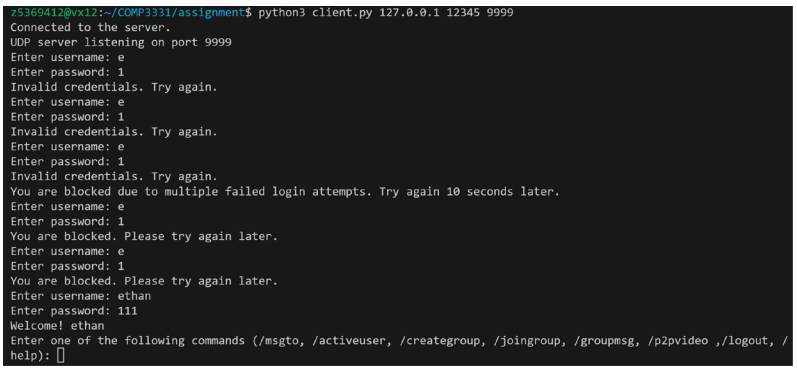
Introduction

The purpose of this program is to develop an online chat service that supports direct and group messaging and P2P video transfer. The client and server talk to each other using TCP sockets. When a client tries to connect to the server, the server accepts the connection and starts a new threading of the `handle_client` function. The `handle_client` function then calls `authenticate` and `handle_command` functions to handle the client's authentication and requests.

Authentication

In the authentication phase, the client is prompted to enter a username and a password. The server loads credentials using `load_credentials` function and checks with user input. If an invalid username is given the server is immediately prompted with an error message and prompted to enter username again. If a valid username enters the password wrong `max_attempt` times, the username will be blocked from logging in again for 10 seconds, even from another IP address. The server will prompt "You are blocked due to multiple failed login attempts. Try again 10 seconds later." The client is allowed to continue entering username and password because the blocking is username-based, not session-based. During the blocking, if another valid username and password are entered, the other user is allowed to log in. However, the blocked username is not allowed to log in. In the below example, e and ethan are both valid users, but e entered the wrong credentials 3 times and is blocked. But ethan can still log in.



Authentication and blocking are done by recording the number of attempts of a username in the server and once it reaches the maximum, the username is blocked and the unblock time is stored in the server. The server will check each time when a user

tries to log in. The authentication returns the username of the successfully authenticated user.

Command

Upon logging into the server, the client will be prompted with a list of valid commands and can send commands in formats such as "`/msgto`". The server handles clients' commands using the `handle_command` function. The function returns an integer, indicating the status of the server's socket.

Each time a user sends a command to the server, it prints the message to the terminal in format:

```
"Received command: {command} from user: {username}."
```

When user issues a valid command but the wrong argument number, the server replies with a message to the client indicating the correct usage such as:

```
'Usage: /logout\n'
```

And the server prints corresponding messages to the terminal.

If the user inputs an invalid argument (e.g. sends a message to an inactive user), the server will reply with the corresponding message indicating the error and print the corresponding message to the terminal.

If the command and the argument (if any) given are all valid, the server replies with a corresponding message, and prints a message to the terminal:

```
"User: {username} successfully created the group chat: {groupname}."
```

Each time the server receives a user command, it immediately generates a timestamp for the command for later use (e.g., message timestamp).

The Client and P2P Video

The client program initiates a UDP socket and TCP socket for P2P and server use. It uses `select.select` function to listen to the reply from the server and stdin simultaneously.

Unlike other commands, the `/p2pvideo` command is implemented on the client's program because it uses UDP sockets to communicate among clients and does not involve the server. The implementation assumes that the user calls `/activeuser` before `/p2pvideo`, it stores the list of active user every time the `/activeuser` is called for `/p2pvideo` use.

When a P2P transfer is initiated, the sender will create a UDP socket and first send:

```
"Start sending file:{file_name}"
```

to indicate the beginning of the transfer. It will then send data in chunks. When the transfer finishes, it will then send:

```
"File send has finished."
```

The receiver keeps listening for incoming UDP packet and tries to decode it, if the decoded information matches the beginning signal, it will then create the file and start to write the following bytes into the file. Each time before it writes the incoming bytes, it tries to decode them and see if they match the ending signal, if so, it will finish writing. It will then print a message and prompt the user to enter a command.

The sender pauses 0.001s after each chunk is sent to allow some time for the receiver to decode and write. This is a slight trade-off between reliability and speed. Without the pause, the receiver cannot decode and write the incoming bytes properly which results in corrupted files being received. The speed is around 16Mbps which is not greatly compromised, sending a 16MB file takes around 8s. Nevertheless, this pause can be easily modified to accommodate for more speed or reliability, it is stored in `myconstant.py` as `UDP_SEND_PAUSE`.