Client-Server Secure Messaging Platform

Computer Science A Level Non-Exam Assessment

Sam Poirier

30 November 2018

Contents

[1 Analysis 3](#_Toc531363359)

[1.1 Analysis Plan 3](#_Toc531363360)

[1.1.1 Interview 3](#_Toc531363361)

[1.1.2 Questionnaire 3](#_Toc531363362)

[1.1.3 Project Background 3](#_Toc531363363)

[1.1.4 Objectives 3](#_Toc531363364)

[1.1.5 Solutions 3](#_Toc531363365)

[1.2 Interview 4](#_Toc531363366)

[Questions 4](#_Toc531363367)

[Responses 4](#_Toc531363368)

[Analysis 4](#_Toc531363369)

[1.3 Questionnaire 5](#_Toc531363370)

[Questions/Responses 5](#_Toc531363371)

[Analysis 5](#_Toc531363372)

[1.4 Project Background 8](#_Toc531363373)

[1.4.1 Specific Client Needs 8](#_Toc531363374)

[1.4.2 Existing Solutions 9](#_Toc531363375)

[1.5 System Objectives 13](#_Toc531363376)

[1.6 Solutions 14](#_Toc531363377)

[1.6.1 Framework 14](#_Toc531363378)

[1.6.2 GUI Framework 14](#_Toc531363379)

[1.6.3 Integrated Development Environments 15](#_Toc531363380)

[1.6.4 Database Formats 15](#_Toc531363381)

[1.6.5 Solution Decisions 15](#_Toc531363382)

[2 Documented Design 16](#_Toc531363383)

[2.1 Database Structure & SQL Statements 16](#_Toc531363384)

[2.1.1 Entity Relationships and Attributes 16](#_Toc531363385)

[2.1.2 Normalised Data Structures 16](#_Toc531363386)

[2.1.3 Data Dictionaries 16](#_Toc531363387)

[2.1.4 SQL Statements 17](#_Toc531363388)

[2.1.5 Data Flow Diagram 18](#_Toc531363389)

[2.2 System Design 19](#_Toc531363390)

[2.2.1 IPSO Chart 19](#_Toc531363391)

[2.2.2 Top-down diagrams 20](#_Toc531363392)

[2.2.3 Client Flowchart 22](#_Toc531363393)

[2.2.4 Server Flowchart 25](#_Toc531363394)

[2.3 Data Structures & Dictionary 27](#_Toc531363395)

[2.3.1 Circular Queue 27](#_Toc531363396)

[2.3.2 Client Data Dictionary 27](#_Toc531363397)

[2.3.3 Server Data Dictionary 27](#_Toc531363398)

[2.4 Class Design 28](#_Toc531363399)

[2.4.1 Client Classes 28](#_Toc531363400)

[2.4.2 Server Classes 29](#_Toc531363401)

[2.4.3 Generic Classes 29](#_Toc531363402)

[2.5 User Interface Design 30](#_Toc531363403)

[2.6 Algorithms 33](#_Toc531363404)

[2.6.1 Circular Queue 33](#_Toc531363405)

[2.6.2 Hashing Algorithm 34](#_Toc531363406)

[2.6.3 Integer Merge Sort 35](#_Toc531363407)

[2.6.4 String Merge Sort 36](#_Toc531363408)

# Analysis

## Analysis Plan

### Interview

It is important to interview the client, Mr Ovayolu; a secondary school teacher, in order to establish the key features that they require for the program to include, as well as what features are less important not needed. I will work from the responses the client gives in order to accurately tailor the software to their requirements.

### Questionnaire

I will develop a questionnaire to determine the demand from the general public in order to tailor the software to be desirable to users other than the initial client, as long as the general demand does not contradict the client’s needs. I will use google forms to create the questionnaire.

### Project Background

Within the Project Background I will detail the specific Client needs, in order to fully flesh out the purpose of the software. I will also analyse a number of existing pieces of software that fulfil similar solutions to determine crucial aspects that should be included in my project, but also determine flaws which I should make sure not to include in the software

### Objectives

Under the objectives section, I will detail the exact objectives the project must meet, along with the specific criteria of each objective. This will allow me to ensure that the final product meets with all the client expectations.

