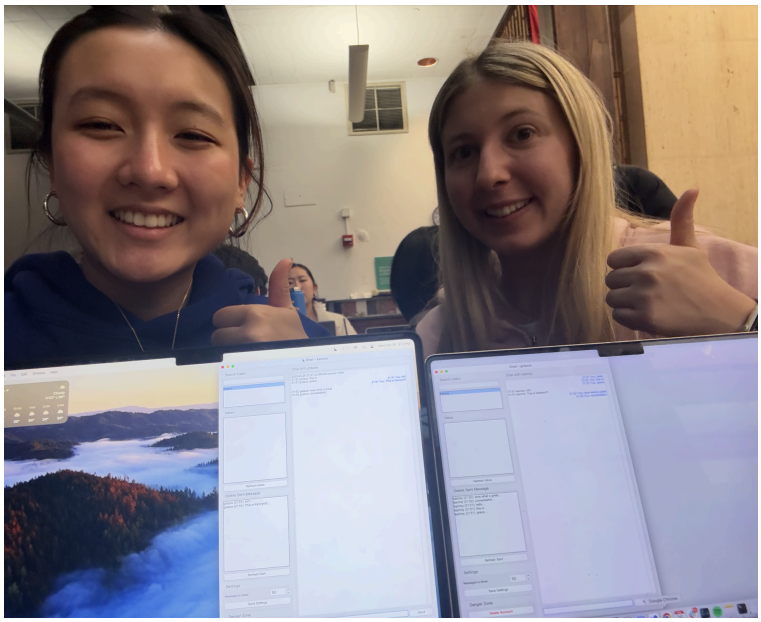
Engineering Notebook #2

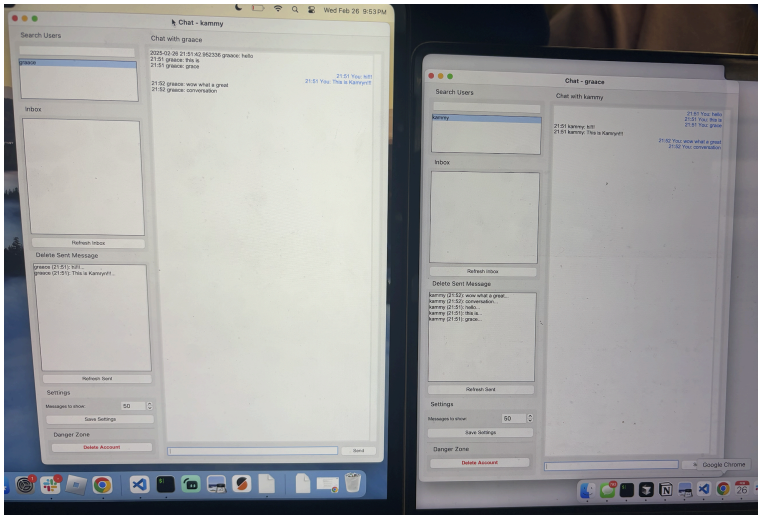
Kamryn Ohly + Grace Li

Link to Github: https://github.com/kamrynohly/distributed-systems-design-2

Link to Live Notebook (with videos!):
https://docs.google.com/document/d/1kC1dU5g-jA3qQlwvC7pxrMSXKUD7kRxc6Kewmv5-JYc/edit?usp=sharing

Demo Video:
https://drive.google.com/file/d/1Py2pnOCs6C8dqdbELynPOACELdsaOtkE/view?usp=sharing





Introduction

Thank you for taking the time to read through and test our second design project. For this project, we built upon our original custom wire protocol project (Github link here), to replace our custom protocol with a gRPC protocol. We used Python and its library Tkinter to accomplish this goal, replacing minimal lines of code between our first version and this one. In particular, most of our work went into the main.py of the client and the main.py of the server. We were able to substantially decrease the lines of code in our project, deleting entire files that were originally dedicated to our custom protocol serialization and deserialization. We are happy to have analyzed the pros and cons of utilizing a gRPC approach versus a custom approach in the rest of our notebook. We will start by discussing the necessary architectural changes that we chose to make, followed by our technical implementation, our testing, and finally, our analysis on the benefits and downsides of this approach. We hope that you enjoy it!

