Project Report

Program Design

The program is split into two files written in python 3.9. The server.py contains the server code and the client.py contains the client code.

Server.py
The provided python server file had wrapped the entire server in a custom Thread class, however this is different to my implementation. The main thread of the server waits in a loop for new connections. When a connection is established, a controller function, called ‘process_connection’, is passed into a Thread object along with the connection socket and runs asynchronously to the main thread, which continues to listen for new connections.

Each connection is assigned a separate thread. Within this controller function, the connection socket waits in a loop listening for any messages from the client, and calls any required functions depending on the request. The primary server functions correspond to the commands in the spec with the exception of EDG, which was delegated to the client side. ‘Process_connection’ directly calls ‘authenticate’, ‘ued’, ‘scs’, ‘dte’, ‘aed’, and ‘out’. In addition to these, the auxiliary functions ‘log_active_connection’, ‘remove_device_from_log’, and ‘log_file_update’ are called when needed for log file manipulation. Finally, the helper function ‘get_file_path’ dictates the server directory structure for incoming client files.

Client.py
In this file, the controller is instead located in the main function, which takes care of validating primary user input, including the command line arguments and client commands. The P2PClient‘ class takes care of all client server interactions and client side functionality. This includes the methods ‘authenticate’, ‘edg’, ‘ued’, ‘scs’, ‘dte’, ‘aed’, ‘out’, ‘uvf’ and the additional function ‘get_help’, which prints a list of the commands and their basic description.

The implementation of the UDP server was more convoluted. During the object instantiation, a subprocess is created to run the ‘_udp_listen’ method using python's multiprocessing module. The subprocess will then run on a completely separate core, and allows the ‘out’ method to terminate the listening loop without being blocked by the line:

```
data, senderAddr = self.udpListeningSocket.recvfrom(MAX_SIZE)
```

The ‘_udp_listen’ method running in the subprocess operates similarly to the server.py’s server implementation. The main thread will sit in a loop listening for any connections, and delegate any connection to a new thread running the ‘_udp_recv’ method. This allows a client to receive multiple incoming files sent via UDP at the same time. The same applies to the ‘uvf’ method, which, for every valid input, would create a separate thread on the main process running the ‘_uvf_send’ method that handles the transmission of the miscellaneous file. Again, whilst not specified in the spec, this allows a single client to send multiple files concurrently. This added functionality does run into behaviour that is not thread safe when transmitting multiple instances of the same file to the same device concurrently, and will cause data to be corrupted. This has a simple fix which can be solved by adding some overhead that rejects an UVF call if the selected file (filename) is already being sent to the selected user.

The UVF implementation also uses some rudimentary congestion control via the use of a stop and wait protocol. The sender will send a data segment, and will wait for an ack from the receiver before transmitting the next segment. As packet loss is essentially a non-factor when dealing with localhost, the aim of the ack is to prevent the sender from rapidly dumping all segments at once, causing some loss. It also has the added benefit ensuring all packets arrive in the same order.

Application Layer Message Format

Since I was lazy, I used Pyhon's Pickle module to encode a python dictionary as the main data structure storing the application layer messages. This is an inferior approach to using something similar to JSON objects, as this effectively ensures only python programs can easily interpret our messages. This is a step even further backwards to something more space and performance efficient like those used for HTTP, which uses bit streams to represent its data.

The most basic messages are the acknowledgements. These contain a single "status" field and are used in response to a request from the sender. In addition to these acknowledgements, all other response messages will include a status code.

```
{
        "Status": XXX
}
```

Status codes:

- 200: general Success
- 400: general Failure
- 401: Request Unauthorised, Client is unknown
- 403: Request Unauthorised, Client is known
- 404: Found no Results
- 418: Request Blocked, client terminates

Response messages can also contain a "result" field. This can range from the result from a computation (SCS), or the active edge device information requested with AED.

Requests must contain the "cmd" field, which differs based on the type of request. This application supports the following commands: "EDG", "UED", "SCS", "DTE", "AED", "OUT", "UVF", "HELP". As part of a quality of life feature, these commands are not case sensitive on the client side as all commands are converted to uppercase letters.

The general request format is:

```
{
        "cmd": command,
        "devicename": devicename,
        "Auxiliary_args": …
}
```

The auxiliary arguments can range from file id's, file names, computation type, file size, etc based upon the command.

For purely data centric messages, a preceding header message is sent containing

```
{
        "cmd": command,
        "devicename": devicename,
        "file_name": file_name,
        "file_size": file_size (bytes)
        "Segments": # of segments
}
```

The UED command also includes the file_id. Following this, the receiver will listen for "segments" number of segments, which will contain purely data bytes, and write those bytes to a new file named "file_name".

System Operation

The Non-CSE functionality is pretty basic. The server and at least one instance of a client is run. Before any interaction occurs, the server processes the input, clears all previous logs, reads from the credentials.txt file, creates the server socket, and begins the listening loop. The client then initiates a TCP connection to the server (which then creates a separate thread to process it) and begins authentication. Once authenticated, the client's UDP port begins listening and the client enters the input loop.

During the loop, the sender issues a request to a receiver (either the server or other peers), and the receiver responds with the requested content. If it's a file transfer, the sender would send the header and the subsequent data segments. All specified operations were implemented in the final client-server program.

Upon termination, the client terminates the subprocess listening for UDP connections, joins all threads sending files, joins all threads receiving files and closes the connection.

Improvements and Extensions

// improvements: split class into a client class and a subclass for udp_client class.
Unlike the server file, my client program is encapsulated in a custom 'P2PClient' class. The reasoning for this was, in early stages of development I planned on creating two separate classes based on the split functionality of the CSE client and the Non CSE client. The base class would be the Client class, which would encapsulate all the Non CSE client functions, whilst the 'P2PClient' subclass would inherit the base class and add upon it the P2P file sharing functionality. In the end, due to time restrictions, this was never fully realised, and the two classes merged into one larger 'P2PClient' class. This can be easily implemented with some refactoring, resulting in cleaner, more portable and better designed code.

Another improvement could be rethinking the application layer message format. In its current state, pickling a python dictionary is both inefficient in terms of processing (extra work), portability (python dictionary instead of JSON object), and data size (vs bitstreams, requires more bandwidth to transmit). With more thought, a better format could be chosen.

A final alteration would be my p2p transmission protocol, where it currently utilises an inefficient stop and wait procedure to artificially throttle the throughput, without any retransmission functionality as it's assuming the client server to be run on localhost. A similar system to what TCP uses, pipelining, etc could be used to improve its efficiency and reliability over wider networks.

Quirks

There were some assumed functionality that isn't explicitly clarified in the spec nor the forums. For example, when sending files to either the server or other peers, any existing file with the same file name in the receiving end is simply overwritten and the upload-log.txt just logs the user sending an updated file.

Referenced Code

Took the log message format from an online post: square brackets enclosing uppercase words describing the nature of the event. Looked too nice to pass up.

```
[EXCEEDED AUTHENTICATION ATTEMPT LIMIT] ('127.0.0.1', 52531).
[CONNECTION CLOSED] ('127.0.0.1', 52531) disconnected.
[NEW CONNECTION] ('127.0.0.1', 52532) connected.
[AUTHENTICATION REQUEST] ('127.0.0.1', 52532)
```