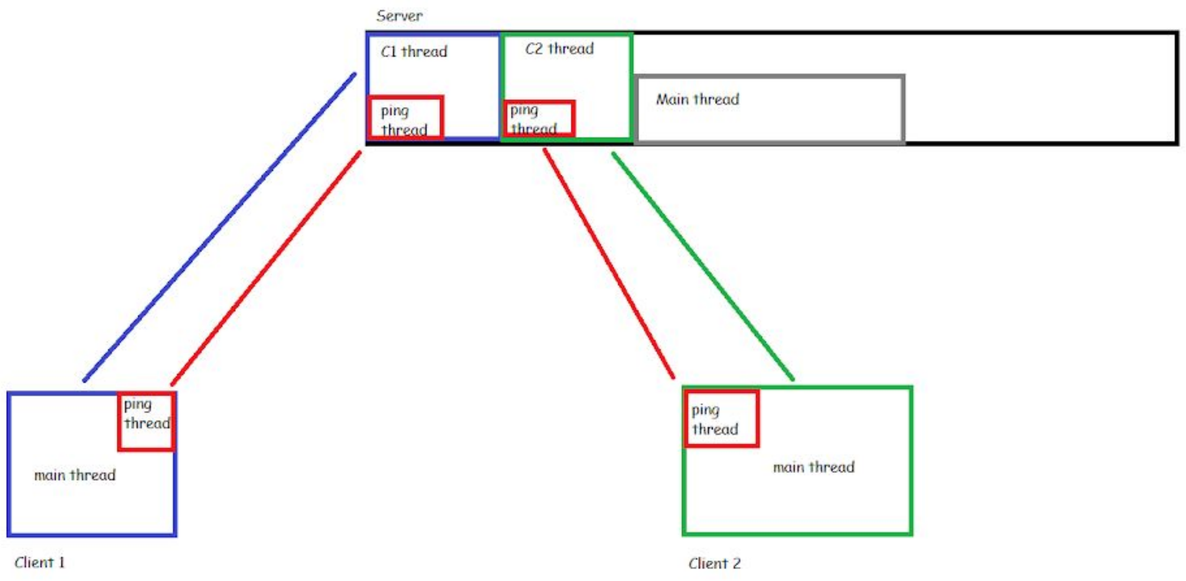
Z5257800, Frank Merriman

How to run

My source files (client.py and server.py) are written in python 3.7.3 on the CSE environment.
To run the server, place the file in an empty directory containing a file called “credentials.txt” and run the command: `python3 server.py port password`
To run the client, place the file in an empty directory, ensure the server is running and run the command: `python3 client.py 127.0.0.1 port`

Program design

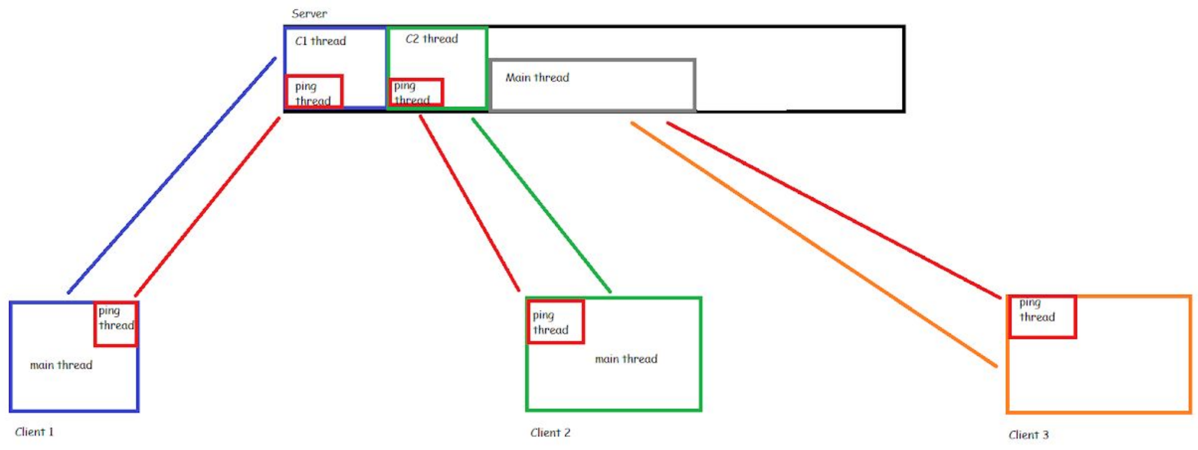
My submission implements the multi-threaded design, meaning multiple clients can concurrently connect to, and send commands to, the server.

