COMPSYS 302 – Python Project Report

Author: Sam Morgan

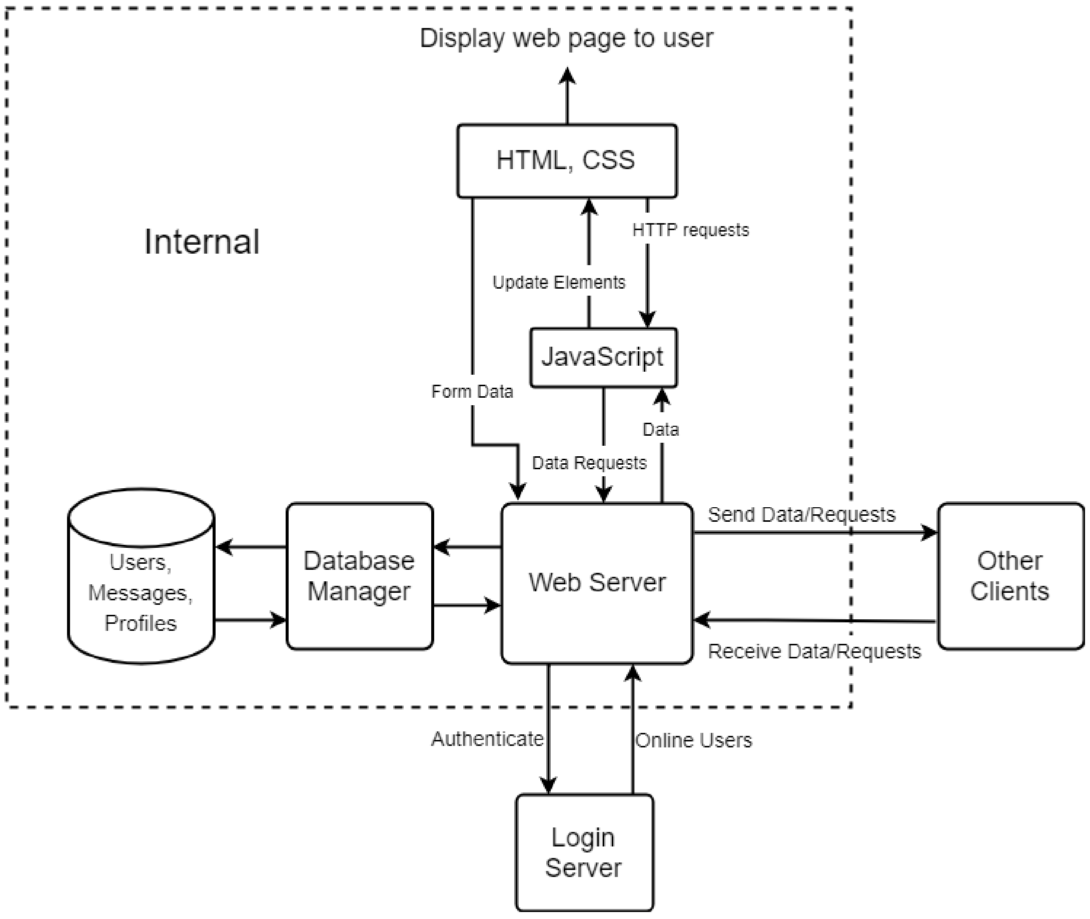
UPI: Smor264, ID: 6397731

Key Application Requirements:

The primary goal for this project was to create a secure peer-to-peer based messaging application. The implemented system achieves this by presenting the user with a login interface to access their account as long as they have the correct credentials (with optional two factor authentication). After a list of online users has been retrieved from the login server all further communication occurs between clients with no message data being stored on the login server. The user is presented with a simple and intuitive interface in which they can send/receive messages and files, view other user's profiles, and update their own. Entered passwords are immediately hashed and stored as such.

Top Level View of System:

The HTML and JavaScript handles the front-end and user input and passed information along to the CherryPy webserver backend. When the user enters their details, their password is hashed and checked by the login server, if the credentials are valid the user is presented to the main page which contains all the necessary interfaces. While at this main page, some JavaScript is periodically refreshing certain page elements such as the currently online list and message log with up-to-date data from the webserver.



1 - Top Level System View

Significant Development Issues:

An issue that was encountered during development was ensuring the user was presented with timely information without heavily utilizing system resources and retaining responsiveness. In order to achieve this careful consideration was given to how often a page should be refreshed. Threading was used to help separate the background task of staying logged in and regularly reporting to server. A refresh rate of around five seconds was chosen as this kept all key elements up-to-date and feeling responsive without blocking other commands and services

Another issue discovered was relaying information between frontend and backend. Multiple solutions were implemented to address this. HTML forms were used to send data such as updating profile information, login information, messages, and files. JavaScript HTTP requests were used to fetch data such as saved messages and online users from the webserver. This allowed key elements to be refreshed with having to reload the entire page.

