COMP3331 BitTrickle Assignment
Ryan Wong – z5417983

Section 1: Code Organisation

My implementation of BitTrickle used python due to the vast number of libraries which are well-supported with good documentation. My directory structure is simple, in the parent repository, I have a README.md and a src directory. In src, the client and server files as well as the helper files are there. I then have 4 subdirectories, 3 for mock clients (yoda, hans and vader) with their files which they wish to publish and 1 for the server (which has credidentials.txt). When running tests, clients would be in their own working directory to simulate their own computer, and call client.py. Similarly, the server would be executed from the server directory.

Section 2: Program Design and Data Structure

The two main files would be client.py and server.py. To improve code clarity and reusability, two files: server_helper.py and client_helper are also used and included in server.py and client.py respectively.

Server Design:

I chose to use a single threaded server due to multithreaded being unnecessarily complicated due to locks being needed. The server first parses the command line input port, then reads from the credentials file, storing it in my data structure, initialising everyone as inactive with no published files (explained later). It then opens its UDP socket, bound to localhost and the given port. Then finally, it enters the main loop which runs until ctrl-c. In each loop, it listens for packets and executes the relevant functions like auth(), get(), sch() included from server_helper, sending a response to the client and printing to the server terminal if a message is sent or received. After this, or if no packet is received, the server just goes to the next step, which is client timeout checking. It checks if the time now is greater than time timeout time of each active client, making them inactive if it is.

To store user and file information, I wrote a class called User in server_helper which stores a bool: is_active, string array of published file names, integer of TCP welcome port and datetime string of timeout time. On server startup, it made a dictionary, where the keys were each client's username from credentials.txt, and the values where instances of User class. E.g. {"hans" : object1, "vader": object2}. This allowed simple and easy search and retrieval of information like active peers, published files, who published a file.

Client Design:

On startup, the client parses command line input port, starts a UDP socket for client server communication as well as a TCP welcome socket for p2p communication. It then enters an authentication loop which prompts the user for username and password, sending this to the server over UDP. If successful, it exits the loop, starting two daemon threads, one which controls the TCP welcome socket and listens for connections, the other sends heartbeat packets to the server every 2 seconds. After the auth loop, it enters the main interactive loop, listening for all commands lpf, lap, xit, sch, get, pub, unp, sending the corresponding packets to the server. After this, it listens for received packets, and depending on the command again, it does its task like printing active peers, downloading a file etc.

When "get <file_name>" is called, and a OK is received with another peer's TCP port, it opens a new file in the receiving client's directory and sends first the name of the wanted file to the other peer. It then receives and writes chunks of 1024 bytes to the file.

Like server, client.py uses a helper file client_helper.py which has only a class definition. This class is called PeerConnectionThread, which inherits Thread class from the threading module. When a connection is received on the TCP listening thread, it creates an instance of this class, passing it the connection object. This class's overridden run() function opens the file from the peer's directory and sends it as bytes through TCP to the other requesting peer.

Section 3: Application Layer Protocol

Client-Server Message Protocol:

REQUEST structure:

- In all lowercase
- Where "command" = get, lap, lpf, pub, sch, unp, hbt
- data could be "filename" for pub and unp, "substring" for sch
- data is optional (e.g. lap doesn't need data)

---

command username\n

datetime\n

data

RESPONSE structure:

- Again, all lowercase
- Data would be like the active peers, publish files, file search return
    - All on same line, separated by a **space**
- Error message and data is optional (e.g an OK doesn't need an error message, and an ERR doesn't always include data)
- "Result" is either OK or ERR

Result\n

datetime\n

Error msg\n

Data

Peer-Peer Message Protocol:

This message is only used when transferring files

First, just the name of the file is transferred from client peer to server peer:

"file_name.txt"

Then, the server immediately starts sending bytes. No connection termination message is needed as when the client receives no data, it terminates the socket.

Section 4: Known Limitations

1. Packet loss or corruption
   The server-client UDP connection is NOT reliable and does not support retransmission, buffering, ACKs/NACKs for lost or dropped packets. This is because we assumed perfect conditions, but in the real world, user experience would not be great as they could potentially send commands to the server receiving no response at all. Or if a heartbeat packet is dropped, the server might think the client went offline, which could lead to unexpected behaviour.
2. Heartbeat mechanism
   The 3 second timeout and 2 second hbt frequency are random and static, which could lead to bad and undefined scenarios if the hbt packet is delayed past 3 seconds, indicating a timeout, but then a hbt is received.
3. Same named files
   BitTrickle currently doesn't support files with the exact same names, but different contents.
4. Database
   Doesn't use a reliable database like PostgreSQL/MongoDB etc. text file is unreliable for credentials, and a dictionary of client objects doesn't scale well and querying is SLOW. Furthermore passwords are kept in plain text.