[Repo link]

Re-implement the chat application you built for the first design exercise, but replace using sockets and a custom wire protocol/JSON with gRPC or some similar RPC package.

Does the use of this tool make the application easier or more difficult?

This tool makes the application much easier to implement than using sockets and selectors. With gRPC the data parsing becomes significantly easier as long as the types are consistent with the .proto I only have to worry about the actual high level implementation over the low level implementations. For our case, we couldn't figure out how to implement our "server push" where we could keep track of clients in the server and notify a client when a message is sent without the client specifically requesting the data. However, for this specific issue I wouldn't say it made it harder to actually implement as the logic was just different entirely.

As a side note, it is clear why this framework is widely used, especially with the communication between different microservices which use their own tech stack, frameworks, and libraries. This standardized, type safe way of sending serialized information appears to be one of the better advancements within system architecture.

What does it do to the size of the data passed?
Check the graphs to see that the size of the data is smaller than JSON

- Wire Protocol has the smallest data footprint, using **2,532 bytes (WP**) on average per operation compared to **2,612 bytes for gRPC** and **2,715 bytes for JSON.**
    - Clearly the size is Wire protocol < gRPC < JSON
    - HOwever message delivery was faster
- For high-performance applications: Wire Protocol is clearly the superior choice, with dramatically faster for the messages.
- Wire Protocol offers the smallest data footprint
- JSON shows the largest protocol overhead in terms of data size, which is expected given its text-based nature and additional formatting requirements.

How does it change the structure of the client?

In our case, the client structure changes only by introducing a thread that continuously polls the server for `InstantaneousMessagesRequest`, or when a user receives messages while logged in. High level implementations in other functions did not change for this design exercise as all the standard functions did not require constant information

from the server. In Design Exercise 1, we were able to refresh our client when the server sent new data without requiring the client to send a request first. Clearly, our approach with gRPC felt inefficient and less scalable, as our current model would overwhelm the server with client requests.

The server?

- The server needed to be refactored into a class, and because of that I needed to turn my helper functions for processing each different type of request into a class function
- Also, because gRPC uses threads to service client requests, I needed to add a lock on the "database" parts of the server (the big dictionary storing all the accounts, messageId, etc.) in order to avoid race conditions. This was not needed in the socket version because the selector processed requests one at a time

How does this change the testing of the application?

- Unit tests are easier because we can start the server in the unit testing file + use stubs
- Also less parsing and formatting needed for the tests, which is nice
- Still there are overall the same test cases and test formats, so it's not that different

Engineering Notebook

2/23/25

- Started reviewing gRPC videos to strengthen my understanding of the protocol
    - Google's remote procedure call
    - Sends type safe serialized information across
    - Utilizes .protofiles to compile
    - From the basics
        - Unary RPC:
            - Client sends a single request.
            - Server sends a single response.
        - Server-Side Streaming RPC:
            - Client sends a single request.
            - Server sends a stream of multiple responses
            - (Server pushing multiple data points to client).
        - Client-Side Streaming RPC:
            - Client sends a stream of multiple requests.
            - Server sends a single response.
        - Bidirectional Streaming RPC
            - Client and server both send streams of multiple messages.

Code to generate from protofile

- python -m grpc_tools.protoc -I./ --python_out=. --pyi_out=. --grpc_python_out=. message_server.proto

Planning for our use case:

- Me thinks bidirectional RPC is best for our use case so that the client doesn't need to "exit" after response is delivered
- It would be nice to implement the publish and subscription model discussed in class
- Tomorrow is finding out what I need for the client

2/24/25

- After further research it is possible to have the server send a message to the client, without the request from the client, but it was hard to find implementation details on the internet or through documentation
- Goal:
    - When Client A sends a message,
    - the server pushes it to client B's active stream
    - If the recipient is not online, store the message.
- https://groups.google.com/g/grpc-io/c/FKeg4yfB-Jo
- https://grpc.github.io/grpc/python/grpc.html
- https://stackoverflow.com/questions/56739947/what-is-the-mechanism-of-grpc-server-side-pushing
- https://github.com/itisbsg/grpc-push-notif/blob/master/client/client.go
    - He subscribes to alerts in go using the client, but implementation seems different here, so we probably have to use the context, need to read
- Perhaps the server code somehow retrieve the stub that is connected from the client and access its data.

