC messages are binary which makes it more annoying to read as a human compared to the easier JSON styles
- gRPC required more imports and we often ran into annoying project structure based errors as our gRPC files all lived together and were isolated from everything else

Overall, the learning curve was much harsher than anticipated, but we can see why gRPC is used, as its ability to generalize and make things run so much quicker is worth the tradeoff.

4.2 Size of the Data Passed

In theory, gRPC reduces message size due to binary serialization (protobuf), which is more compact than JSON. We can observe this was the case in our efficiency test. However, in the scaling tests, we actually saw that sometimes the amount of data increased as the number of accounts increased. There are a few potential reasons for this:

- The biggest thing to note is that for gRPC, we used an interceptor to count bytes, which counts the full structured protobuf message, while JSON and the custom protocol don't
- We also know that gRPC utilizes persistent connections so there will be more background traffic
- Lastly, it could be that our gRPC methods have field numbers, types, etc... or just have more metadata here

4.3 Structure of the Client

Before, we manually handled connections via sockets and we needed threading and asynchronous handling for live updates. We also used custom/JSON serialization. With gRPC, we use gRPC client stubs that are automatically generated. We handle messages through clients via gRPC streams, and we don't need manuel socket management or JSON parsing. We also have less polling overhead now.

4.4 Structure of the Server

Before, we utilized socket listening and connections, and the server handled JSON parsing. We also had threading for many clients. Here, gRPC automatically handles connections, message parsing is implicit with protobufs, and streaming support is native. The server now implements the gRPC-defined service interface instead of manually reading socket data. We see this with the decreased time statistics in the scaling tests and efficiency test.

4.5 Testing Impacts

Before, testing had to manually mock create and handle socket connections, parse through error outcomes, and had a custom client/server set up. Now, gRPC use generated stubs and have predefined message formats. It also offers more efficient code, but something we noticed is that the way we structured our project with grpc being a different folder than the old client and server and testing folders meant we had to create `__init__` files, so that our tests could import the proper files.