CS 4390.002 Team Report

Project: Server Based Chat

May 1st, 2023

Team 10
Zia Kim · sxk180181
Gregory Post Davis · gap170001
Jejun Park · jxp163630
Noah Castetter · nbc170001

Table of Contents

Introduction

The objective of this project was to create a server-client chat application that utilizes UDP connection for client-server connection initialization and TCP connection for client-client connection and chat. To achieve this, we implemented socket programming and multithreading in Python. Our server program efficiently manages incoming connections, validates and authenticates clients, encrypts and decrypts messages, and keeps track of multiple chat sessions concurrently. Additionally, we implemented error handling and logging to ensure the reliability and stability of the system. Through this project, we gained valuable experience in designing and implementing networked applications using socket programming.

Files

- Two main python files for execution: *server.py* and *client.py*
- Three supporting files: *authentication.py, encryption.py,* and *chatHistory.py*
- Previous iterations: available on GitHub
- Video: available on YouTube

How to run

Also included in the compressed file as *README.md*

1. Open two or more terminals and locate the project folder.
2. Run code with command *python server.py* in one terminal.
3. Run code with command *python client.py* in all other terminals.
4. Enter username and secret key.
5. Observe as the TCP connections are made. From the client terminals, type in commands. Following are available commands:

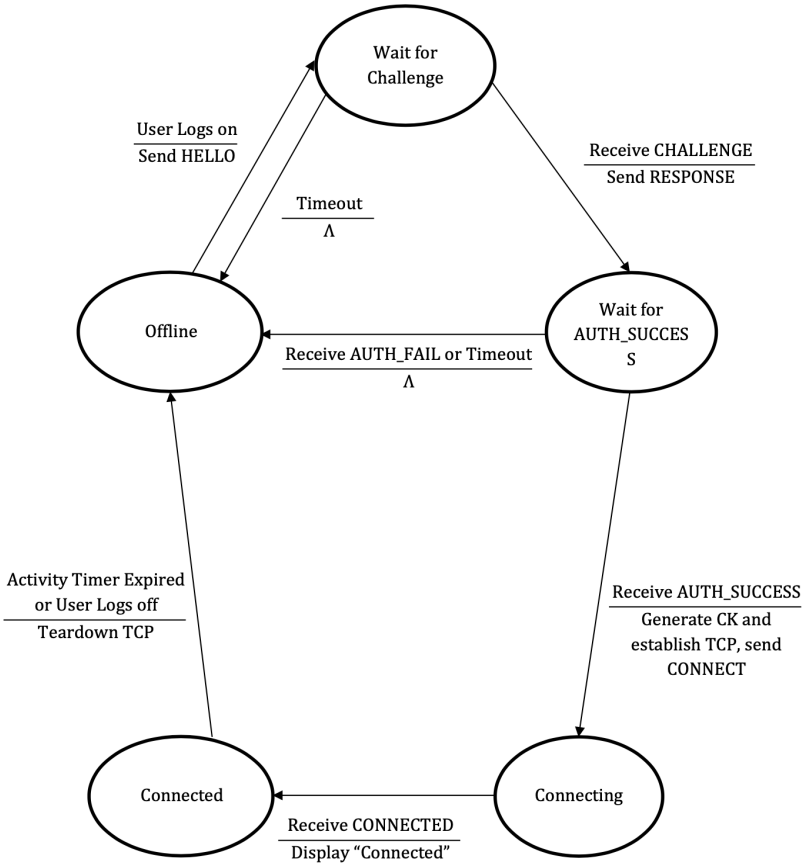
a. *chat client-id*: start a chat session with client-id.
b. *end chat*: end the current chat session.
c. *history client-id*: display the history of past messages with client-id.
d. *log off*: tear down TCP connection and end connection.

Following are some aspects of the program to be aware of when testing:

