Computer Networks – Assignment Report – z5117408

USING PYTHON3

Program design and how it works

Server: The server basically consists of the main thread and then one extra thread per client. The main thread is responsible for reading in command line arguments, opening the credentials file and starting and accepting connections on the welcoming socket. The main thread also populates Client information structures (username and password fields only) for each client in the credentials file. This structure is seen below;

```
class Client():

    def __init__(self, username, password):
        self.username = username
        self.password = password
        self.clientSocket = None
        self.peerIP = None
        self.peerPort = None
        self.isBlocked = False
        self.blockTime = None
        self.isOnline = False
        self.triesLeft = 3
        self.loginTime = None
        self.logoutTime = None
        self.timer = None
        self.offlineMessages = [] # stores the messages as strings
        self.blockedClients = [] # stores username strings, not client structs
```

Upon client connection a new thread is created, and the client socket is passed to the function that this thread runs (new_client_handler) so that it talks only to that client.

The client handler then demands authentication, which I decided to keep distinct from the general servicing loop because this enforces the authentication process to happen (and happen now). It uses some of the keywords described in the application layer message format section to let the client know about their authentication status. The Client structure is used heavily here, for example, for keeping track of the number of tries left, marking login times, setting the inactivity timer thread, online status and peering information.

Once authentication succeeds, the thread simply goes into an infinite service loop. The inactivity timer is reset here because getting a command here means the client is active. The request is then passed to the request handler class (a singleton), which extracts the keyword and arguments, then calls the correct function based on this keyword. The client handler thread then returns to the servicing loop and this behaviour repeats.

Client: The client program has the main thread and two extra threads. The main thread, like in the server, basically does setup (handles command line arguments and setting up the server and peering sockets). The main thread also calls a login function which is separate from the generic service handlers for the same reason as in the server. Upon authentication, the two extra threads are created, one being a thread for terminal input, the other for receiving responses on the server

socket. The main thread then goes into an infinite loop which is used to accept peers on the peering welcome socket. The client uses a client information structure to store information about the client themselves and any peers. As with the server, the sockets respective to the connection are passed to the thread handlers so that they only communicate on those sockets. The client uses two different but similar methods for handling requests/responses. The client uses a dictionary that maps keywords to functions for sending requests to the server or other peers but uses a response handler singleton class which works in the same way as the server's for handling responses (from both peers and the server).

Application layer message format

My application layer message format is very simple and basically consists of a keyword and then a variable amount of variable length data lines, each separated by a newline. For example, a message request could look like this;

MESSAGE\n
yoda\n
this is my message to yoda

The server (and client when parsing responses) then only needs to split on newlines, use the first line as a keyword to decide which function to call, and then call that function passing in the remaining lines as arguments.

Design trade-offs

- I decided to go with the very simple application layer format described above with the benefits of being enough to implement the required functionality easily but at the cost of being unscalable (the protocol cannot deal with being split over two sends/receives).

Possible improvements/extensions (and how they could be done)

Refactoring: the code could obviously be refactored to be more concise but this is my first python assignment so please excuse the slight code reuse and non-sequential control flow (not sure how to fix that in python as it runs top to bottom)
More robust: I could have made the system more robust with respect to clients logging out or ending abruptly in some other manner (e.g. via ctrl-c or socket time outs/error). These only happen very rarely though and the spec promises behaving clients so it wasn't worth the implementation time to me.

Circumstances that break my system

I haven't found any situations during my testing so far that consistently break my system, however, sometimes a socket appears to timeout or become non-responsive and this can cause the whole system to break down. This is because it runs under the assumption that clients only ever disconnect by calling logout (as per spec). This also means that interrupts like ctrl-c can break the system.

Credits

command pattern credit: https://stackoverflow.com/questions/42227477/call-a-function-from-a-stored-string-in-python

also credit to the sample programs from the early labs (TCP[Server/Client].py and the assignment page as I used those to get the basics of my solution up and running.