Hoya Lee

z5226463

COMP3331: Computer Networks and Applications

Assignment Report

November 2020

Program Design

Program Structure

- Client directory
    - client.py *client entry file*
    - Any other files that are necessary for a client to upload
- Server directory
    - server.py *server* entry file
    - credentials.txt file for user authentication

Program Design

The program has a **multithreaded structure**. The server has a one to many relationship with its connected clients as it serves these clients in different threads. The only way to remove these relationships is to disconnect the client by issuing XIT or SHT commands.

Application Layer Message Format

Requests or Commands are communicated in string format throughout the application. All of the commands implemented are sent in the format *command username arguments* where username is the username of the user logged in. Moreover, error messages are sent back to the client in a string format as well.

How the system works

Terminal 1: *Server*

```
cd server
python3 server.py 3000 shutdown_password
```

Where *3000* is a port you want to run your server on and *shutdown_password* is the password the server requires for SHT command

Terminal 2: Client

```
cd client
python3 client.py 127.0.0.1 3000
```

Where *127.0.0.1* is the address of the server and *3000* is the port of the server. There can be **more than one** client if desired.

Once the server starts, it will start to listen to every new connection made and require each connection to authenticate themselves. Then, it will listen for any requests made by the connected and authenticated clients to process their requests.

On client start, it will attempt to establish a TCP connection to the server and will send different requests to process its desired commands.

Design Trade-offs

One of the trade-offs is between multiprocessing and multithreading. Multithreading was chosen because it is easier to manage communications between multiple threads.

Another trade-off is the format of messages. Strings were chosen other than other complex data types such as JSON because it is small (each characters are 1 byte) and lightweight, which is the best fit for an assignment of small scale.

Possible Improvements and Extensions

Currently, the client side is not refactored and deemed messy. The client side code can be improved by making it more object oriented and further abstraction of different functions. Moreover, another improvement that can be made is implementing locks. Although race conditions in the context of this assignment are negligible (as all processes/requests are atomic) there can be multiple race conditions when used in real life. For example, a *client x* might download a large chunk of file and *client y* might delete that file while the other client is downloading the file. If locks were implemented, these kind of race conditions would be mitigated. Furthermore, the application message format can be improved as well. Currently, the application uses strings purely as it was deemed to be the simplest form to communicate. However, as different features would be added, it would make more sense to use other structures like JSON format to communicate more effectively and efficiently.

Limitations

As mentioned in the section above, the program would not work with race conditions as it was implemented assuming that the requests were atomic. Moreover, the application would not work in different operating systems other than UNIX based systems as it uses *sys.stdin*, which is considered as a file in UNIX.