Architectural Design & Changes

We began this second design challenge offline on February 20th by diagramming the necessary changes that would need to be made to our original design challenge. We started by outlining the changes that we wish that we could have accomplished in the original design challenge, such as redesigning the interface to make deleting messages more clear, adding enums for operations instead of hardcoding as strings, and making the layers of our model, view, and controller more distinct. We worried originally that our initial layers were not distinct enough to support a seamless transition from custom protocols to gRPC, yet we were pleasantly surprised to find that our layers were separated enough to make it relatively easy!

After brainstorming original fixes, we then focused our attention on denoting the files that we believed that we would need to refactor. This included updating the server and client files to remove the socket connection and replacing it with gRPC service calls, eliminating the use of delimiters and JSON, and revising tests to accommodate the new communication protocol. Additionally, we defined the gRPC services and message structures needed to implement functionalities such as user registration, login, message handling, and account deletion. Defining these protocols, we knew, would be the most critical step of refactoring the architecture. We wanted to ensure that we supported all of the same features with enough information, yet also enough flexibility and scalability if we wanted to change anything. To accomplish this, we aimed to make each service as modular as possible, which included breaking up our initial "setup" functionality from our first design challenge into its individual parts. Rather than retrieving all other usernames, user settings, and pending messages in one singular service, we created three services that could be called sequentially by a client. This design choice ended up paying off, as it improved the way that we could write our test cases later in the project, and it made debugging far more convenient than only testing the larger combination of functionalities.

With our proto services created on paper and ideas in-mind for how to refactor our existing code, we dove into understanding how gRPC works and how we could set up our services to be managed by the server and convenient for the client. By the second time we worked on the project, we had four main questions to answer prior to diving into the technical implementation:

1. How will we set up these protocols?
1. How can we publish messages & subscribe to them?

2. Do we need to use streaming?
3. Should it be a bidirectional stream?

In our original custom protocol, it was easy for us to send arrays or dictionaries of information, as we could fine-tune our delimiters to inform us about when an array of information concluded. However, one of the first concerns that we noted with gRPC is that we can not pass arrays of unknown length without some workarounds. Additionally, we wanted to ensure that we still had socket-like functionalities of instantly receiving updates as they arrived. Thus, we decided to use a unidirectional stream of information running in a background daemon. While we could have sent information in batches or chunks of predetermined size, we believed that using streams in gRPC was a better solution as we wanted to simulate a user "subscribing" for updates on new messages.

With our choice to use gRPC streams, we had to address the fact that streaming requires more delicate care on the client-side and server-side than sending messages in batches. In particular, it requires that we are always listening and looping over our stream iterator in the background, while a user is interacting with the UI on the main thread. To ensure that we do not block the main thread nor cause data races between listening for updates and using the UI, we decided to use queues to handle the sending of messages and monitoring them. The "MonitorMessages" service is the main source of our listening for updates, which the client monitors on the "messageObservation" thread. We also decided to only use a unidirectional stream to avoid opening unnecessary streams between clients. For example, if Client A is sending a message to Client B, the action of sending a message is not a stream, yet Client B has subscribed to a stream for incoming messages. If we opened a bidirectional stream, it would need to have Client A streaming to the server, which is streaming to Client B. However, if you consider how heavy keeping open two streams (an incoming and outgoing) per client is for all clients who are active, it is quite expensive. It is especially expensive and unnecessary if a client does not send messages, yet only wants to receive them. For these reasons, we separated the sending of a message into a simple service, while the monitoring of messages opens a stream.

After creating this architectural design for our new approach, we tried to understand at a high-level how we could remove all references to our custom protocol. The good news is that we abstracted the layers of serialization and deserialization in a middle layer between our computation and our interface, meaning that we were able to easily plug in the gRPC protocols by calling our pre-existing functionalities. However, we knew that some of our UI code in the Client side needed to be altered to adjust to streams of information.

With visualizing most of these architectural changes up front (and of course running into many refactors as we implemented the code), we can now discuss our technical implementation.

Technical Implementation

The bulk of the technical implementation was done across the proto, server main.py, and client main.py. We outline our major technical changes in each file below.