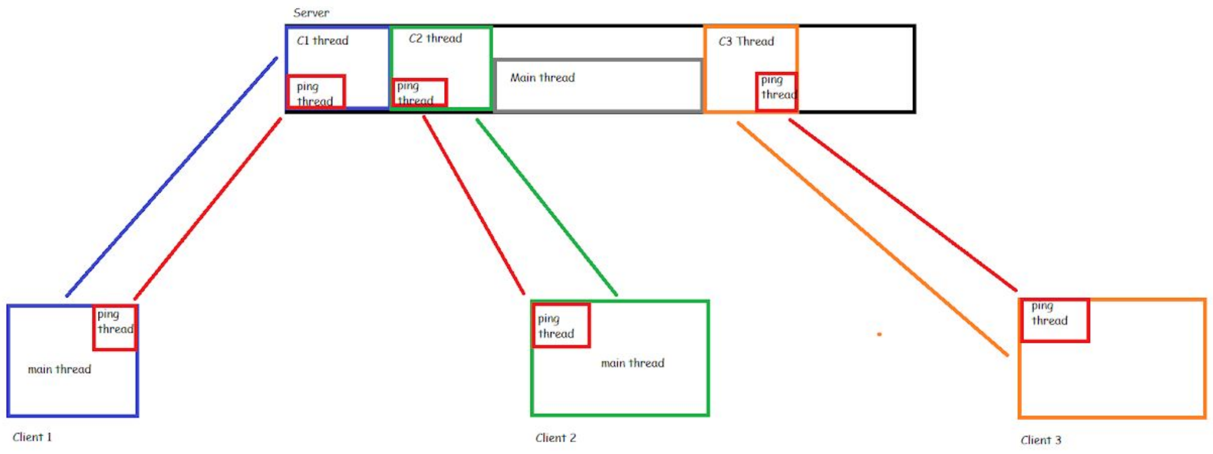


This diagram visualises what the server looks like when handling multiple clients, however it is important for me to note the server only has one socket open, whilst each client has two.

Please note that the positioning of the threads on the server image end are not an accurate representation of which threads are created by/encompassed by which, the main thread creates all sub-threads

Each time a client connects to the server socket, a new thread is created to handle that new client socket (and an extra ping thread for the clients ping socket). The ping threads exist so that on server shutdown, all clients are aware that the server is offline instantly, as there is no longer data being sent from the server to the clients ping socket. I have illustrated this in the following two diagrams:





The main thread is initially waiting for a new client socket to connect. On receiving this new connection, the server accepts the connection and creates a new thread for each socket, sending the sockets as arguments to their respective server threads. After the threads are created by the main thread, the main thread loops and waits for a new connection.

The code executed by each client thread is identical. It first performs the login stage as specified in the spec, a lock being acquired and then released when login is complete (as to keep the read/write of credentials.txt thread safe). It then enters a loop waiting for the client to send data which contains a command and all relevant information to execute it. Likewise a lock is acquired after receiving this data, and then is released when the command is completed. The loop will then start again, with the server thread waiting for the client to send more command data.

Application layer message format

I used a combination of strings and dictionaries to exchange data between the client and server, with the latter being sent in json form, both being encoded and decoded using UTF-8. Encoded strings were used for sending single words or phrases between sockets, mostly in regards to the username/password sign up and server status responses to commands e.g “Your message was successfully added to the thread” or “That thread does not exist”. All commands were sent as dictionaries dumped to json, which were then reconstructed on the server end. Dictionaries proved a useful way to send multiple arguments to the server in an easy to access and (humanly) read way. To access the username of the person who issued the command you just need to use `cmd_dict[“username”]`and so on.

Files were uploaded and downloaded on both sides in 1024 byte chunks, the sender first telling the receiver the size of the file, and then looping to send these chunks, while the receiver would loop to receive data until the sum of these bytes was reached. A similar approach was used in the case of RDT (being the other command we could make no assumptions about file size). The server would tell the client how many lines it should expect to receive, and then loop through the lines sending them as strings, while the client receives and prints lines until it reaches the specified count.

Trade-offs and improvements

A lot of things made difficult in the program stemmed from the specs insistence on using plain text files for storing thread information. This meant that although sending the thread messages was easy, there was a tradeoff in manipulating this data in the MSG, EDT and DLT commands.

The fact that user messages had a line count, but upload messages did not was complicated by plain text. All line numbers were thus not a 1:1 reflection of file lines, but were instead just plain text that needed to be parsed from a string, rather than an actual integer. This caused extra dictionaries and lists to be introduced in the program that tracked these variables as their expected type, so that it was possible to perform the necessary calculations to generate new, accurate line numbers.

There are many edge cases that Salil allowed us to assume our programs would not be tested on. Many revolve around people using numbers as their names to trick the code. It is impossible to simply determine the difference between a line starting with a line number and a line starting with a username (the first being a message, the latter being an upload message), if people have names that are just integers, e.g. username = “5”.

These cases would be tedious to fix by code and would be more easily fixed by changing the data from plain text to json files (or any similar technology). This is the improvement I believe could be made (if the spec had allowed it). For example the number name case would be instantly addressed since you would now be accessing message[“username”] or message[“number”], rather than strings you have gotten from breaking up one long string.