COMP3331 Assignment Report

Zeal Liang z5325156

Program design:

- For the server, I first parsed the command line parameters in the main program and then maintained a daemon thread. This daemon thread continuously listens for tcp connections from the corresponding port in an infinite loop, and creates new threads to handle subsequent work. The purpose of using an additional daemon thread is to solve the problem of ending the current program with Ctrl-C without repeatedly restarting the terminal during testing.
- To cope with the multiple threads for clients, I designed a parent class clientThreadManager to store their shared data. For example, the list of active users and blocked users, as well as the UDP port information for each client. The corresponding getter and setter functions are also available.
- The main part of the server that handles requests is in clientThread.py. I have designed many helper functions in here, such as recieveSocket and sendSocket. In the run function, I have handled different commands from the client. It's all very clear. All the code that reads and writes files is surrounded by locks to prevent multiple threads from competing for resources.
- On the client side, I also use ArgumentParser to handle the command line arguments first, and then access the server to authenticate after setting up the sockets. When the authentication is successful, the client creates a new thread to run a UDPServer for P2P connections. On the client side the user can also enter help to get an introduction to the command
- The main process then waits in an infinite loop for the user to enter a command.
  The different commands are handled separately in a clear and logical way, basically first handling the wrong input and then collating the relevant information to send to the server, and then waiting for a reply. It is worth mentioning that for the "UVF" command, the client creates a new thread to process the upload file. This allows the main process to continue executing other commands from the user at the same time.

Application layer protocol details and data structures:

- The server and client communicate via TCP. For example, when the client sends a request to the server to upload a file, the sequence is:
  - Client:
    1. pack a json with command name and fileID and dataAmount and fileSize. calculate the size of this json packet and send this size to the server via struct.pack (a 4-byte packet)
    2. then send the header json packet.

3. then send the files to be uploaded in batches in a loop

- Server:
  1. The server first receives a 4-byte packet, unpacks it and gets the size of the header to be accepted next.
  2. Then run clientSocket.recv(headerSize). to get the header. Decode the header to get the file information, and then create a new file according to the file name in the information.
  3. Then receive 1024 bytes each time in the loop and writes to the file at the same time and maintain a counter. When this counter is greater than or equal to the size of the file recorded in the header. The loop ends. The same process is followed to reply to the client.

Trade-offs considered:

- In the client-side peer-to-peer UDP server, Instead of accepting data and writing data in the same time, I cached it into a list and then immediately continued accepting remaining packets. After all packets are accepted, the file is written in a new loop.
  The reason for this is that UDP is an unreliable protocol. Although there is no packet loss over the road for local transfers(loopback), there is an upper limit to the receiver's buffer. When sending large files (several Gigabyte), If the speed of processing data on the receiver side is less than the speed of sending data on the sender side, eventually the receiver's socket buffer will overflow, which means that packets will be lost. Then because not enough data is received, the loop will not meet the condition, resulting in a jam.
- In addition to separating the processing data from the receiving data part, I also set the UDP receive buffer to 1Gb. This way at least no packet loss occurs when I test transferring a video file of 3 Gigabyte size.