### Solutions

Here I will detail the technical aspects I plan to use in order to develop the software, including programming languages, Integrated Development Environments, general database rules (if applicable) and other technical Solutions.

## Interview

|  |  |  |
| --- | --- | --- |
| Questions | Responses | Analysis |
| What features of an instant messaging platform are most important for you? | • Fast and secure communication between students by 1:1 or group chat  • Easy to understand, clean but functional user interface  • Be able to run in all Windows versions  • View chat history  • Group chat and direct messaging  • Reply to a particular message, both in a group chat and in a private chat.  • Forward the messages to a particular group chat or an individual.  • Get alert when new messages arrived.  • View list of connected users in the school.  • Report button for inappropriate, bullying, threatening messages… | The software is being designed specifically for this user. It is important that as many of these are implemented as possible, and ideally all of them are. |
| How often will the users use the platform? | Break and lunch time | The software will mostly be used during the day, so there will be time available for potential server maintenance; there isn’t a need for 100% uptime. |
| How many users should the platform support? | Concurrent around 100, total around 2000 | The program must support a large quantity of users – some stress/load testing will be needed to ensure the load can be managed successfully. |
| How much message history should users be able to see? | 3 Months or 100 messages. Ideally, user should be able to set this in settings. | The client wants a configurable number of messages to be readable, meaning the server should likely store all messages then send the client the amount they have requested. |

## Questionnaire

|  |  |
| --- | --- |
| Questions/Responses | Analysis |
|  | Everyone who answered the survey has used an IM application at some point, which means that the feedback received is not necessarily representative of all the users, so should not be treated as an absolute representation of the userbase. |
|  | WhatsApp is the most widely used IM application, followed closely by Discord and Instagram. The least widely used applications are IRC and Slack. All of these extremes should be investigated in more detail in order to replicate their success and avoid their mistakes. |
|  | Interestingly enough, although WhatsApp was the most widely used, Discord was in fact the one that was used the most overall, and snapchat which was thoroughly in the middle of the widely used, was one of the only three chosen as most used. This shows that the most widely used applications are not necessarily the best ones. |
|  | Windows and Android Phone are by far the most widely used, and so for the best coverage of users, are the platforms that would be the best to target. |
|  | Again, Windows is by far the most used platform and is therefore the clear frontrunner in terms of OS to develop for. |
|  | 73.4% of those surveyed indicated that Data Security is important for them. This is clearly an important feature. |
|  | 93.3% of users indicated that secure logins where an important feature for them so this also is a crucial aspect of the project. |
|  | Again, most users agree that having a viewable message history is an important feature of an IM application. |
|  | Only just over half of the responses indicated that they cared about transparency regarding their personal data and what was done with it. |
|  | Most people agreed that an easy to understand/use user interface was a good feature. It could be worth developing a few different layouts to receive feedback about. |
|  | A large number of those surveyed indicated that a customisable UI would be wanted. More detail might be needed into what exact customisations are wanted. |
|  | Most users agreed that Direct Messaging would be an important feature. It was also requested by the Client so is a high priority. |
|  | Group chats were also indicated a feature wanted by the client as well as the survey candidates. This means that it should definitely be implemented in the product. |
|  | A fairly mixed response from the survey, but this was also requested by the client so should be included regardless. |
|  | Some of these feature requests are more in the scope of this project than others. These were individually requested and so are low priority. |

## Project Background

Instant Messaging platforms have existed pretty much as long as computers have, originally (in the 1960s) acting as a way of communicating between users on multi-user operating systems. They were also originally used for notification systems like printing. A good example of possibly the first instance of this software is from CTSS (Compatible Time-Sharing System), an operating system developed by MIT which was the first to demonstrate any sort of messaging, and was a precursor to email.