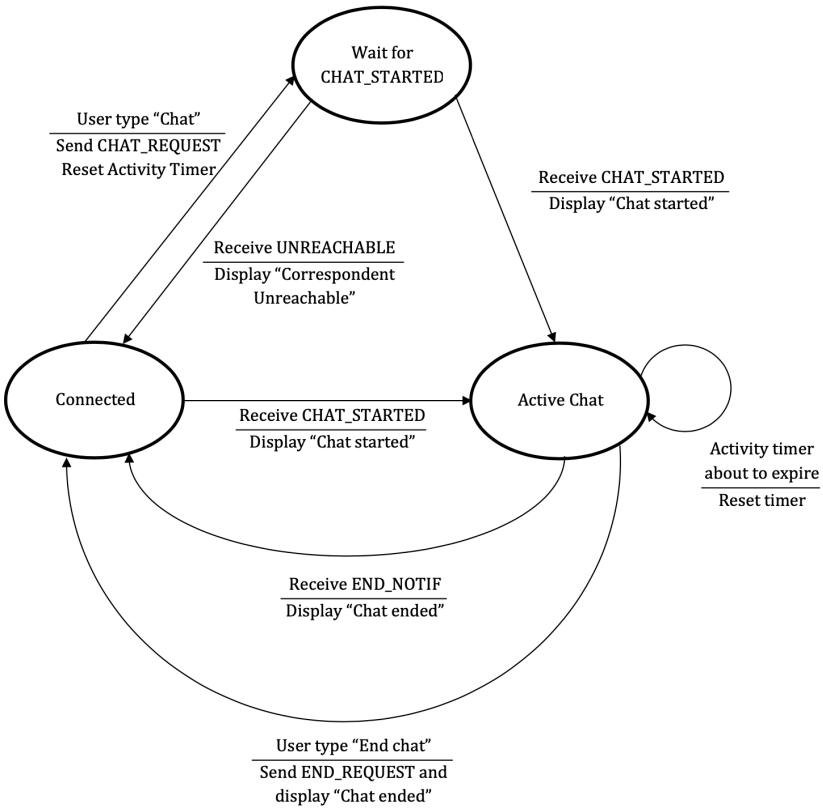
1. The program doesn't consist of the subscription step as it is out of the scope of the project. Therefore, there is a list of pre-'subscribed' users and their secret key, implemented for simplicity of the verification process. They are (clientA, 100), (clientB, 200), (clientC, 300), and so on up to clientJ, counting up to 10 clients for clear validation. Use these to test.
2. *Server.py* doesn't terminate on its own. The UDP welcoming socket stays open so clients can create a TCP connection or tear it down at any time. In order to force the UDP socket closure, use ctrl+C.
3. Although in a real-world setting, each client would have their inherent client-ID assigned by the server at subscription time, it wouldn't be efficient to have ten different *client.py* files. Therefore, we designed a single *client.py* file that allows the user to input their client-ID (username) and the secret key (password) that can demonstrate various users logging onto the server.

Protocol State Diagram — Client

Connection phase:

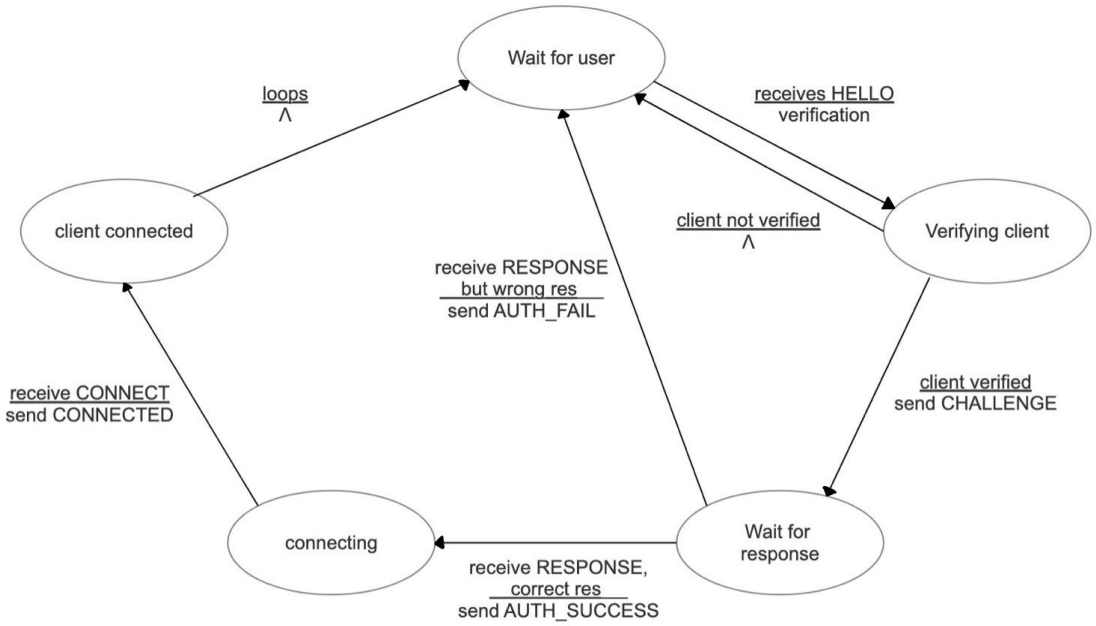


Chat phase:

