COMP3331 Assignment Report

Language used:

Python 3.7

Platform:

CSE environment

Brief description:

Implemented full functionality of all commands and features except UVF (Upload Video File). The only bug in UVF is that even though packets of given size is being sent and received by the presenter and audience device (proven), data is not being written into the audience device's new file (not sure why).

To elaborate on UVF, my program successfully checks if the audience edge device is active using AED. If the audience edge device is not active, the Presenter client will display the appropriate error message. If the audience edge device is active and the file exists in the Presenter client, the Presenter client will then use AED to get the UDP server port number and address of the audience client. The presenter client will then send an initial data message containing the command, as well as the presenter's username and filename and filesize of the file transferred to the audience client. Once the audience client receives this command, the audience client creates a file with the appropriate name and prepares to receive packets from the presenter client. The presenter client will then read bytes from the file, 1024 bytes at a time and send it to the presenter client. The audience client will continuously receive these bytes and append it to the new file it created, whilst continuously checking for the created file's file size. If the created file's file size is the same as the original file size that was sent in the initial command sent by the presenter client containing the original file size (in bytes), the presenter client will then print a message indicating that the data file has been successfully received.

The only thing that is not working in the UVF is that although data packets of 1024 bytes is being sent and received by the presenter and audience device, it is somehow not written into the audience device's new file. I am unsure of why this bug exists as the UDP connection sockets allow the presenter and audience device to receive the initial data message containing the command, presenter's username, filename and filesize of the file transferred perfectly fine, and I checked that data packets of size 1024 are being received on the audience's end. This means that my UDP connection sockets should not be the problem but the problem lies within decoding and writing the binary data into the audience client's file.

Program design:

Server will be run on host with IP 127.0.0.1 (given by TCPServer3.py) and port given (given in command line argument given when server is started). The server will block the user for an amount of 1-5 times (based on command line argument given when server is started). The client UDP server

runs on the same IP as the host server (127.0.0.1) with the port number provided by the command line argument when client is booted up.

I have introduced some global variables and dictionaries to keep track of the number of failed attempts to login from each user, the time blocked of blocked users, number of failed attempts allowed and active edge device sequence number tracking. I also made use of text files to keep track of logs, active edge devices and login usernames and passwords, such as deletion_log.txt, edge_device_log.txt, credentials.txt and upload_log.txt.

Application layer message format:

The message that is sent to both client and servers are originally in a string, which is then encoded into byte form. The message is then sent to the client or server which then decodes the data back into a string. Depending on the commands, the format of the strings are usually different parameters passed in, spaced by a single space, " " or "; ". The format of the strings are different for different commands, and first parameter, which is split by spaces in the string sent indicates the command type, e.g. list_AEDs, login, delete_file, server_compute etc.

For authentication, the client will concatenate the credentials passed in, as well as the user's host and port number into a string, encode it and then send the data to the server. The server will then authenticate the client. The server will then send a message to the client, which indicates the state whether the login was successful, the login has failed where the client can try again, the login has failed and the client is now blocked for 10 seconds or where the client has already been blocked and the block duration has not expired. The client will then print the appropriate error message and carry out the appropriate actions (e.g. log in, exit client etc)

Transport layer Protocol used:

TCP

Design tradeoffs considered and made, potential improvements:

- Too many if else statements, which is used to navigate through different possibilities and different error messages displayed in the client and server. This could affect readability and debugging abilities
- For the server side, I put all the server functionality focused in a single, big function. This could affect scaling and modifications in the future as the code is not as modular due to time constraints. However, for the server side, the different applications and functions of the client are split into subfunctions, each focusing on one command. These subfunctions are then called in the main function, which breaks down the command passed into the client as inputs, which then splits the input string into parameters and passes it into these subfunctions, which then carries out the specified operations.
- Code is harder to read at the cost of the client and server's functions being easier to debug. This is as error messages and status messages are printed frequently throughout the course of a function being carried out so that the client and server users know exactly what is happening when a function is carried out. This is a design trade-off that is worth to make in

my opinion since debugging can be difficult if you don't know which part of the function is not working, especially for larger functions like SCS.