Key Features:

The implemented system has additional features that improve the functionality of the system and end-user experience such as:

Protection from HTML and SQL injection. Messages (sent and received) and profile data is cleaned of HTML tags (‘<’ and ‘>’) so that if another user tries to send HTML to modify what the recipient sees on their page it will just appear as plain text and will not change the displayed page. This can protect the user from being fed misleading information and visuals. All inputs to the database are sanitized so that other users can’t send data that would delete or modify the database in potentially undesirable/malicious ways.

Two Factor Authentication (via google authenticator app). If the user desires they can enable two factor authentication from the main screen when they login. They are presented with a unique QR code which they can scan using the Google Authenticator App on a smart phone. Next time the user logs in, they will also have to input a passcode that is generated every 30 seconds by the app. This passcode is matched against one generated by the webserver and will allow the user to login if their credentials are valid and the passcodes match. The user can disable this feature once they have successfully logged in by pressing the enable two factor authentication button again. This system gives another barrier to stop unauthorized access to the users account, should their username and password become compromised, making the users account more secure.

Embedded Images, Audio, Video. If the user is sent a file it is saved to their computer. If the file is a support image, audio, or video format it will be embedded in the messages window so that the user can immediately see the file without having to locate and open the file themselves.
Automatically refreshing content. The main webpage will periodically refresh the content of the online user list and the current message chat so that the user will always be presented with up-to-date information removing the guesswork and manual refreshing to know who is online and whether or not they have received new messages.
Threading to regularly report online status. To maintain a responsive user interface reporting online status to the login server is performed on a separate thread so that it does not interfere which other important requests which the main thread handles such as user commands.

Peer-to-peer Suitability:

A pure peer to peer system would have likely been very problematic and time-consuming for this project due to the difficulty of initially finding peers on the network and correcting validating the clients are who they say they are. This could potentially be achieved if each client had a list of every user's username and password, however this creates security consoles. The hybrid peer-to-peer/login server approach removed this problem by making it very easy to discover other users and validate credentials whilst retaining the core peer-to-peer element of only sending messages between clients and not to any third party/middleman such as a central server which would be a very visible target to DDOS (Distributed Denial of Service) attacks, which would render the entire service unusable, or hacking which could compromise every users message history. With the current implementation if the login server went down, currently logged in users would still be able to communicate and potentially with some additional features in the future could be able to perform as login servers themselves, allowing users to login off each other as a backup. This allows the network to continue operating at a slightly less secure level. The current peer-to-peer methods such as sending/receiving messages and profiles fit the client's needs well as it reduces the risk of confidential information being accessed, by spreading out where confidential data is stored.

Protocol Suitability:

The protocol gave clear in-depth details on how clients should interact with each other. This ensured each client could communicate with each other regardless of individual implementation (so long as they abided by the API). It helped give structure to communication between machines which is essential for any type of networking application. Without it, it would be near impossible for clients to reliably communicate different types of information to each other. When compared to the proposed drafts by each team, the official protocol kept the core feature list the client required, as well as extending the potential for encryption.

Tools Used:

Python as a programming language fits the back-end part of this application nicely, as it facilitates rapid development/prototyping and easy debugging as well as support for a wide variety of custom libraries specifically designed for our purpose. A few of these key libraries include CherryPy, a minimalist object orientated web framework which can handle a large amount of the webserver backend which helps speed development time. PyCrypto which provides secure hashing functions and encryption algorithms. Urllib and Urllib2 facilitates easy interaction with URLs. JSON and base64 for fast, simplistic ways to safely encoding data to be sent over the web.

HTML and CSS were the tools used to develop the front-end human-machine interface. These are both tried and true languages for creating web-based interfaces and provide the developer with a variety of options for their design from a very basic functional interface to a beautiful, clean interface. HTML and CSS both have a relatively shallow learning curve which helped to create functional webpages rapidly.

JavaScript was difficult to work with as it is bad at identifying syntax errors and because of this perform in unintended ways without directly notifying the user. However it facilitated communication between the HTML and web server which would have been otherwise impossible.

Future Improvements:

Key improvements to my application in the future would be:
Greater offline support, to ensure the network can still function if the login server went down. With the correct implementation (as stated in the API documentation) each client could act as the login server for validating login credentials and sharing information required to connect online users to each other. This feature set would make the system much more robust and reliable.
Greater level of encryption. In the application's current state, messages are stored in plain text in the database and when they are sent to other users. To increase security in case the user's computer becomes compromised, all entries in the database should be encrypted to prevent an unauthorized user from easily accessing the contents. The messages should also be encrypted when they are sent over the internet. In its current form, all messages send to and from my application are vulnerable to Man-in-the-Middle (MITM) attacks, where an attacker can intercept my messages and read or alter the contents immediately. If endpoint authentication and message encryption and hashing were implemented it could prevent almost all MITM attacks. These security features address key vulnerabilities and would help the application further meet the clients goal of having a secure network.