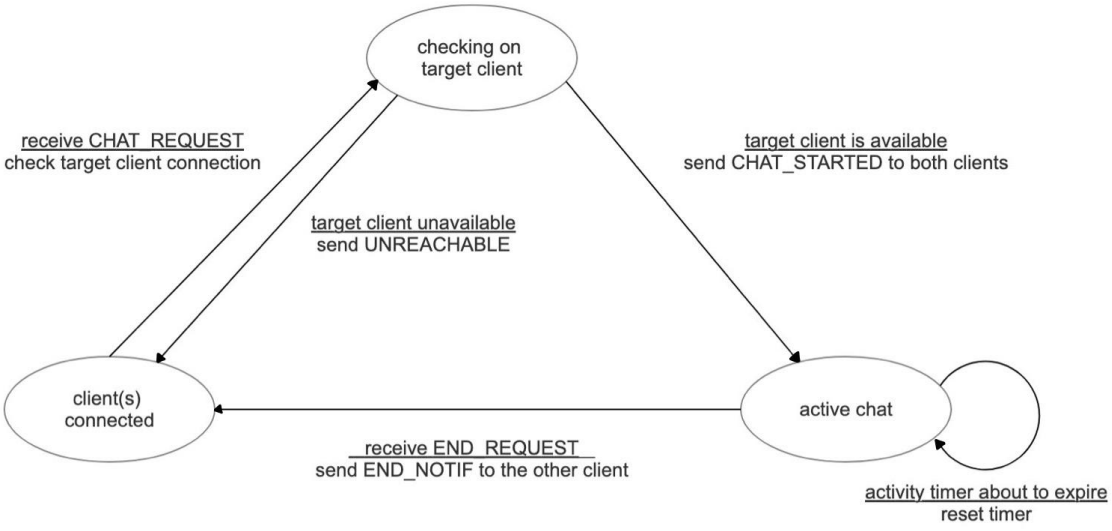


Protocol State Diagram — Server

Connection phase:



Chat phase:



Authentication Scheme

The *authentication.py* is responsible for the authentication of the client. Authentication is based on the challenge/response mechanism which is composed of two basic components: a challenge and a response. The goal of the challenge is to require a response that only authorized clients will know. Clients that successfully answer the challenge are allowed access.

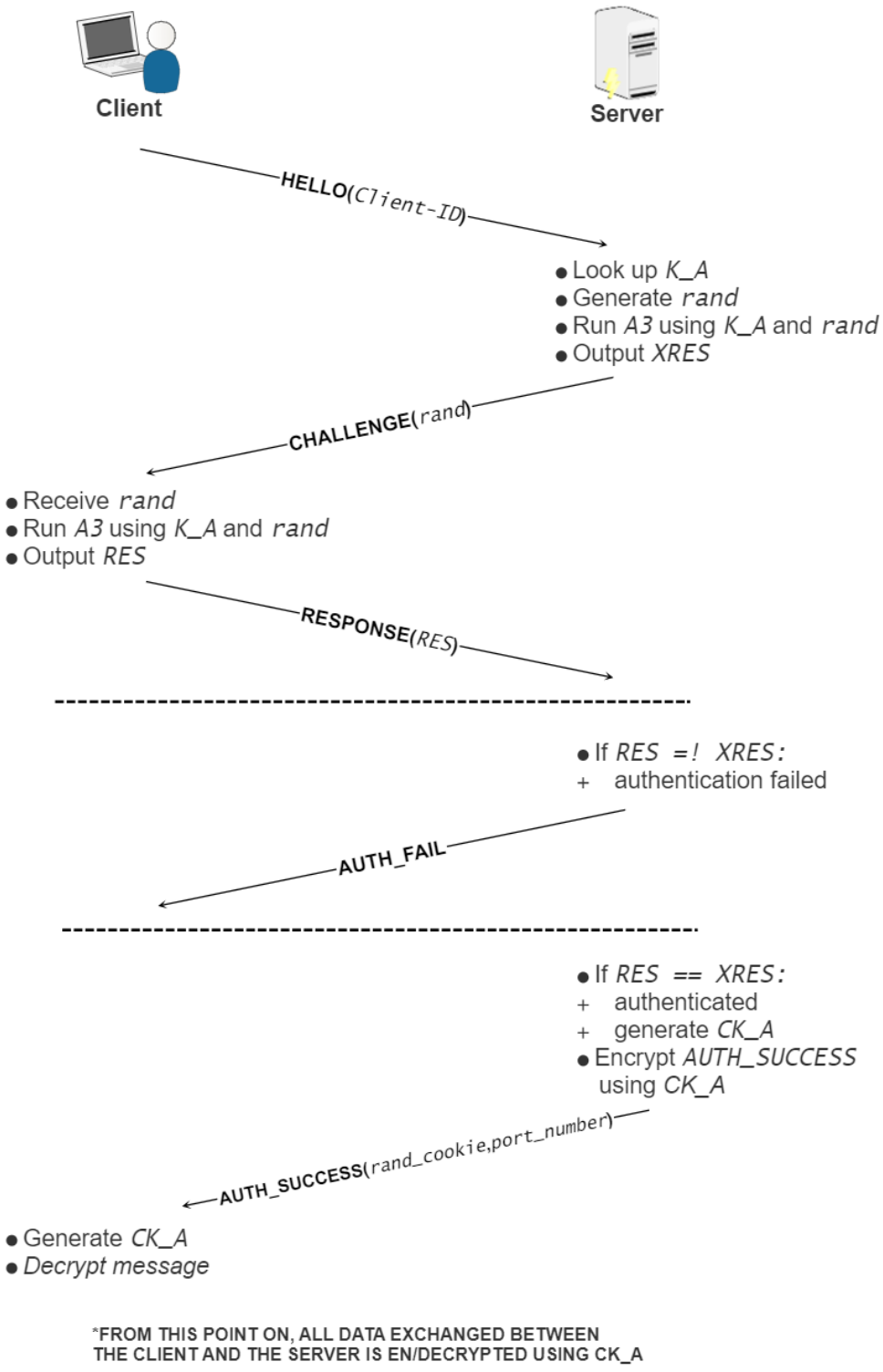
Initial phase of the authentication process begins with the incoming client sending *HELLO(Client-ID)* to the server. Then, the server extracts the *Client-ID* from the message and looks up the client's secret key(*K_A*) associated with that *Client-ID*. The server passes the *K_A* to the *server_hash* function in the *authentication.py.* The *server_hash* function, also known as *A3* algorithm, takes K_A as an input, generates *rand* which is a random integer of 10 digits, and concatenates *rand* and *K_A* to create a string called *hash_input*. This *hash_input* is then processed by the hashing algorithm SHA256 to create *XRES*.