Service.Proto

In this file, we outlined and implemented the grpc server, as shown in the image below. We made a few key abstraction choices, such as creating a message for messages (Message), as well as an enum for each action response's status.

```
// The service definition.
service MessageServer {
    rpc Register (RegisterRequest) returns (RegisterResponse);
    rpc Login (LoginRequest) returns (LoginResponse);
    rpc GetUsers (GetUsersRequest) returns (stream GetUsersResponse);
    rpc SendMessage (Message) returns (MessageResponse);
    // Stream because it is an array of messages
    rpc GetPendingMessage (PendingMessageRequest) returns (stream PendingMessageResponse);
    // Stream because we are subscribing for updates
    rpc MonitorMessages (MonitorMessagesRequest) returns (stream Message);
    rpc DeleteAccount (DeleteAccountRequest) returns (DeleteAccountResponse);
    rpc SaveSettings (SaveSettingsRequest) returns (SaveSettingsResponse);
    rpc GetSettings (GetSettingsRequest) returns (GetSettingsResponse);
}
```

Server-side main

This file primarily implemented the functionality outlined in the proto file for all services in the above definition. Each service required the correct service_pb2 input type and service_pb2 output type. Streams were handled accordingly.

A notably complex implementation here was MonitorMessages, as it needed to handle subscribing the client's RPC requests to updates to new messages, as well as remove the client from the list of active clients when they are no longer reachable. It outputs as yield as it is a stream return. The docstring is shown below.

```
def MonitorMessages(self, request : service_pb2.MonitorMessagesRequest, context) -> service_pb2.Message:
    """
    Handles a client's RPC request to subscribe to updates about new messages.
    This service also handles adding and removing a client from the clients who are active and reachable.
    Clients will create a stream with the server through this monitor service, which will be stored in
    self.active_clients.

    Parameters:
        request (MonitorMessagesRequest): Contains the client's details.
            - username (str): The username of the sender.
        context (RPCContext): The RPC call context, containing information about the client.

    Yields (stream):
        Message: The message that is to be delivered from a different client to the client who called this service.
    """
    try:
        logger.info(f"Handling client {request.username}'s request to monitor for messages.")
```

Client-side main

Our client implementation primarily married the business logic of UI flow with the corresponding functional grpc calls. Adapting our implementation from the previous assignment primarily involved re-implementing the callback functions with our new grpc architecture. We made a best effort to abstract common units of logic, such as the setup instructions necessary for both registration and login, to their own function components. An example is visible below.

```python
def _handle_setup(self, username):
    '''
    After successful registration or login, handle:
    (1) Fetch and return list of online users
    (2) Fetch and return user's settings
    '''
    try:
        logger.info(f"Setting up users and settings for {username}")
        user_responses = self.stub.GetUsers(service_pb2.GetUsersRequest(username=username))
        all_users = [user.username for user in user_responses]

        settings_response = self.stub.GetSettings(service_pb2.GetSettingsRequest(username=username))
        settings = settings_response.setting

        return settings, all_users
    except Exception as e:
        logger.error(f"Failed in setup with error: {e}")
        sys.exit(1)
```

We also took this time to improve some of our UI flow logic, such as routing the user directly to the logged-in state if they register an account. A snapshot of this is shown below.

```python
def _handle_register(self, username, password, email):
    """
    Sends a register request to the server and handles the server's response.
    If the user successfully registers, this also calls upon our set-up functionalities.

    Parameters:
        username (str): the user's username
        password (str): the user's password
        email (str): the user's email
    Returns:
        If successful, shows the chat, otherwise presents a failure message.
    """

    response = self.stub.Register(service_pb2.RegisterRequest(username=username, password=password, email=email))
    logger.info(f"Client {username} sent register request to server.")
    if response.status == service_pb2.RegisterResponse.RegisterStatus.SUCCESS:
        logger.info(f"Client {username} registered successfully.")
        settings, all_users = self._handle_setup(username)
        self.show_chat_ui(username, settings, all_users, {})
    else:
        logger.warning(f"Register failed for {username} with message {response.message}.")
        messagebox.showerror("Register Failed", response.message)
```

