Assignment Report

Program Design:

My program design follows a simple client-server architecture, in which the server is running on an infinite/indefinite loop and can be contacted by clients who intermittently login and logout. When a client logs on, it contacts the server and makes a request. Following this, the server will translate the request and process it accordingly. Then finally, the server will send the requests' response back to the client. In this way, all communication is done via the main server rather than from client to client (p2p).

In terms of the p2p file transfer, my program design utilises a simple p2p architecture, wherein, as a user logs on, it opens a UDP socket so as to act as its own server (audience client) and can also send requests so long as other users are also active (presenter client). This means that as more users log on, there are more peers to be able to communicate with, and it also means the "server" is intermittently connected, since it is only active when clients are logged on and running their own UDP "server".

Application Layer Message Format:

For the client sending command requests, the message format is a string, as attained from an fget()s call. For server sending command replies to a client, the message format is also a string that is prefixed with the actual command reply and appended with the command prompt string, so that only one response has to be transferred for each command request.

For UDP p2p file transfer, I built a simple reliability protocol within the application layer. For the presenter client, it first sends a packet of data containing the sequence number of the next packet of data it is about to send. Then this is followed up with the actual data packet which is simply an array of bytes sent over in smaller buffered chunks. On the audience side, the client will receive the sequence number packet and compare that with the sequence number it is expecting. If the correct sequence number is received, the audience client will send an acknowledgment packet containing the next expected sequence number.

How the System Works:

Server:
My server program works by first initialising and creating a TCP server socket for all clients to contact. The server will wait until it receives a client contact request, in which case it will create a separate thread to authenticate the specific client. This is so the server can still respond to other client contact requests whilst also responding to authentication and command requests from the connecting/connected client. If a client can authenticate itself, the client's thread on the server side will continue to receive command requests until the user logs out, in which case that client thread will finish. If a client cannot authenticate itself, either

by inputting an invalid username/password more than the maximum allowed failed login attempts, the user is blocked and their current running client thread is closed. If the user failed at the password authentication step, any client who tries to log on using that same username will be unable to log on during their blocked time (10 seconds from failed login). The server run indefinitely until the program is terminated.

Client:
My client program works by also first initialising and creating a TCP client socket to contact the server. It will then be prompted to input a username and password as per request from the server. If the client gets blocked during this authentication step, the client program will exit. Upon successfully authenticating and logging in as an active user, the client program then creates two threads. The first thread, client_input_thread, is responsible for all client command inputs. This is done so that during the command input phase, a client can receive and display server responses (such as messages sent from other clients) without having to input a particular command, otherwise the client program would be blocked at a recv() function call and unable to print to the client's terminal any replies from the server or other clients concurrently. The second thread deals with the UDP client port to be used for p2p video transfers. This thread opens up a listening client UDP socket to wait for any incoming p2p file transfer requests from another client. This is done because the p2p component of the task is not associated with TCP or the server side itself, so creating a UDP socket upon a p2p request would not be viable since the audience client would not know when they had to open up their socket. This client program will continue until the client inputs the logout command, in which all threads will finish, the client will close its server socket and the program will exit.

Segments of Code Borrowed from Web:

As detailed in the client.c code file, lines 94-97 were adapted from the site https://www.scaler.com/topics/display-hostname-and-ip-address/. The code is used to obtain the clients IP address.

Tradeoffs Considered:

In client.c, one tradeoff that had to be made was the graceful cancellation of the client_audience_thread. Since the p2p video transfer functionality, specifically the server side, is run indefinitely until the client closes its connection with the server, the UDP server socket is blocked waiting for a request from another client. This means that when a client logs off, we have to issue pthread_cancel() on the respective thread. This is not good practice, however, since no cleanup is made within the thread itself before it is forcefully terminated.

Improvements:

Following from the client_audience_thread tradeoff, one improvement that could be made is invoking a proper cleanup function that closes the socket associated with the thread and frees any memory prior to cancelling the thread. Additionally, I could place the recv() statement in a

loop that, whilst it does not receive anything, it waits for the client_active variable to equal LOGOUT, in which case it can conclude the thread gracefully, and then in the input_commands thread, we can wait for the thread to close using pthread_join().

Another possible improvement that could be made is implementing a timeout for idle clients to improve network congestion and lower server load. This could be addressed by setting up a timeout on recv() operations, specifically the recv() on client side waiting for server responses, since if a client hasn't made a request to the server in a certain amount of time, the client would not receive any replies. Therefore, after the timeout, the client server will close the thread of the idle client, and the client program will close.