Next, the server sends *CHALLENGE(rand)* to the client which extracts *rand* from the message. Then, the client passes its *K_A* and *rand* to the *client_hash* function, also known as *A3* algorithm, which concatenates *rand* and *K_A* into a single string called *hash_input*. This *hash_input* is then processed by the same hashing algorithm SHA256 again to create *RES*. The client takes *RES* and sends it back to the server as *RESPONSE(RES)*.

In the next phase, the server receives *RESPONSE(RES)*, extracts *RES* and passes *RES* and *XRES* to the *compare_hash* function in the *authentication.py*. The *compare_hash* function compares *RES* with *XRES*. If *RES* and *XRES* are different, the function returns *0*. This means the client has failed to prove its identity and thus failed the authentication process. The server sends *AUTH_FAIL* back to the client and the authentication process terminates. If *RES* and *XRES* are identical, the function returns *1*, meaning the client's identity is authenticated by the server and the server generates *CK_A*, which is the key used for encrypting and decrypting messages. For a detailed process on how *CK*_A key is generated, please refer to the **Encryption Scheme** part of this document. Lastly, the server sends *AUTH_SUCCESS (rand_cookie, port_number)* back to the authenticated client.

At the same time, the client also generates *CK_A* key and receives the encrypted messages *AUTH_SUCCESS* from the server. Finally, the client decrypts the messages and extracts *rand_cookie* along with *port_number*. From this point on, any other messages are encrypted and decrypted using *CK_A* until the client tears down the TCP connection with the server.

Authentication Schema



Encryption Scheme

The *encryption.py* is responsible for the creation of the *CK_A* key and en/decryption of messages. Encryption and decryption are techniques used to secure data and protect it from unauthorized access or interception. Encryption is the process of converting plain or readable data (plaintext) into an unreadable form (ciphertext) using an algorithm or mathematical formula. This ciphertext can only be read or decrypted by a client who has the key or knows the decryption algorithm. Decryption, on the other hand, is the process of converting ciphertext back to plaintext using the key or the decryption algorithm. Decryption allows the recipient to read the message in its original form.
Encryption process happens concurrently on the server side and on the client side. On the server side, when the client is authenticated, the server calls *cipher_key* function, also known as *A8* algorithm, in *encryption.py* and passes *rand* and *K_A* to it. The *cipher_key* function takes *rand* and *K_A* and concatenates into a single string called *hash_input*. Then hash_input is processed by the hashing algorithm SHA512 to create *CK_A* key.

Next, the server calls *encrypt_msg* function in *encryption.py* and passes *CK_A* and a plaintext message *AUTH_SUCCESS(rand_cookie, port_number)*. This function takes *CK_A* as an encryption key and runs the XOR encryption algorithm on the plaintext message which generates an encrypted message. By using *CK_A*, the server encrypts a plain text message *AUTH_SUCCESS(rand_cookie, port_number)* into a ciphertext and sends it back to the authenticated client.

Simultaneously on the client's side, the client also calls *cipher_key* function, known as *A8* algorithm, in *encryption.py* and passes *rand,* which was sent by the server, and *K_A* to it. The *cipher_key* function takes *rand* and *K_A* and concatenates into a single string called *hash_input*. Then hash_input is processed by the hashing algorithm SHA512 to create *CK_A* key.

Then, the client receives an encrypted *AUTH_SUCCESS(rand_cookie, port_number)* message from the server, calls the decrypt_ *msg* function in *encryption.py* and passes the received message and *CK_A*. The *decrypt_ msg* function takes *CK_A* as an decryption key and runs XOR decryption algorithm on the encrypted message which generates a plaintext form of the *AUTH_SUCCESS* message. Finally, the client extracts *rand_cookie* and *port_number* from the decrypted *AUTH_SUCCESS(rand_cookie, port_number)* message. From this point on, all the messages exchanged between the server and the client is en/decrypted using *CK_A*.

Later on, both clientA and clientB connect to the server and initiates a chat session. Since clientA and clientB have different *CK_A* keys, the server works as a translator between the two clients. When clientA types a message, it triggers *encrypt_msg* function. This will encrypt the input using clientA's *CK_A* key and sends the encrypted message to the server. Upon receiving the encrypted message from clientA, the server decrypts the ciphertext by calling *decrypt_msg* in *encryption.py* using *CK_A* key associated with clientA. Then the server encrypts the plaintext message by calling *encrypt_msg* using receiving client's *CK_A* key which is clientB. Next, the server relays the encrypted message to clientB. When clientB receives encrypted messages from the server, it triggers *decrypt_msg* function which takes clientB's *CK_A* and the encrypted message and decrypts the message into its original form.