2/25/25

- For Instantaneous delivery we were told to create a separate channel for messages to be delivered and stored in the server and have that checked every so often (pinged from the client)

  Server Side:

    - dictionary `self.instantMessages` to store messages by recipient username
    - Implement `GetInstantaneousMessages` RPC method that:
        - Accepts a username parameter to identify which messages to retrieve
        - Uses locks to prevent race conditions when accessing the message queue
        - Returns all pending messages and clears the queue for that user

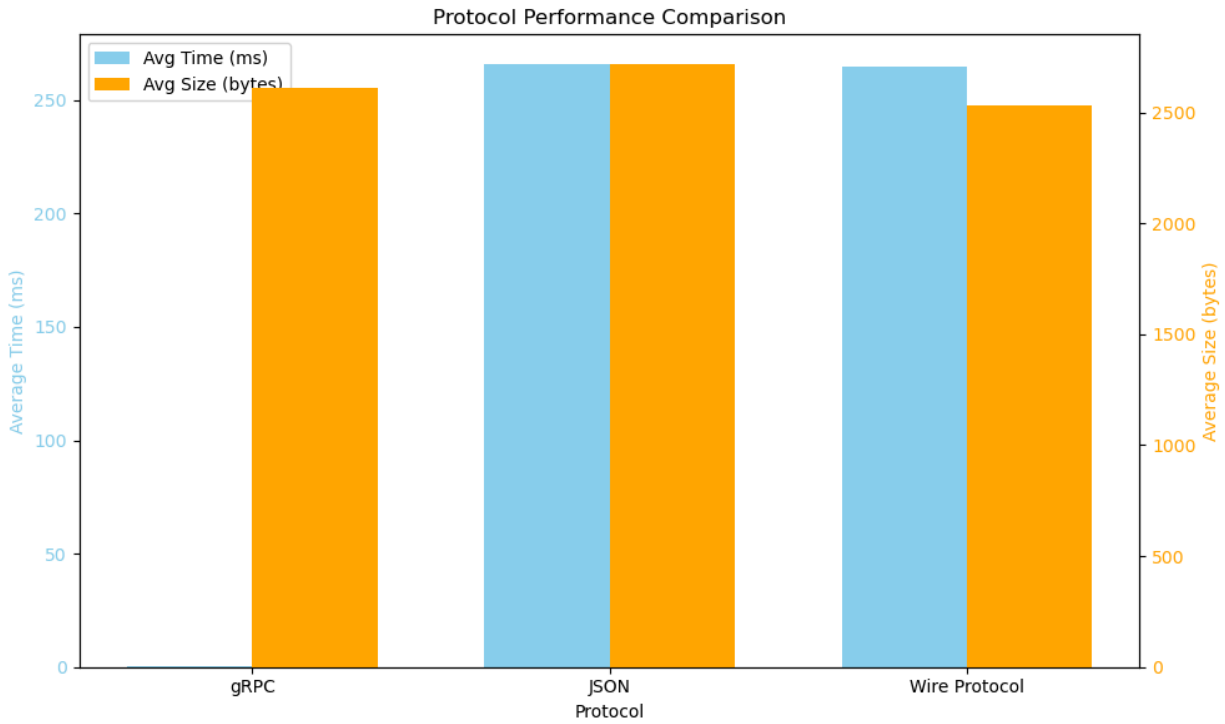
        - Formats messages with required fields (fromUser, time, message, messageId)
    - When a message is sent and the recipient is offline, it's added to this queue

