COMP3331: Assignment 1

z5312055, Bonson Wong

October 29, 2021

1 Client-Server Communication

The main challenge of this project is designing a consistent mode of communication that will allow the client to simultaneously: recieve live notifications, send messages and setup asynchronous communication channels with peers. Using the given starter code, we assume that our application will follow the pattern of alternating between client request and server response. All design considerations are based on this premise.

Our client faces the first problem of being able to simultaneously parse commands sent by the user, and send update requests that check of messages, user activity, etc. This is handled by two separate threads on the client side. The main thread waits for user input and adds it to the command queue. The second thread will check the queue and proccess each command according to the specifications. Each command is parsed as a request, and if valid, it will be sent to the server. If the queue is empty, our thread simply sends out an update request to the server. This structure ensures that only one request is sent at a time, and that the client is sent updates regularly. Furthermore, when waiting for replies from recipients (i.e when requesting a private channel) our user can continue their activity until the recipeint responds. One downfall with the current design is that if the client sends commands constantly, they will not recieve any updates as the command queue is constantly full. To negate this, we can simply add an update request after every command our user inputs. Since our client is dictating the information flow of our application, the server side loop is much simpler. With virtually no deviation from the start code, our server simply waits for client requests, and responds according to the spec.

With the current implementation of the client, our second major problem is recieving and responding to queries sent by the server. This breaks our loop because our client is constantly receiving commands from the user, so we have no idea which input given by the client responds to which request sent by the server. To solve this issue, we leverage the functionality of our command queue. Since we only require a confirmation query from the user, we can append the two possible responses to the command queue. If the user responds one way, the other non-corresponding response is from the queue. The server can then process the client response as a normal request sent by the client.

2 Status codes and Request Methods

Status codes consist of only a confirmation, error, and connection end response. Request methods exist as four letter acronyms that correspond with user commands. Both are referenced as constants in their own separate files.

Loosely based on the SMTP protocol, our messages between the client and server consist of a status code/request method and a message body. Typically, a client message will store its parameters in the message body and any function corresponding to its request is stored as the request method. Responses made by the server will contain a status code indicating whether the client request has been processed correctly, and a message body that will be displayed by the client. Since most of the information on how the clients request was proccessed, we make the server responsible for generating a descriptive response. Although this simplifies our client side program, this requires more data to be sent by the server than is typically necessary.

3 Server Backend

To keep track of users and their activity on the server, several data structures are employed in our program. One such data structure is the User class. Each entry in our credentials file is instantiated as a User object where information on their name, password, incoming messages, and blacklist is stored. This list of users and other important information is stored and processed in the data.py file. This includes much of the functionality that keeps track of user logs and the list of client threads. ClientThreads contain information specific to the users current session. Relevant information such as the client address, socket, and timeouts are stored here.

Now mostly depreciated, auth.py handles processes that involve the credentials file itself. Adding new credentials, validating users, etc, can also be handled here. However, this requires opening and parsing the credential file each time, and is replaced by the functionality provided by data.py.

Messages are handled in message.py. Since we already have a message/notification queue for each user, this file is only responsible for appending and popping messages to the User class.

Other than setting up ClientThreads, server.py is responsible for temporarily blocking users and disconnecting inactive users. Both are handled by keeping separate threads that update either the time inactive of each user or the time left they have before they are unblocked.

4 Peer to Peer Connections

P2p connections requires our client to somewhat take the role of a server here. Since multiple connections cannot be sustained on a single socket, we must sustain a separate socket for each peer. This requires us to update the client to support a separate thread for incoming connections and a list of peers to interact with. On the server side, we store our client's welcoming port for potential peers. After accepting a private connection request, the server will respond with the senders welcoming port and address so our client can setup a new peer connection. Similar to our server, we will create a separate thread for handling the socket for another peer. Named PeerThread, this process is responsible for recieving and displaying messages sent by the other peer. On top of this, it will terminate itself when the peer logs out.