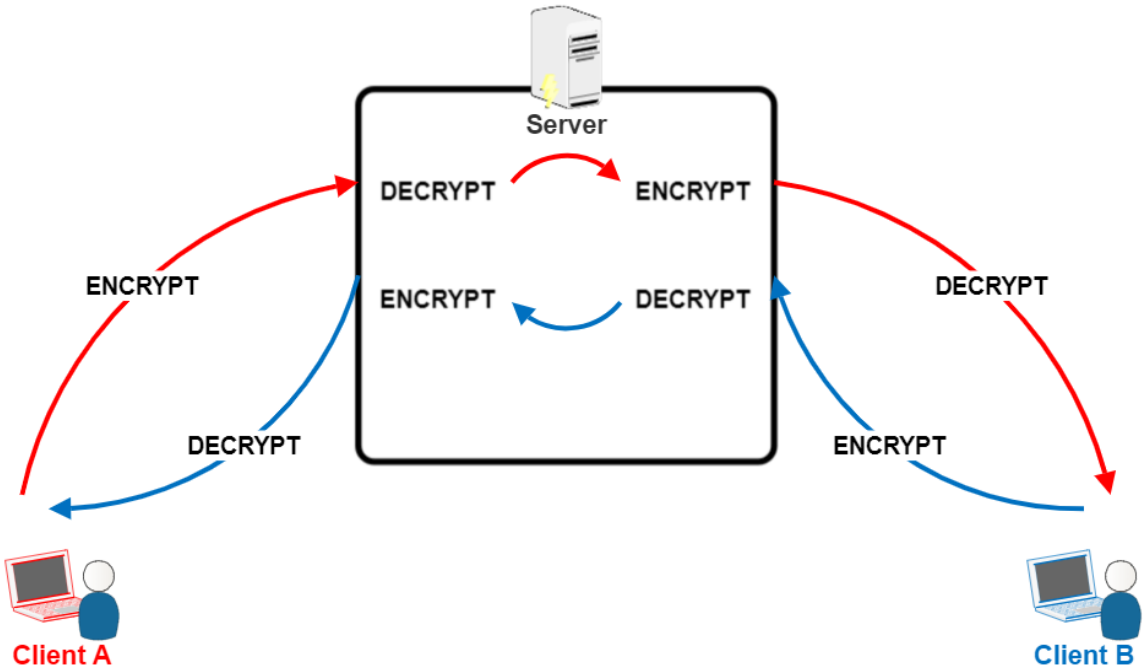
The same process applies to clientB as well. ClientB encrypts user messages using its own *CK_A* and forwards it to the server. Then, the server decrypts the message using clientB's *CK_A* and encrypts again using clientA's *CK_A*. Lastly, the server sends the encrypted message to clientA which clientA decrypts the incoming encrypted message using its own *CK_A* key.

Encryption Schema

* Run concurrently with authentication process

- Run *A8* using rand and *K_A*
- Output *CK_A*

- Run *A8* using rand and *K_A*
- Output *CK_A*



Hardware setup and configuration

We used personal laptops as our primary development machines. Due to the fact that all of our development environments are different, we had to deal with multiple different blocks throughout the development process. Following were some of the hardware configuration of machines used in the process:

- Windows 10, intel core i7, 16GB of RAM
- Windows 10, intel core i5, 16GB of RAM
- Mac OS Ventura 13.3.1, intel core i5, 16GB of RAM

We assigned static IP addresses to *server.py* and *client.py* in order to ensure the best consistency in our tests. Network issues that affected the stability of the application, due to firewall settings, router settings, or Wi-Fi signal strength were dealt with using active logging and error handling.