Testing

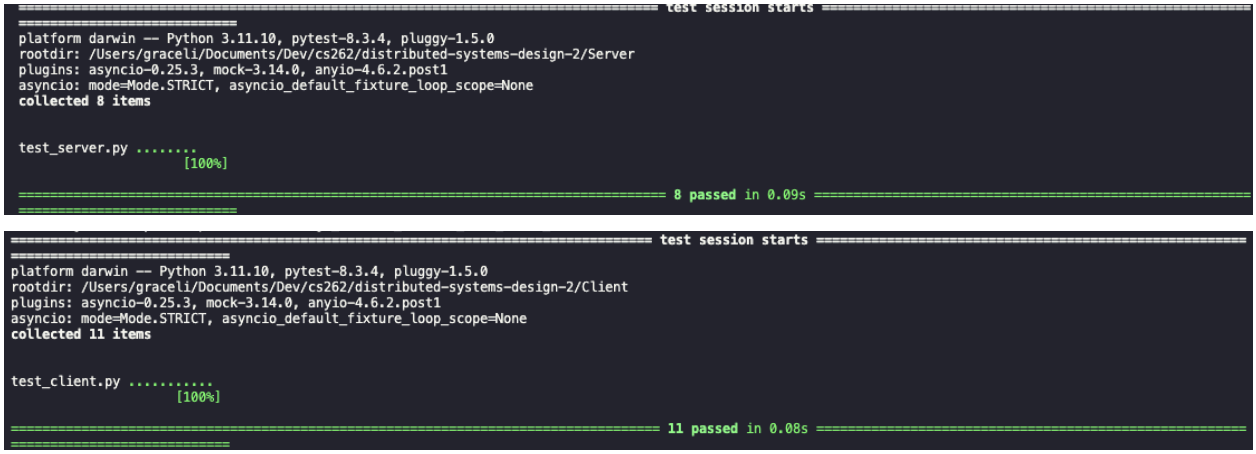
We created a comprehensive test suite to test both client and server side functionality with a total of 20 test cases. On the client side, we used mock objects to simulate and test server responses. We show some of those mock messages used to test pending messages below.

```python
def test_get_pending_messages(self):
    # Create mock messages
    message1 = MagicMock()
    message1.sender = "user1"
    message1.message = "Hello!"
    message1.timestamp = "2023-01-0 1 12:00:00"

    message2 = MagicMock()
    message2.sender = "user1"
    message2.message = "How are you?"
    message2.timestamp = "2023-01-01 12:01:00"

    message3 = MagicMock()
    message3.sender = "user2"
    message3.message = "Hi there!"
    message3.timestamp = "2023-01-01 12:02:00"
```

On the server side, we thoroughly tested the MessageServer class that implements the gRPC service. We show our successful test cases below.



While creating our tests, we wanted to ensure that all of the services that we provide had appropriate test cases as well. For these reasons, we were proud of our earlier decisions to make our services as modular as possible. It proved to be super helpful for ensuring that our code worked in the way that we expected!

Analysis of gRPC vs. Custom Protocols

Switching from custom protocols to gRPC turned out to be relatively simple, especially since we had kept our custom protocols as an abstraction layer instead of embedding them directly into the core functionalities of the code. For example, functions like user authentication were not tightly coupled with the custom protocol. By using helper functions to handle serialization and deserialization, we were able to isolate the protocol changes from the core actions and services. This made the transition smoother, and while we did need to adjust the UI in a few places to accommodate different data types, we found it easy to replace our custom protocol with gRPC without major issues.

One downside to using gRPC, however, is that it can be more restrictive, particularly when working with data that has a variable length. With our custom protocol, it was simple to modify delimiters to mark the start and

end of a list or dictionary. With gRPC, on the other hand, we have to use streams to handle this, which adds complexity. Managing iterators and keeping multiple streams open for clients became an additional challenge, which wasn't an issue with our custom protocol.

A major advantage of gRPC, though, is how easy it is to modify the protocol without requiring significant refactoring. Changing our custom protocol used to involve a lot of fine-tuning to ensure indexing wouldn't break, but with gRPC, we can add or remove fields easily because of its use of unique identifiers. This makes it much more flexible and scalable in the long term.

That said, one final downside of gRPC is that it can be heavier to transport, as it passes entire objects and iterators instead of just simpler character strings. This can be more resource-intensive, which might be a concern in environments with limited bandwidth. Still, the tradeoff for easier modification and cleaner structure made it a worthwhile choice for our project.

Conclusion

Thank you so much for taking the time to check out our second design exercise. We absolutely learned a lot during this process, and we really enjoyed being able to resolve and refactor our existing code. If you have any questions, we are happy to answer them at Kamryn Ohly (kohly@college.harvard.edu) and Grace Li (graceli@college.harvard.edu).

AI Disclosure

- **Helped summarize bulleted notes for our engineering notebook into more coherent and concise sentences.**
- **Assisted with writing tests and debugging UI bugs.**