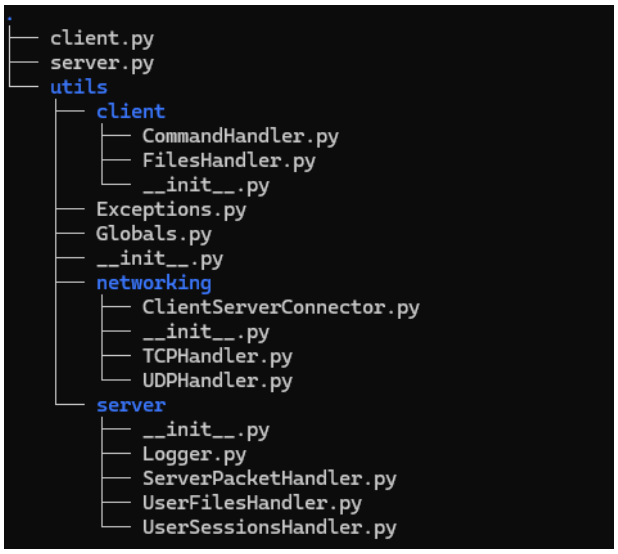
BitTrickle Report - Connor Li (z5425430)

1. Codebase Details

The program is written in python. The structure of the codebase is the following:



```
.
├── client.py
├── server.py
└── utils
    ├── client
    │   ├── CommandHandler.py
    │   ├── FilesHandler.py
    │   └── __init__.py
    ├── Exceptions.py
    ├── Globals.py
    ├── __init__.py
    ├── networking
    │   ├── ClientServerConnector.py
    │   ├── __init__.py
    │   ├── TCPHandler.py
    │   └── UDPHandler.py
    └── server
        ├── __init__.py
        ├── Logger.py
        ├── ServerPacketHandler.py
        ├── UserFilesHandler.py
        └── UserSessionsHandler.py
```

2. Program Design

The entry-points of the program are server.py and client.py. The following are the main components and their interactions:

Common

- There are several common modules used by both the client and server to aid in operations. These are:
    - **UDPHandler:** contains several classes including the UDPPacketHandling and UDPPacket. UDPPacketHandling has methods for both forming a UDP packet with the necessary fields, as well as retrieving data from fields in the UDP header or application protocol header. UDPPacket has field sizes and offsets, which provides a centralised way of keeping track of the header's structure. It also has a variable for storing the total size. There are also classes extending TypedDict that contain packet-specific payload structures, as well as a class for each type of packet request with methods to form the packet and parse the packet to retrieve a TypedDict.
    - **TCPHandler:** Simple class storing constants for TCP (packet size, encoding format).
    - **Exceptions:** Used to store application-specific exceptions.
    - **Globals:** Stores global variables (like enumerated packet types, and server ip / client ip).

Server

- Creates an instance of ServerPacketHandler which is given the UDP packet received from a client, and then carries out different operations depending on the packet type (GET, HBT etc.)
- This module interacts with two other main component instances of UserFilesHandler and UserSesssionsHandler. As the names suggest, these have various methods to handle files published by peers, and sessions of active peers.
- Depending on the outcome of the operation, the handlers within the ServerPacketHandler instance will either raise an exception to be caught in main, or return a UDP response packet formed with the necessary data.
- Any packets received or sent get logged using the NetworkLogger class within Logger. This has two methods, log_received_event() and log_sent_event()

Client

- The initial connection to server and authentication of the client is handled by a module called ClientNetworkHandler, found within ClientServerConnector. Its purpose is to continue to get authentication details until an OK packet is returned from the server. Once this occurs, it spins up a thread to send heartbeat packets, and a thread to listen for incoming TCP connection requests. It then returns the client->server UDP socket back to main.
- An infinite loop is used to continuously get commands until program shutdown (initiated by the xit command). The CommandHandler class has static methods for both getting the command from the user and executing said command. If the command is not valid (i.e. wrong number of inputs or non-existent), then it raises an exception which is caught in main.
- Otherwise it executes the command by calling the command's handler in CommandHandler. The basic structure of the handlers is that they create a request packet, with a specific payload (once again, determined by that packet's class in UDPHandler), send it and then get the response. Most packet types will print out a success or failure message depending on the type of response received.
- The GET request makes use of certain file-related logic which is contained within FilesHandler.

3. Data Structures Design

Client Session Storage

- Python dictionary of type dict[str | tuple[str, int], UserSession].
- The reason it can take either str or tuple[str, int] as a key, is that usernames are not always passed in a request packet. So to be able to find the session just from the IP and port is useful.
- Dictionary was used because it has O(1) lookup time. It's probably much more likely that lookups will be done rather than a traversal of the entire values set (using lap for example), so the interpreter's optimisation of list/array traversal compared to dicts is less of an issue. For example, to get the listening address during GET, first the username is retrieved and then their session has to be found using this.

Published Files Storage

- Python dictionary of type dict[str, list[str]].
- This creates key-value pairs between a filename and a list containing usernames of peers who have shared a file with that name.
- Dictionary used for similar reasons to client session storage.

4. Application Layer Design

The following is the application layer protocol message general structure for UDP:

[UDP HEADER] [MESSAGE TYPE] [PAYLOAD SIZE] [SRC IP] [PAYLOAD]

Where:

- UDP HEADER is just the std UDP header (src port, dst port, length, checksum) (8 bytes)
- MESSAGE TYPE is the type of message (GET, HBT etc.) as a number (2 bytes)
- PAYLOAD SIZE is the size of the data payload (2 bytes)
- SRC IP is the IPV4 address of the source (4 bytes)
- Total header size including UDP header and application-specific header: 16 bytes.

The UDP payloads are structured for each type of message as follows:

Message Type	Request Payload	Success Response Payload
AUTH	“Username,Password,Listening_Port”	“”
HBT	EMPTY	N/A
LAP	EMPTY	“Username1,Username2,...,UsernameN”
PUB	“Filename”	“”
LPF	EMPTY	“Filename1,Filename2,...,FilenameN”
UNP	“Filename”	“”
GET	“Filename”	“Publisher_IP,Publisher_Port”
SCH	“Substring”	“Filename1,Filename2,...,FilenameN”
ERR	N/A	“”

- Everything above is a string encoded using utf-8, and the rule is just to comma separate different data items. This makes parsing easy as the string is just decoded and split on “,”

5. Known Limitations

There are several known limitations, these are:

- If a file is published, but is then deleted, then that behaviour is undefined
- If there are several files with the same name but different contents
- If a client crashes or otherwise manages to exit when a file is transferring, the half-downloaded file remains.
- If the server is closed but the client is still active, and then the server is restarted, the client will continue sending HBT packets but will receive ERR packets in return, as the client is no longer authenticated. Ideally there should be a mechanism to log out if an ERR is received in response to a HBT.