Issues and lessons learned

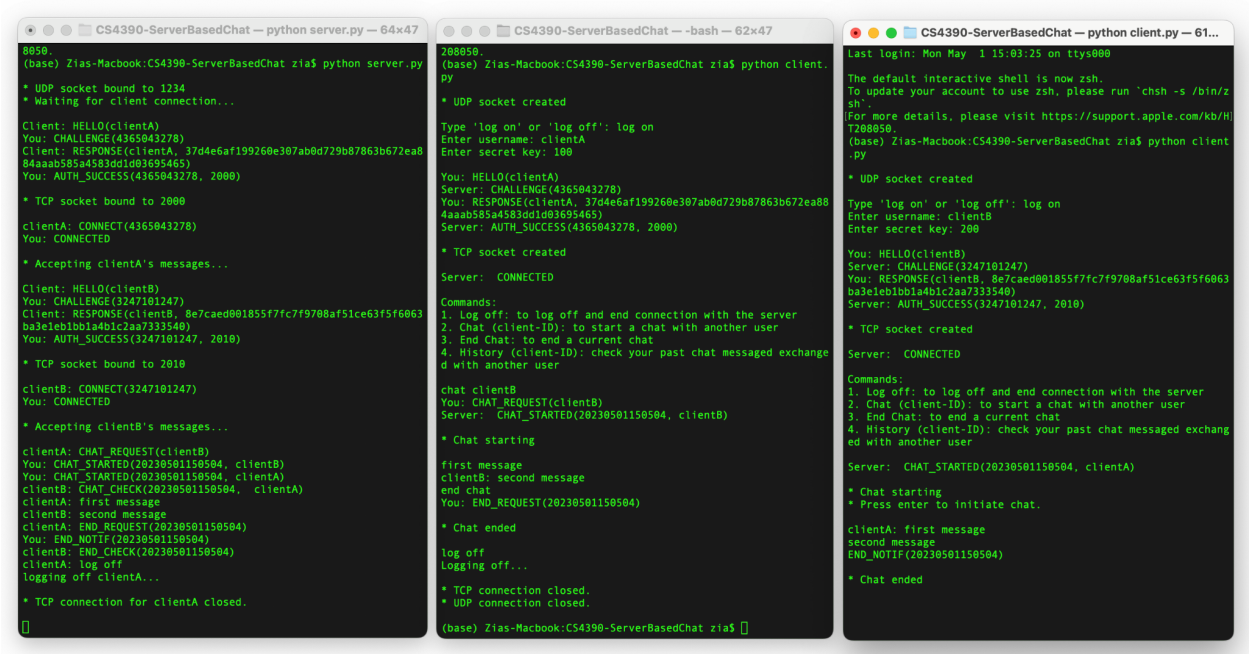
1. Our first attempt of TCP connection management by multithreading caused a lengthy block to the team's progress midway. This code would run on Windows machines but not on Mac and Linux machines, which stopped half of the team from progressing further in coding the client-to-client chat. The problem turned out to be the fact that we were trying to generate TCP connections within threads, when we should accept the connection in the main thread, then spawn the worker after the connection happens.
2. The overall development process of the program was very linear. For example, implementation of message history couldn't be worked on before client to client chat was implemented, and client to client chat couldn't be implemented without a single client connection was functional. As four individuals with limited experience in group development, we faced the challenge of finding effective ways to divide the tasks among the team members without falling into bottlenecks. To overcome this, we utilized agile development techniques, such as weekly meetings and daily communication through Discord to ensure everyone was aware of the progress and any blocking issues. We also made use of version control tools, such as Git, to manage code changes and avoid conflicts.
3. The team misjudged the amount of time and effort it would take to implement certain features, particularly a smooth client to client connection, which caused us to fall behind schedule. As a result, we had to spend more time debugging and wrapping up the program, although we had planned the last week to be dedicated to testing and writing reports. Nonetheless, the project gave us a better understanding of how to handle group project timelines better and to allocate additional time for unexpected roadblocks.
4. There was far more documentation on the now deprecated "_thread" in favor of "threading" so having to choose between using an out of date model or going with the one with fewer examples and less documentation.

Future improvements of the project

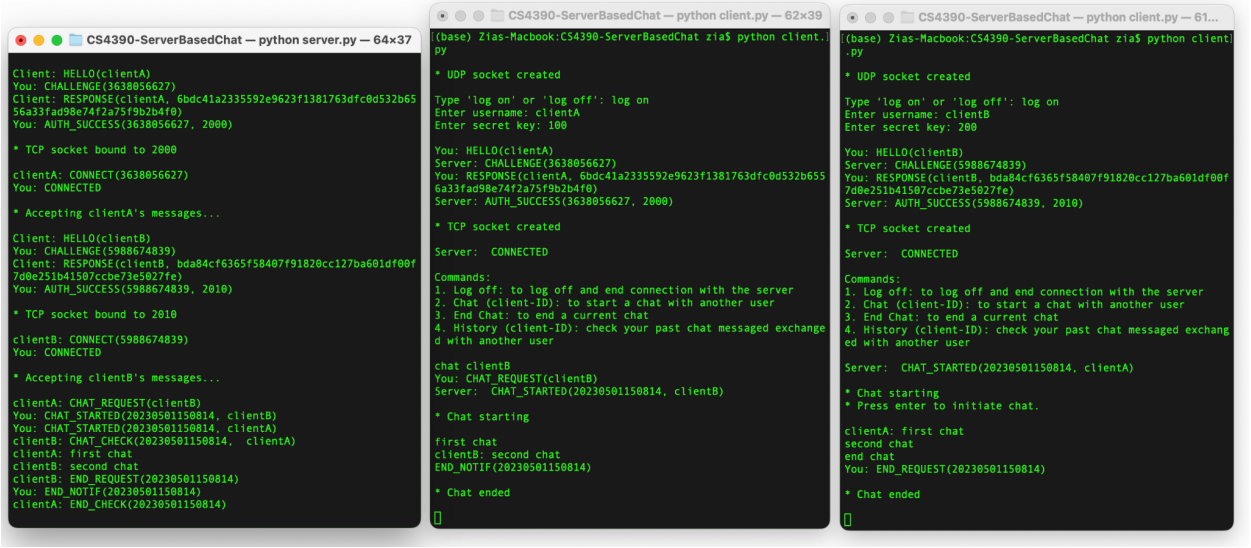
If groups had been assigned or topics had been picked earlier, the students would have more time to start to familiarize themselves with the programming language or the concepts relevant to the project. The project was assigned a week before spring break, which only meant that students would go on a break immediately after getting some grasp on the concept. While other circumstances caused delay in our project, stretching out the process likely would have been very useful to us.

