CS 262 - Distributed Systems
Leticia Schettino
Adolfo Roquero

Engineering Notebook - Chat App

Plan for project

Version 0 - working solution in Python meeting all of the specs, console as the UI.
Version 1 - re-implementing v0 using gRPC and protocol buffers
Version 2 - layering UI with Flask on top of v0 and/or v1 (Work in Progress)

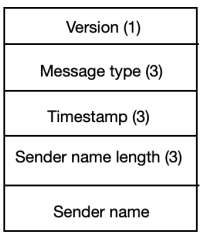
Design Decisions

Wire protocol specification:

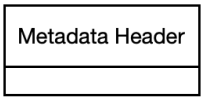
The comments in this section pertain to v0, where we use the wire protocol we defined and sockets for the communication between client and server. Below, are the details for the wire protocol we specified. Since our implementation is in Python, which is not a strongly typed language, all of the content passed back and forth are encoded strings. ‘Type checking’ to ensure valid form for the requests is done upon receipt at the server.

- Valid message types, specified in protocol.py file which both client and server have access:
    - CL_SIGNUP=’1’
    - CL_LOGIN = '2'
    - CL_LISTALL = '3'
    - CL_DEL_USER = '4'
    - CL_SEND_MSG = '5'
    - SRV_SIGNUP = '21'
    - SRV_LOGIN = '22'
    - SRV_LISTALL = '23'
    - SRV_DEL_USER = '24'
    - SRV_FORWARD_MSG = '25'
    - SRV_MSG_FAILURE = '26'

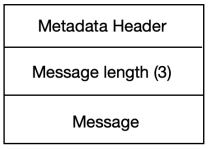
- Metadata header
    - This header is included with any message/request/reply being passed from server to client or from client to server. Within this header there is the protocol version, name of the sender (either client’s username or ‘server’, to identify a request or reply was sent by server), timestamp of header creation and the message type.



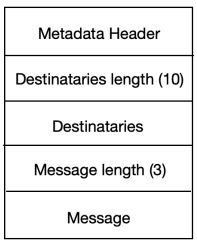
- Client request to login, sign-up and delete user (CL_LOGIN, CL_SIGNUP, CL_DEL_USER),
    - All 3 of these commands/requests from client to server are packed in the same way, since for all of these 3 requests to be completed, the only necessary information is the username of the client and message type, both of which are included in the metadata



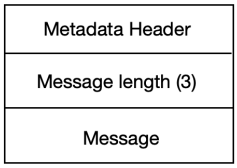
- Client request to listall (CL_LISTALL)
    - A user can request for the server to list all registered users in the chat application by running the ‘listall <filter>’ command, where the filter is written in terms of the two text wildcards ‘*’, ‘?’. To complete this request, the server once again uses the data from the metadata header (sender name and message type) as well as the regex string which is used to filter the usernames to be listed.



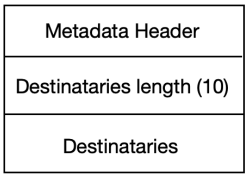
- Client request to send message (CL_SEND_MSG)
    - In order to send a message to other users, a user must specify who the destinataries are then write the message. This, along with the metadata header, is sent to the server in the request. There is a larger memory allocation for the header containing the length of the destinataries since the user could be sending the same message to a large number of users. The message length is capped and any characters after the character limit (999) does not get sent to the server in the client request.



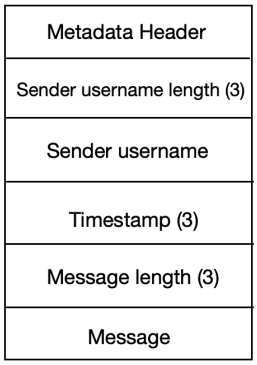
- Server reply to client login, signup and delete user (SRV_LOGIN, SRV_SIGNUP, SRV_DEL_USER, SRV_MSG_FAILURE)
    - Once the server receives a request from the user to sign up, login or delete a user, this request is processed and a reply is sent back to the client, which varies depending on whether the request was processed successfully or not. Regardless of whether the server could correctly process the request, the reply from the server to the client takes the format below, where the message includes the text the server sends back to the client to be printed for the user.
    - The distinction between whether the request was processed successfully or not is found in the metadata header, under the message type. For instance, the server replies to a client login request with either a SRV_LOGIN message type (in case of successful login) or SRV_MSG_FAILURE (in case login failed) and both replies have the format below.