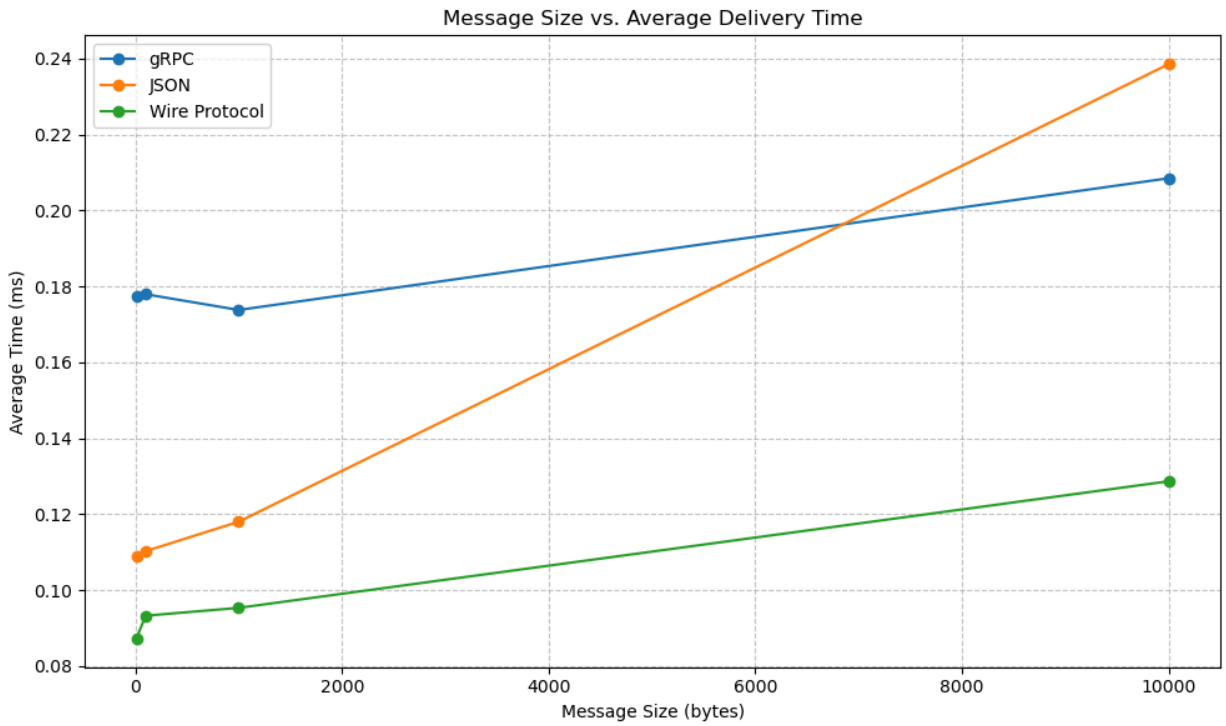
    Client Side:

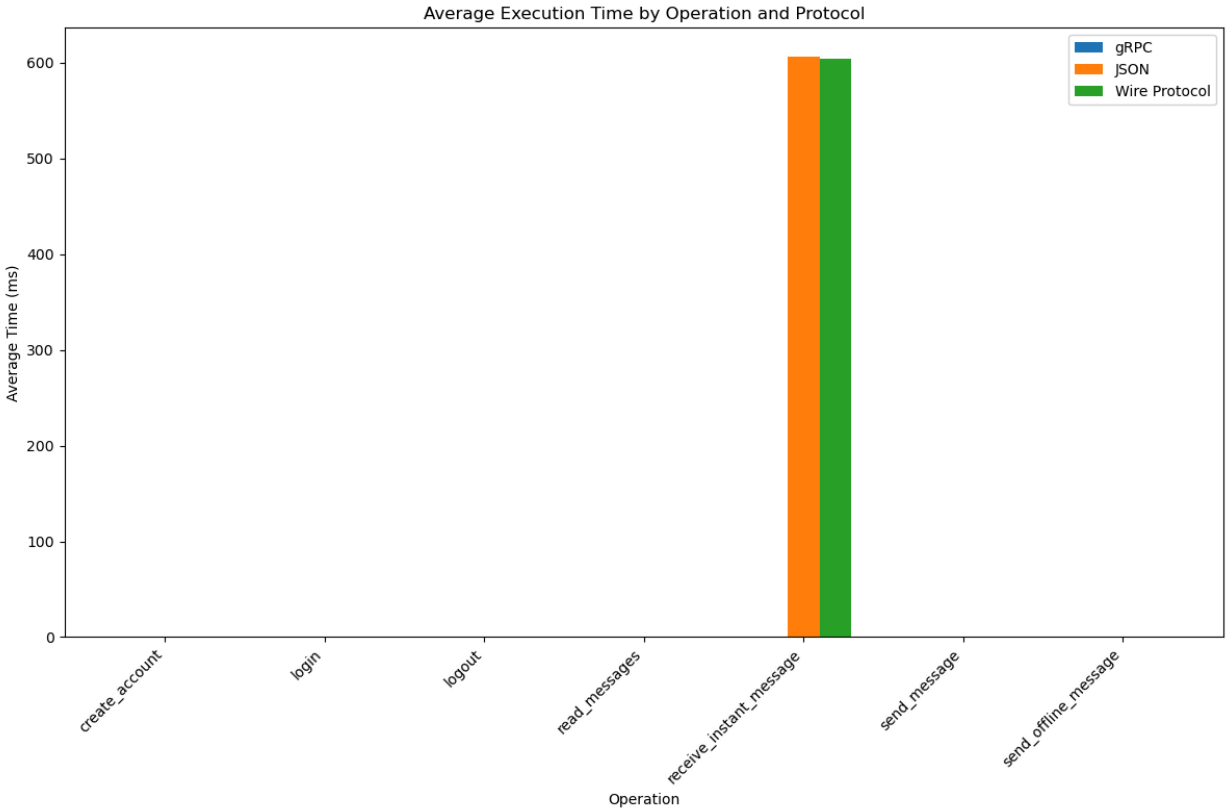
    - Implemented a `poll_messages` method that runs every 500ms using Tkinter after method (to help moderate how often messages are sent
    - Checks if user is logged in before making the RPC call
    - Calls server's `GetInstantaneousMessages` with current username
    - Processes any received messages and adds them to the client display
- Furthermore, we weren't able to apply this server push idea (sad)

