Gordon Xie z5160618 COMP3331 Assignment Report

Application Layer Protocol Format

My application protocol format was a simple json message which was encoded and decoded to be sent through the transport layer. Helper functions jsonify and dejsonify were made to simplify this process and avoid code repeat. There is just one key component to each message, which is the 'operation' key. Every message must contain this key as it describes the purpose of the message, what's to be expected in the rest of the payload, and how to respond to it. As such, each message is simply in the following format:

```
{
'Operation':[operation name],
'Variable1':[variable1],
'Variable2':[variable2]...
}
```

By first identifying the name of the operation, the program can know which variables to expect in the rest of the payload so they can access them and use them to perform the operation. All acknowledgement messages work in the same way.

Program Design

Client

The first configuration of my client was just a perpetual while loop which handled both sending and receiving messages. It would also handle input from the user. This was very messy, but because it was under the assumption of that it would be the only client connected, it worked.

The second configuration of my client needed to implement multithreading to implement the multi-client shutdown functionality. That is, it needed to be designed so that if one client shutdown the server, all other connected clients would also shutdown/exit. In order to do this, I created two separate threads to run each client, one which handled all listening and one which handled all sending. The listening and sending threads would communicate with each other through the use of global variables where necessary (such as for multi-step features e.g. logging in, upload, download). This way they can tell each other whether or not the client has used the operation correctly, if it can be performed by the server, and if the client has already logged in. The implementation of a listening thread was crucial since the client had to be permanently listening to the server for a shutdown message in order for the multi-client shutdown to work. Setting the sending thread as Daemon also helped implement this feature, because that ensured that when the sending thread wanted to shut down, the listening thread could do the same by throwing an exception which was handled by exiting the thread. On the other hand, when the listening thread received a shutdown message from the server, it would exit, causing the sending thread to do the same as it became the only running daemon thread.

I had to put in small sleep timers throughout both threads to ensure that messages were printed at appropriate times. This could lead to inconsistencies in performance.

Server

The first configuration of my server was implemented through the use of a perpetual while loop which just kept listening for messages from the connected client. It would break down the json message and immediately take the ‘operation’ value so that it could identify what the client wanted to do. It would then process it accordingly on the server side, using whatever additional variables the client provided in the payload (depending on the operation). The loop could be broken through XIT or SHT.

In the second configuration, my server implemented multithreading quite easily by using a loop which created a separate thread for each client that connected to the server. Each thread would instantiate a ClientSocket to communicate with the client and perform all the operations of the first configuration. Basic locking was implemented for shared resources such as files, but it was done on a very broad level. Essentially, any functions which needed shared resources was wrapped in a lock so that other threads could not access it. This is quite flawed though, as I will elaborate on later in this report.

The server would also keep a list of all the created clientsockets so that when one of them sent a shutdown message, the server could send a shutdown message to *all* the clients, allowing them all to shutdown at the same time. The sever itself would then be shutdown using os._exit()

Trade-offs/Improvements

Sleeps
Due to the way I designed my system with extensive use of multithreading so that packets are constantly being sent/received, sleeps were used to ensure that messages were printed at appropriate times. This is because either listening/sending threads could print at any time depending on the operation it was handling at the time. However, the use of sleep often leads to inconsistencies in performance when it comes to multi-threaded programs. This could probably have been avoided through better planning and design.

Locking
My implementation of locking was minimal due to time constraints. I simply locked functions which used shared resources on the server side. However, this can lead to delays in processing for other threads, particularly when a particular function may take a long time to complete, meaning that the thread handling it holds the lock so that other threads cannot operate at the same time. This could probably be improved by designing the system so that only resources are locked, not entire functions.

Edge cases

There were certain edge cases I didn't account for because the spec or forums said they would not be tested. Notable ones were:

- I created a new credentials.txt file to store login details if the client connected and there was no credentials.txt already in the server. The specs said that we can assume the file will always be there, but it also said that SHT would delete them, so I had this create the file for ease of use
- Didn't specifically handle when multiple users try to log on simultaneously since the was suggested in the specs/forum that this was an atomic process. In my experience, it should work anyway but I didn't do any extensive testing to account for it.
- Didn't do any error handling or messages for when the client tried to download a file they already had or upload a file that was already in a specified thread, it would just overwrite any existing files with the same name. This edge case was not mentioned in the spec or forums.
- I hard coded the localhost IP into the server since a tutor told me that it would be the only one used. When running the client, 127.0.0.1 must also always be used for the server ip in the command line.
- Also did not account for cases where user would suddenly exit using ctrl+c. This usually caused unexpected errors which had to be resolved by using a different port or terminal.

Circumstances where it may not work

Sometimes I get random errors when connecting the client to the server, always fixed by either using a new port or restarting the terminal. I did this with really bad internet though, so that may have been the issue.

Also did not account for cases where user would suddenly exit using ctrl+c. This usually caused unexpected errors which had to be resolved by using a different port or terminal.

Unfortunately, all the code was written by me and no snippets were taken from other sources.