- Server reply to client listall request (SRV_LISTALL)
    - When the server receives and processes a listall request, the reply comes in the format below, where destinataries is a string with the list of users that matched the username filter regex included in the client request.



- Server response to client send message request (SRV_FORWARD_MSG)
    - When the server receives a request from a client to send a message, it processes this request and if processing is successful, it ‘forwards’ the message from the client that requested the message send to all the valid listed destinaries. There is a timestamp field sent in along with this message which represents the datetime in which the user requested the message to be sent (while the timestamp in the metadata header represents the datetime when the server sends this message).



Decisions pertaining to functionality and usability

The spec for the project had ample room for interpretation, which meant we had to decide on how we wanted the system to work in specific scenarios. The most relevant design decisions we made with regards to how the app functions and/or how the user is supposed to interact with the application are highlighted below.

Login or Signup:

- Our design allows for one user to login from different clients and therefore hold multiple different client connections. This represents a user logging in from different devices, for instance a phone and a PC. Under this scenario, any message directed to this user (by username), is sent to all of the logged in devices.
- When a login or signup fail, the socket is disconnected and the connection is closed. In v0, to attempt another login or signup, the user must re-run client.py which prompts the creation of a new socket. In v1 (GRPC) the user is re-prompted to login or signup until a successful attempt goes through or the connection is disconnected by user. This difference in behavior depending on the version is a result of the greater simplicity to implement this behavior using GRPC.

Delete User:

- A user can only delete one self.
- In v0, once a user successfully deleted themselves, the connection is closed.
- All the pending messages sent to that user (deleted user) are deleted.
- If the deleted user has sent a message to another user that wasn't logged in, the pending message will still be delivered to the destinatary although the sender no longer exists.

Sending/Receiving Messages:

- Invalid usernames can be passed as destinataries to a message, and the server will simply ignore those invalid usernames
- The server only stores messages pending delivery, which is a message sent with valid destinataries (i.e. registered users). In v0, if a user is logged in, it will receive these messages instantaneously, otherwise the messages will be received (in chronological order) by the destinatary user once destinatary user logs into their account.
- The point above implies that if user x requests a message to be sent to a user with username y which is not a registered user according to the server, then even if in the future a user signups with username y, they will not receive the message sent by user x.
- Lastly, When a user logs in, they receive all of the messages that were sent to them since their last active session, including from users who have since deleted their accounts.

Listall

- There are 2 valid text wildcards that can be used together with the listall command: '*' and '?', with their usual meanings as in regular expressions defined by the Unix Filename pattern matching (https://docs.python.org/3/library/fnmatch.html).
- All username are LOWER-CAPPED for matching
- Examples:
    - "" matches everything
    - "*" matches everything
    - "ad*" matches users signed up as "Adelle" and "Adrian" but not "Alex"
    - "A*" does not match "Adelle" since username is lowercapped on signup

Technical decisions and details

Server: Use of non-blocking IO instead of MultiThreading

To handle concurrent requests from multiple clients, we had several options:

- Single-threaded non-blocking socket:

In a single-threaded program, to handle concurrent requests from multiple clients, the server can set its socket to non blocking mode, so that if a socket call on behalf of one client can not be processed immediately, other client requests are not delayed.

- Multi-threaded:

In a multithreaded server, instead of setting the sockets to be non blocking, the server can create a separate thread to handle each client request. Thus, if a call to a socket function by one thread blocks, only that client request is affected; other threads are free to continue processing requests from other clients.

We considered both approaches to be similar for a low number of concurrent client connections. However, a multi-threaded server is limited by the number of threads on the CPU (8 cores on a

Mac M1) which can limit the scalability of the system for a large number of simultaneous clients (which is likely in a ChatApp). Thus, we choose to use Non-Blocking IO.

After doing some research, some sources propose a combination of both approaches to handle a maximum number of concurrent requests, however implementing a Multi-threaded Non-Blocking requires an engineering effort that went beyond the timeline for this project.

In order to implement the non-blocking IO server, we use the `select.select` function which enables non-blocking polling such that only client sockets that are ready to be read from are returned.(https://docs.python.org/3/library/select.html#select.select)

Unit testing

- For unit testing, we used the python library *unittest,* which simplifies unit testing implementation.
- Our approach to unit testing the system is to run the server code, then run the test code, which replaces the UI. In our unit tests, we test requests which we expect to be both successful and unsuccessful requests for each of the requests the client can make to the server. We then proceed to verify both that the request sent from client to server is correct and that the server correctly processes the request and replies correctly to the client.