Chat session demo

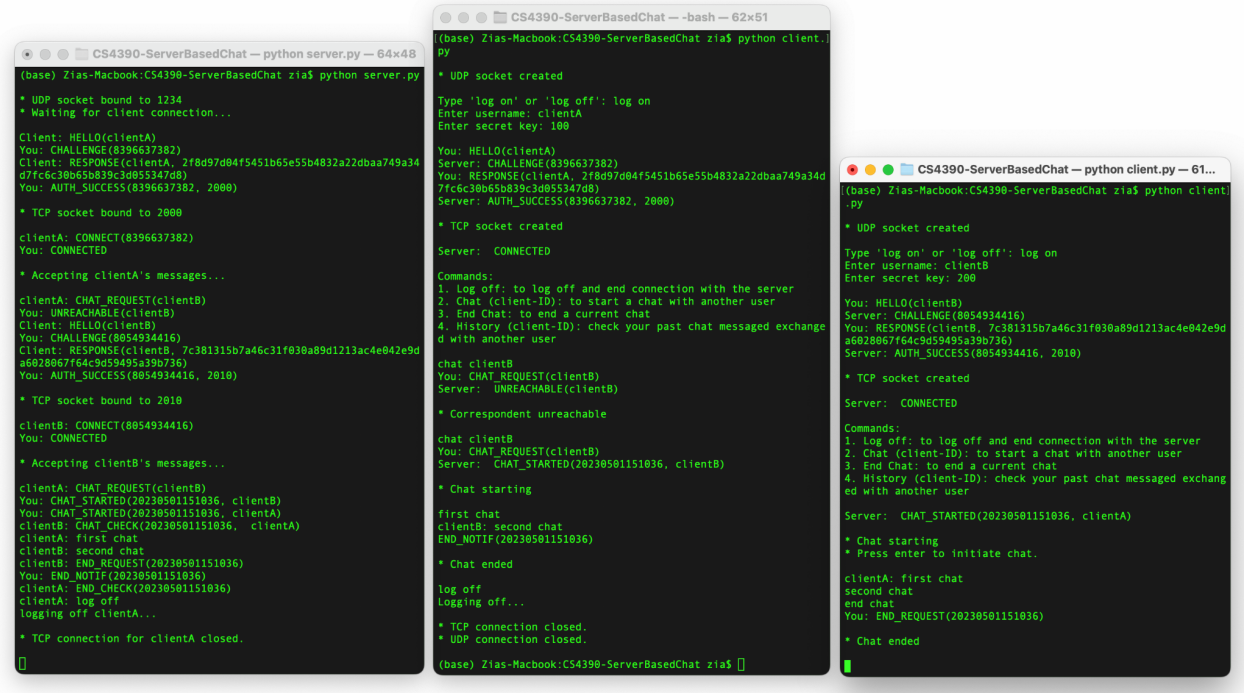
1. Basic chat initiated and closed by A



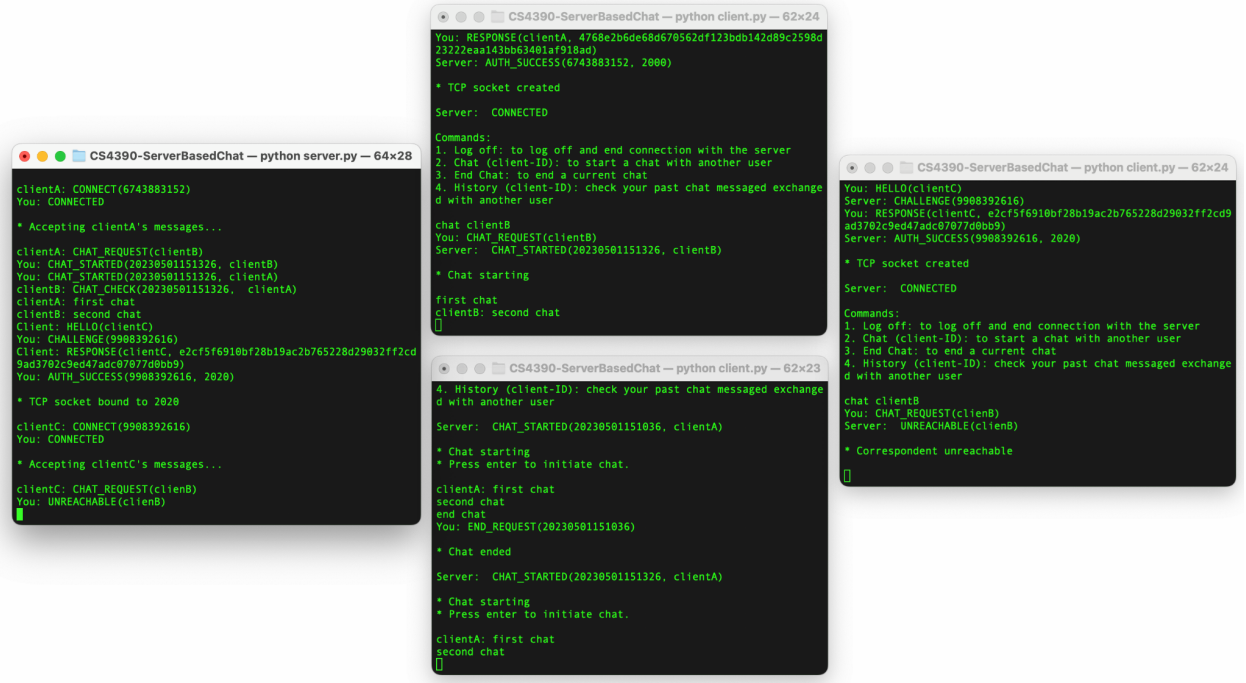
2. Basic chat initiated by A and closed by B



