# COMP3331 Report - Python 3.9.2

z5257541 - Marcus Oates

Term 2 - 2022

## 1 Program Design

### 1.1 Server

The server parses the given port number and attempt limit arguments, killing execution in the case of an illegal argument. The server wipes all log files and populates the `invalidLogins` counter and `allUsernames` array. It then creates and binds a TCP socket to the host IP address and the given port number. The server then listens for new connections, starting a new `ClientThread` for each new connection.

In a `ClientThread` the server attempts to log the user in. If unsuccessful then the clients session is terminated, else the server initiates the function `ClientThread.body`, which sends a prompt to the client, and when a command is received it is routed to its respective function, unless it is an invalid or empty command, in which case it is rejected. This process is looped until the client logs out.

### 1.2 Client

The client creates a TCP socket and connects to the server based on the program arguments, and it also creates a UDP socket that is then handled by the `AudienceThread`. In the `main` function the client loops while awaiting new data from the server. It splits this data in command payload pairs and acts on them accordingly. If a UDP command is realised, then the client begins to upload data to the audience. The audience, which is another client, also has an AudienceThread running for the duration of execution, this is the case for all clients, which receives and saves the data from the Presenter client.

## 2 Application layer message format

### 2.1 Server

As the scenario we are simulation does not require intensive computation or facilitate many commands, the application layer messages sent from server to client are quite simple. In `server.py` in the `ClientThread` class there is a function `send` that formats the command and payload to be sent to the client. These command and payload are separated by a delimiter, ∼, and since multiple messages may be sent to the client each message is separated by another delimiter, | . For example, to display an error the string `ERROR∼BCM requires a message body|` would be sent to the client.

### 2.2 Client

Once the message string is received at the client it splits the string into each message, and then each message into a command and payload pair, using the aforementioned delimiters. Then the client can act upon each command payload pair.

## 3 Design Choices

### 3.1 Server limitations

While the server stores data based on its local time, storing the clients local timezone alongside their login information would mean that we could send messages that reflect the time of events for the user, no matter their location.

Three additional commands that would improve the streaming experience would be a to list the users in a given room, to delete a message from either a broadcast or separate room building, this may be restricted to the user who sent the message or the room creator, and also to view offline users and the time these users were last active.

### 3.2 UDP limitations

Currently, the UDP file transfer must occur while no other actions are taking place for the audience, which was a given assumption. However, a server implementation that permits user action from the audience while receiving the file data would be a more accurate representation of a streaming service.

Also, an audience user may not receive more than one file at once from UDP, which is also a given assumption. However, instead of saving all incoming data until an end-of-file signal is received, we could instead split received data and save it based on the sender, until an end-of-file is received for each incoming UDP stream. An even greater improvement would be to split incoming data from the same Presenter based on some protocol prefix, that identifies what file the data should be stored with, in this way we could achieve multiple files being received at the Audience client from different Presenters and multiple files from the same Presenter.

## 4 Borrowed Code

There are three small segments of code that I have appropriated from online sources. The first is in `server.py` lines 198-205, the try-except format was taken from linuxpip.org. The second is in `client.py` lines 15, 16 and 92, the colour of printed text implementation was taken from joeld. Finally, the try-except format at lines 30-34 was taken from Elizafox.