COMPSYS 302

# Python Project: Login Server Based Network

Report written by Lite Kim

June 2018

Department of Electrical and Computer Engineering

University of Auckland

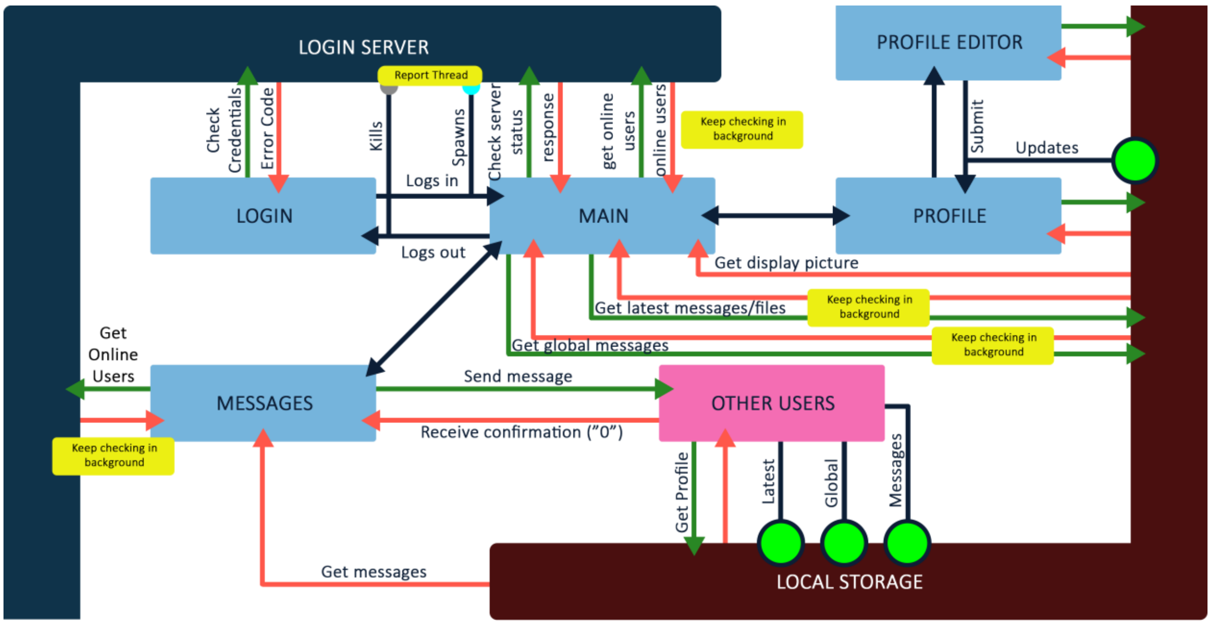
## Requirements

The client had concerns over the privacy of their confidential information being monitored by rival companies. Thus, they have requested a private peer-to-peer networking system to allow for secured communications within the executive team. The client has also mentioned that the system should run on a Linux based operating system.

### The proposal

In response, we have developed a login server-based network system that allows only for people recognised by the server to be able to communicate. With this system, the client does not need to have any more privacy concerns as the server does not store any confidential information, only a list of users and who is online. To gain access to this information, the accessor must also provide credentials that are recognised by the system. If the credentials are not there, then they may not even see the list of users. Each executive member will be supplied with login credentials and a server manager will add or delete entries of credentials manually.

## The System Architecture



In reference to the image above:

- Light blue box: Page
- Blue arrow: Page linking (navigation)
- Green arrow: Request
- Red arrow: Response
- Green circle: Updates information in local storage
- Yellow box: Background process

From the login page, when the user inputs the username and password and hits submit, the page performs the signin function which first calls upon authorizeUserLogin. The function checks if the input parameters can retrieve a list of users which requires a valid username and password. If successful, the user must have valid credentials and should be signed in. Before redirecting to a signed in page, signin runs initReport which reports to the server (through serverReport) that the user is signed in and spawns a thread that runs the function backgroundReport which will automatically sign the user in every 60 seconds. (For the purposes of testing, the user is signed in every 10 seconds).

After sign in, the user is redirected to a populated index page. The index page runs four functions in the background along with the auto reporter. The first is checkStatus which checks if the server is online or not. getLatest and getBoard refreshes the latest messages and message board sections every 5 seconds. Finally getonlineusers is run in the background and refreshes every 5 seconds. This function call the /getList API of the server which retrieves a list of online users which is displayed on the bottom right corner of the screen (expandable object upon hover). These sections update and retrieve the latest information without having to refresh the whole page.