Miscellaneous code design

- Using environment variables to store values needed by both client and server, such as server's IP address and connection port. Included in the env.sh file within the git repository.
- Abstract common functions and variables needed for both client and server into shared files (utils.py, protocol.py) for improved code readability and more sustainable code maintenance
- In v0, wrapped client and server operations into 2 distinct classes for simplified unit testing.

Limitations and other considerations

One important limitation from this application is the lack of a more intuitive and functional UI. In our current model, the application runs on the console, which makes it difficult to expand the complexity of the features in the application. We had been working on v2 (initial skeleton code included in repository), using a Flask UI, though were unable to complete this implementation due to time constraints. One specific difficulty we had when building v2 (with advanced GUI) on top of v1 or v0 was that our decision of using non-blocking IO instead of multiple threads or asyncio on the client side meant we would have to re-implement v0 or v1 in order to add any form of pre-built UI (Flask, tkinker, etc).

Problems Encountered/Debugging steps

Wire protocol implementation for Sockets

- Our initial implementation of the wire protocol was directly implemented within both the Client and Server Classes which was very hard to track, update, expand and very tedious to debug
- As a solution, we abstracted the wire protocol via the protocol.py file which defined all of the constants used in the protocol (See above) as well as a util.py file that handles any encoding and decoding of messages as per the protocol.

Unit testing package (unittest)

- Our initial implementation of unit tests had cases being dependent from each other. E.g. Test Case 2 worked with users that had been added by Test Case 1. This error in the design coupled with the fact that Unittest doesn't run test cases classes in order caused us a lot of headaches. To solve this, we learned how to use the SetUp/TearDown and SetUpClass/TearDownClass to make each test case and test function clean up after running which made tests independent of each other.

Comparing GRPC to custom wire protocol

Code complexity

- The code complexity for the GRPC version of the application was significantly smaller than that for the custom wire protocol version. One small metric of complexity that we can use to highlight this difference is the number of lines in each of the files:

File	Wire Protocol + sockets	GRPC
server.py	~300	~170
client.py	~300	~70

- One increased complexity of programming using GRPC as opposed to the sockets and wire protocol is that without having experience with GRPC, there are a few different important concepts to learn, starting from how to correctly write and compile the .proto file. However, once this step is done, the skeleton code provided by the compiler is nearly most of the way done towards working code, missing only the function definitions.
- There is also greater ease to unit test the GRPC implementation since the classes

Performance differences

- While we did not time the exact differences in time to send/receive requests and replies from/to client/server, we noticed performance differences between the two versions when unit testing the code. This difference in performance was noticeable because when making repeated requests from the client to the server, with the intent of unit testing the code, it was necessary to add timed pauses between the request to ensure the server had received and handled the previous request before a new one could be sent in. We did not encounter these performance issues with GRPC.
- It is worth noting, however, that when using the app as a user through the user interface of the application, the performance differences between the custom wire protocol version and the GRPC version were not noticeable.

Buffer sizes

Another important comparison to consider is that using GRPC substitutes the need for the wire protocol and instead we have the proto file specification for messages and allowed functions. Below we have included the comparison for the number of bytes transferred from client to server and vice-versa. As we can see, for all requests the GRPC code is more efficient in its communication.

Call*	Sockets	GRPC
	Client Request	Client Request
Login	35	3
Signup	35	3
Listall '*'	39	3
Listall '*<random>'	47	11
Send message ""	49	26
Send message "Hello"	54	33
Delete user	35	3
Receive message	NA	3

*All communication to client/from server (and vice-versa) includes a metadata header, as defined in the wire protocol above. The sizes above consider the smallest possible username length (a single character) and destinatary lengths for the send message example. By keeping username and destinatary at the smallest possible size, we ensure a fair comparison between the message sizes for both of the wire protocol and the protobuf in GRPC.

The table illustrates how much more efficient the communication is using GRPC than the wire protocol we designed for the sockets.

When it comes to the buffer sizes (in bytes) sent, we can see that GRPC is much better in terms of space taken for all Calls.
Our socket implementation requires the use of metadata as well as headers to encode every message segment from string to its encoding. On the other hand GRPC makes use of the knowledge of the variable types within each message to efficiently serialize the messages. As a result, the GRPC implementation sends far less bytes over the wire. This is another advantage of using GRPC versus using a custom untyped Wire Protocol (which encodes from/to strings - Very inefficient in terms of space and parsing time)