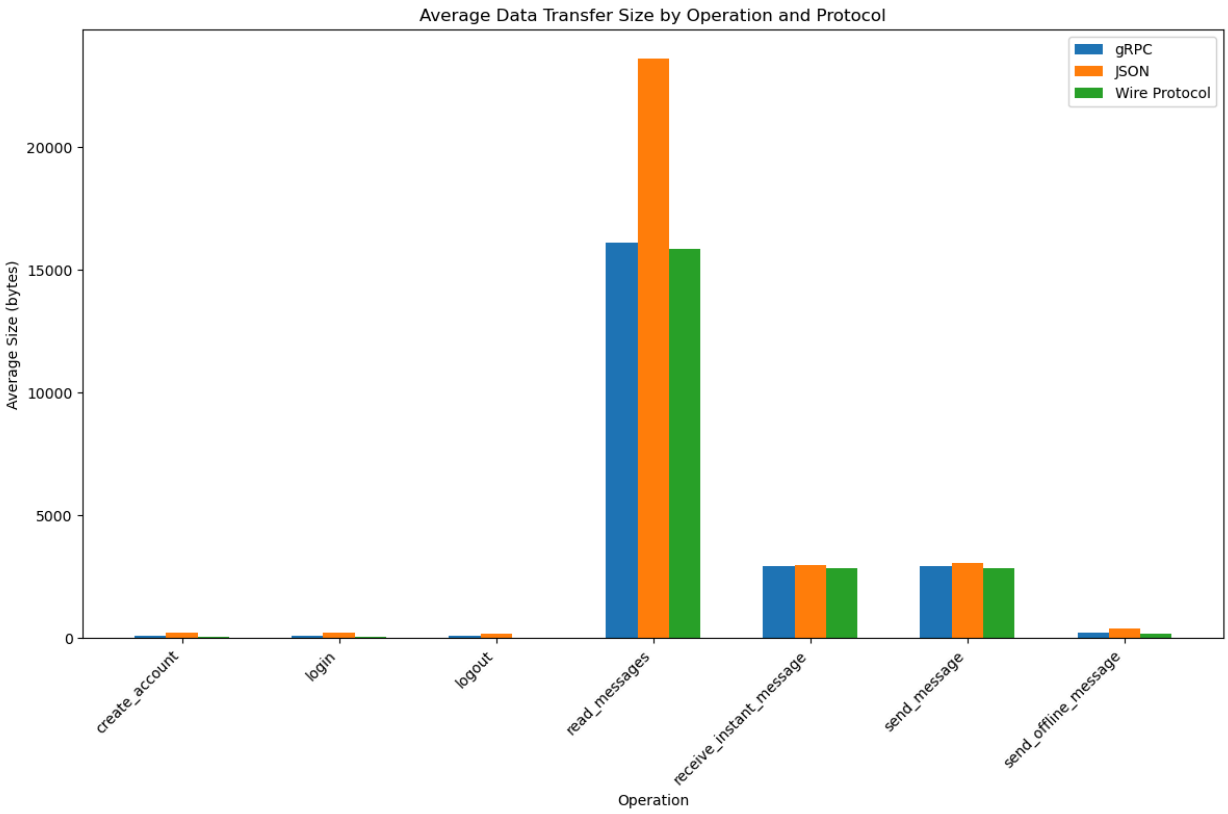
2/26/25

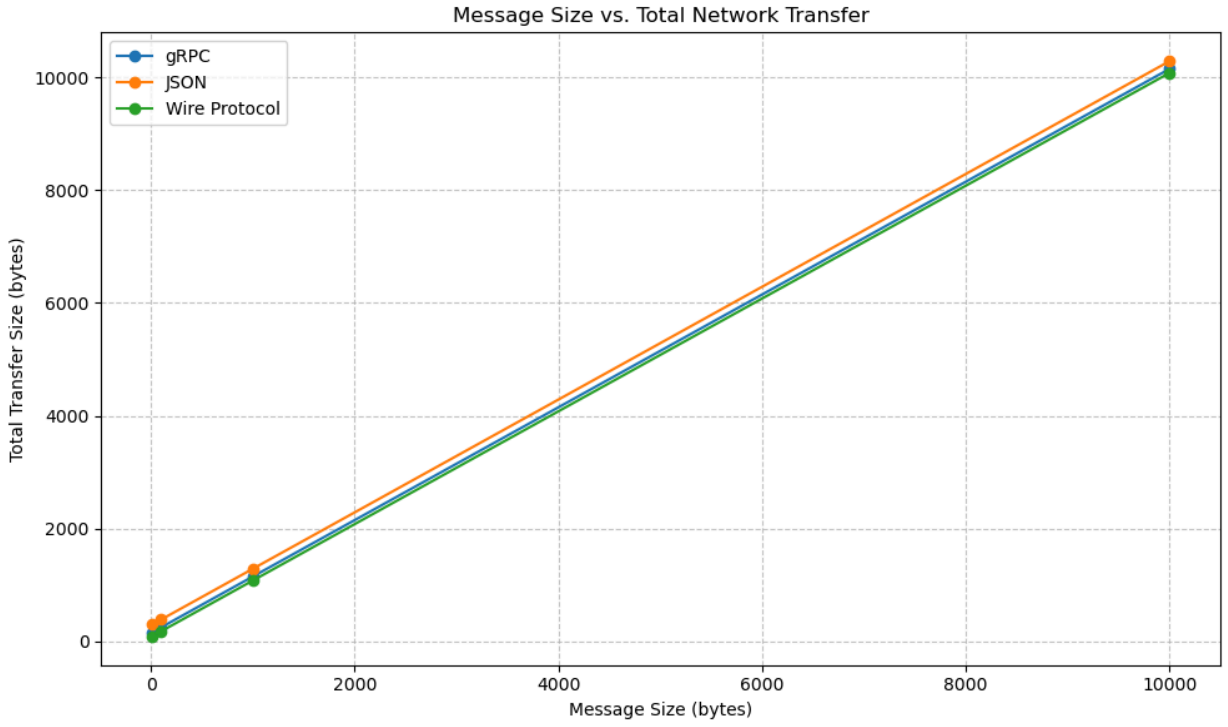
- Implemented experiments (This is also in the github)




Message Size vs. Average Delivery Time
gRPC
JSON
Wire Protocol
Average Time (ms)
Message Size (bytes)
0.08
0.10
0.12
0.14
0.16
0.18
0.20
0.22
0.24
0
2000
4000
6000
8000
10000


Average Execution Time by Operation and Protocol
gRPC
JSON
Wire Protocol
Average Time (ms)
Operation
0
100
200
300
400
500
600
create_account
login
logout
read_messages
receive_instant_message
send_message
send_offline_message


Average Data Transfer Size by Operation and Protocol
gRPC
JSON
Wire Protocol
Average Size (bytes)
0
5000
10000
15000
20000
create_account
login
logout
read_messages
receive_instant_message
send_message
send_offline_message
Operation


Message Size vs. Total Network Transfer
gRPC
JSON
Wire Protocol
Total Transfer Size (bytes)
0
2000
4000
6000
8000
10000
Message Size (bytes)

- For high-performance applications: Wire Protocol is clearly the superior choice, with dramatically faster for the messages.

response times for message operations while maintaining reasonable data transfer size.

- For limited bandwidth environments: Wire Protocol offers the smallest data footprint, which

could be beneficial in bandwidth-constrained scenarios.

- Protocol overhead: JSON shows the largest protocol overhead in terms of data size, which is expected given its text-based nature and additional formatting requirements.