The user can navigate to their personal profile by clicking on their display picture. This runs the /getProfile client-client API but is called upon them self. The /getProfile for this system first looks for a text file named under the username in the /server/profile directory. The text is opened then split into the relevant profile sections. This is put into a JSON format and is returned to them self. With the retrieved information, the system cleans all inputs of any HTML/JS injection then is put into the relevant HTML div sections.

On this page, the user can update their own profile by clicking on the Edit Profile hyperlink. The user is redirected to a new page with an empty/pre-filled form. Before the user edits, the prefilled information was retrieved first and auto-filled for the convenience of providing the user to make minor changes and not have to write out the form again. When the user hits submit, the profile text file is updated and the user is redirected to their edited profile.

At the messages page, the page takes in a parameter. By default the parameter is towards global messages. The client can change the parameter by clicking on an online user's button. This retrieves the message log sent by the other user and displays it on the page. These messages can be of images, audio, video, and PDF which will be embedded onto the message page through the embeddedObjects function.

From the message page, the user can send messages and files as well. The user needs to type in the valid parameters and the message is sent. Upon clicking send, the page is refreshed and if the message was successfully sent, a new div will appear saying "sent successfully".

Other users may send messages or files to the client using the /receiveMessage and /receiveFile API's respectively. Upon being called, text file found in the /messages directory are created or appended (using the appendFile function). The 0000.txt file is for Global messages which only get updated if the globalmessage parameter is 0. The 1111.txt file is for latest messages which get updated every time anyone sends a message. This file only holds the 10 latest messages. Other files are created and appended onto to sort out the messages by the sender (whereas the 0000.txt and 1111.txt are universal).

Messages are also rate limited. Every time sends a message or file, the limitRate function is called. It maximizes the number of messages sent to the client from that particular user at 7 per minute.

There is another background task called getProBack but was left disabled due to problems with multithreading. The getProBack automatically retrieves profile details from every online user by calling upon their /getProfile API's. This is run continuously and takes a while to complete a cycle as there can be many try/except cases and majority of the time is spent opening, closing and saving to files.

## Issues with Development

### Confirming Functionality

Confirming that the functionality is possible by a single individual given that they develop on multiple systems communicating with each other or through the use of a third-party application (such as Postman), however the results are solely dependent on the sole developer when the results should be dependent on all developers. Each developer has their own development methodologies they follow by which result in different systems.

Through solo development, mistakes can be common as they can be overlooked. Functions may seem to work properly but that is only because the user has developed it to their own problems. Working with a team provides the reduced frequency of mistakes as each member can catch each other's mistakes when communicating.

This issue was overcome by working with peers.

### Synchronizing work

The problem with working with peers is that the progression is dependent on the skills of all users. There is a limit to how far a developer can progress before requiring confirmation checks with a peer. This was overcome by working on the same functions as a group, progressing synchronously.

## Features That Improve Functionality

### Threading:

Provides an always responsive user interface. Takes away the frustration of having to wait for functions to finish before the GUI becomes responsive again.

Refreshing sections using threads instead of the whole page. When the whole page refreshes, the user is waiting a longer time for elements to reload as there are elements that do not even require reloading (large graphics). Reloading the whole page is highly ineffective if there is one small element that requires refreshing.

Not refreshing embedded audio, video, PDF's. The latest messages and global messages board refresh and will refresh all elements within. This does not allow the progression of a video or audio past the refresh length. The embedding of elements have thus been moved to a separate page (/messages).

## Peer-to-Peer Methods and Suitability

The client does not have to worry about information leaking from a remote database. The only way to retrieve information is to hack into an account and request for the information. Not only does the user not need to provide the file but also, the IP is tracked. The hacker can be located. Information cannot be retrieved without permission or from another remote location.

There are other more (unimplemented) method calls that boost the functionality of the system. There are certain parameters such as the profile_username when retrieving profiles as when a user is offline, another user may request that profile from someone else who may have that profile.

The problem with peer-to-peer is that transfers are slow and you may not know where to access certain files. A central server can provide much more information and is more global.

## Python

Python provides CherryPy, a Python Library that takes away the complexities of having to develop a server system from scratch. Python however has very different syntax and for a experienced developer who have adjusted to the more universally similar languages may find it a bit hard to get used to.

## Further Developments

The implementation of a pure peer-to-peer system and to finish the user interface of all pages. The implementation of databases to efficiently store information. The use of hashing and encryption for improved data security. And better methods for threading.