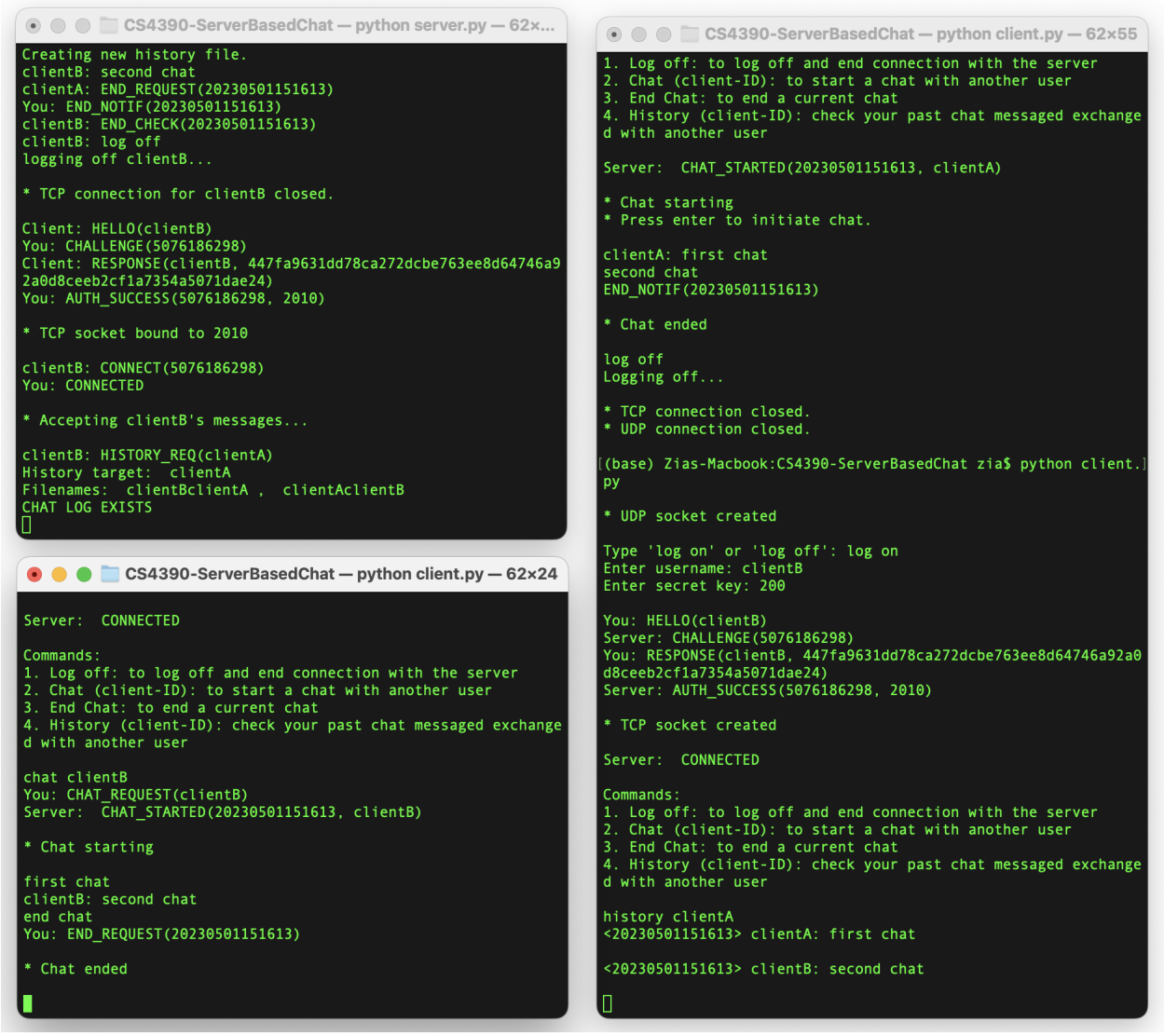
3. Basic chat initiated by A, but B initially not connected



4. Basic chat initiated by C, but B is already in another chat



5. History recall



6. Simultaneous chat sessions

For viewing convenience, the screenshot includes two sessions.