Ali Ali - z5207397
Describing the program design and a brief description of how the program works

Server
The program is designed in a series of while loops. It starts with setting a connecting/accepting socket which allows for input connections. Each connection that is made is then set as a separate thread which goes on to execute the handle_client function. This function is the entire program for dealing with the client and the interaction. The first step is authentication of the client, locking of data structures is implemented using Thread.Condition(), just as the sample example given to us. It reads the credentials file, figures out if the password is within the file or not, and authenticates/creates a new user based on the client's request. The authentication process is all under a single while loop, meaning you cannot progress until you have created a new user or validly logged in with a past username or password.

After this is moved forward into the handle_client_commands section of the program. Within this, it is another while loop which constantly receives messages. If the message received is one checking if the server is alive, it instantly responds with a message saying that it is alive, otherwise, if it's a valid command from a command prompt, it handles those commands. There is also error checking to see if somehow the socket dropped due to a socket abruptly closing, even though we assume graceful closing of the program. Within this function, there are statements to figure out which command was issued, then moving to that commands function, let's use EDT as an example. If EDT is issued by the client it moves to the EDT function. Within this function it runs 5 different checks to ensure that the command was sent with the correct information and number of arguments, after confirming this, it executes. It then, upon completing the intended command, sends a message back to the client confirming that the command has successfully been issued and printing a statement within the server to show the server that it has completed a certain task. It then waits for that socket to send another command through which it will then deal with. This continues for all types of commands.

Areas to note include the SHT command, within this command the THREAD issues the os._exit() which closes the entire program, safely releasing all resources, and then the burden of figuring out the socket is closed is upon the CLIENT who is constantly sending heartbeat messages to see if the server is alive. Another area to note is the UDP and DWN commands, since they are dealing with large file transfers, some of which can exceed the buffer size of recv() within either the client or server size, a HEADER is sent containing just the file size to figure out when the packet has been completely received and to move on with the application rather than getting corrupted information or extra information not relevant to the file and hanging the entire program. Additionally, get_req is a function implemented which checks every byte until it reaches a newline byte, this may seem inefficient to read every byte and adds a new step at every byte that comes in but it is relevant and will be discussed in the application layer protocol. The code is formatted in a way to save information to files and read from files to get information. This is inefficient and spoken about below.

Client

The client is much simpler, it connects to the server and authenticates the user, the assumption is that once runs the client, it has locked the entire server, meaning until it authenticates all data structures are safe to be freely accessed without interference meaning that a server cannot shutdown during this process.It then forks out into two threads after authentication. One thread is constantly sending messages to the server to check if it is still alive, this is for when another client shuts down the server, the client by itself realises the server is down and exits the program. The other thread is waiting for an input to send to the server so the client can interact with the server. Both processes join back together once an input has been received. It then sends the input to the server and then receives whatever the server sends, parses it, executes it if possible, and sends a confirmation message back on the action taken.

In the case of RDT and LST, the client has to evaluate what has been given into a correct data structure to print, essentially making two copies, one at the server and one at the client making it UNSCALABLE use of space especially as the server scales, a way to combat this is to have the server just keep sending information and the user stops once it has reached the end, maybe through a header of the number of threads or messages needed to be received. The consideration here was that it is significantly easier for the server to send all the info as one message to be parsed by the client, but upon consideration that the client may not have as many resources as the server, this should be reversed.

Additionally, with certain commands, extra is asked for the client such as looping to receive information, this should be abstracted and not hardcoded for every command which can lead to scalability issues. But with only 1 command being affected by this, the tradeoff for this current iteration is negligible. The client does extremely little, which is the purposeful design, it sends messages and the server does everything, only the client receives confirmation messages which is how a client-server relationship should be.

The application layer message format

The application layer message is formatted so that only specific keywords are being sent to the server, that being a command in a specific format. The client is the main interpreter of the application layer, it reads every byte till it reaches a new line, realising that the information it received is finished and to stop receiving as that is the "packet" or the "segment" that it must receive at that point to deal with the information. An example of this is

```
connectionSocket.send( "Thread does not exist\n".encode('utf-8'))
```

The syntax and semantics are all hardcoded strings that receive a particular message from a client and immediately respond with another hardcoded string that tells the client it has completed the command (basically an ACK). This makes it extremely UNSCALABLE as it reads every byte for a new line and has to add a UNIQUE string to send as a message to the client any time a new command, new error, or new constraint is added. A solution to this is to implement headers on all data transfers OR organizing the data in a way in which it can be easily parsed by the program without needing additional strings. An example is using HTTP status codes within each message and then printing the relevant message required. Only the result is considered, but it is awful for scalability, performance, and design.

Also, discuss any design trade-offs considered and made.

The biggest tradeoff is to consider the amount of processing to be done at the server and how much to be done at the client-side. With the assumption that the server should handle most of the load as it will have the higher processing power and bandwidth than a client, the maximum amount of processing and parsing of information is done at the server-side. This includes error checking, all data structure storage, and executing commands. Another tradeoff is the hardcoding of strings as confirmation messages rather than using HTTP, this is because this program is limited in scope and does not need such a robust application protocol as it will not fully utilize all aspects and each interaction required can be hardcoded. This is AWFUL for scalability and performance but a tradeoff made for the time constraint for the assignment. Furthermore, get_req is only used for certain commands in the server as testing did not show issues for recv(random amount) but it really should be implemented for every single command to make it consistent. But this is impossible as the input being sent to the server could be anything and there would be a limited way to find out when a packet for the command is recv. Finally, Authentication locks all resources, so if a user is hung on authentication it can make it so resources cannot be fully used till the user authenticates. This can be handled by a timeout for the logging in user or only locking of certain authentication resources rather than forum resources as they are completely different. Additionally can be handled by properly segmenting within the while loop to lock rather than locking everything during authentication.

Describe possible improvements and extensions to your program and indicate how you could realise them.

The client-side is extremely messy, as only certain commands have prints and others need to do other actions before printing a confirmation message, and standardisation of printing the response from the server should be undertaken. This can be done by abstracting the client as done in the server which will lead to less design smells in the future. Additionally, within the application layer protocol, the necessity of checking if a response is equal to multiple different types of the string will slow it down and the HTTP alternative is discussed above. With each rendition of the file due to removal, messaging, and editing, the entire file is completely rewritten, this is an extremely resource-heavy task especially if the server and forums threads scale as it will take longer and longer to utilise commands. A way to resolve this is to only edit out a particular line by reading the file and having header or information within the first few spaces rather than rewriting the entire file. Additionally, I should have used ACKS and/or NACKS rather than reading till I encountered a new line symbol in the byte stream. I do have a response for every single comment which is an ACK/NACK but it isn't designed as such, just a coincidence. Implementing that properly in my program would be beneficial. Furthermore, all information is stored within files that need to be read entirely to figure out certain attributes like highest msg number, this is not performance friendly as threads expand. Extensions could include implementing a database rather than holding all information in files and lists, allowing for better data management. Can be done using python database extensions like psycopg2. Additionally, a front-end implementation to make the input of commands to be easier such as buttons on websites. Can be done using HTML/CSS/Javascript. This all allows for extra scalability and performance.