Hashing has also existed as a concept for a long time, with the term originating from its non-technical meaning (to "chop" or "make a mess" out of something), and was first coined in the 1960s. For this project, we will be specifically looking at Cryptographic Hashing which usually consists of a mathematical algorithm that maps data of arbitrary size to a bit string (a hash) and is designed to be a one-way function. The purpose of this function is to obscure a user’s personal data so that it is secure.

### Specific Client Needs

I will be creating a secure Client-Server messaging platform. My client is Mr Ovayolu, a Computer Science teacher who wishes to set up a secure platform for the teachers and students to discuss work with each other. The focus will be on a secure design, hosted by the client to ensure they know exactly where their messages/data is stored and that it is not being sold to companies by the owners of the software. This is important as a large amount of the data on the system will belong to minors, so the school has extra responsibility to protect them. The end users will be teachers or students who may not be very ‘tech-literate’; the application must be easy to set up and use.

### Existing Solutions

#### Internet Relay Chat

The first IM platform to run over the internet was Internet Relay Chat (IRC), which was made in 1988. By 1989 IRC had spread across the world with over 40 servers worldwide.

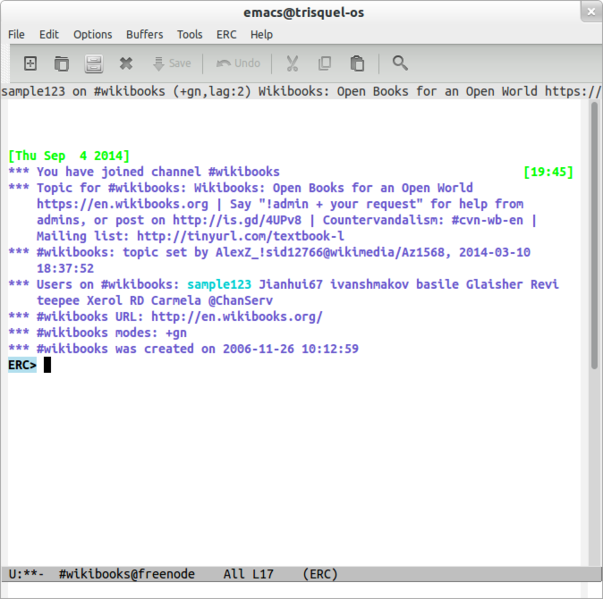


Figure 1.1: An IRC Client called ‘ERC’ which runs within the text editor Emacs.

|  |  |
| --- | --- |
| Pros | Cons |
| Client-server model – the client’s GUI or functionality can be upgraded without touching the server allowing for all sorts of clients to be compatible with each other. | Complicated to set up servers correctly and securely. |
| Can be secure if set up correctly. | Some clients are complicated to use and set up. |
|  | Little support for multimedia formats, reporting messages, file transfer etc. |

#### Discord

Released in May 2015, Discord is a multiplatform Instant Messaging and voice-over-internet Protocol (VoIP) application. It runs on pretty much every platform available and is free to use.

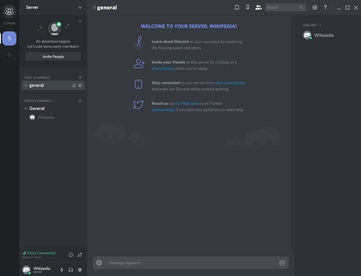


Figure 1.2: The Discord desktop application after creating a new server.

|  |  |
| --- | --- |
| Pros | Cons |
| Extremely easy to create new accounts and servers – all implementation is abstracted away. | No personally hosted servers – all data is controlled and stored by Discord Inc. meaning that a user is not in control of it’s data’s security or status. |
| Servers are secure in the sense that only invited users can join. | Written using Electron – a framework with a famously high memory footprint on the PCs running it, so may not work well on older machines. |
| Lots of support of extra features, for example VOIP, image embeds, direct messages, file transfer. |  |

#### WhatsApp

WhatsApp is a free Instant Messaging, Voice and Video Chat client. It was founded in 2009 and is owned by Facebook. It is supported on many platforms including most smart phones.

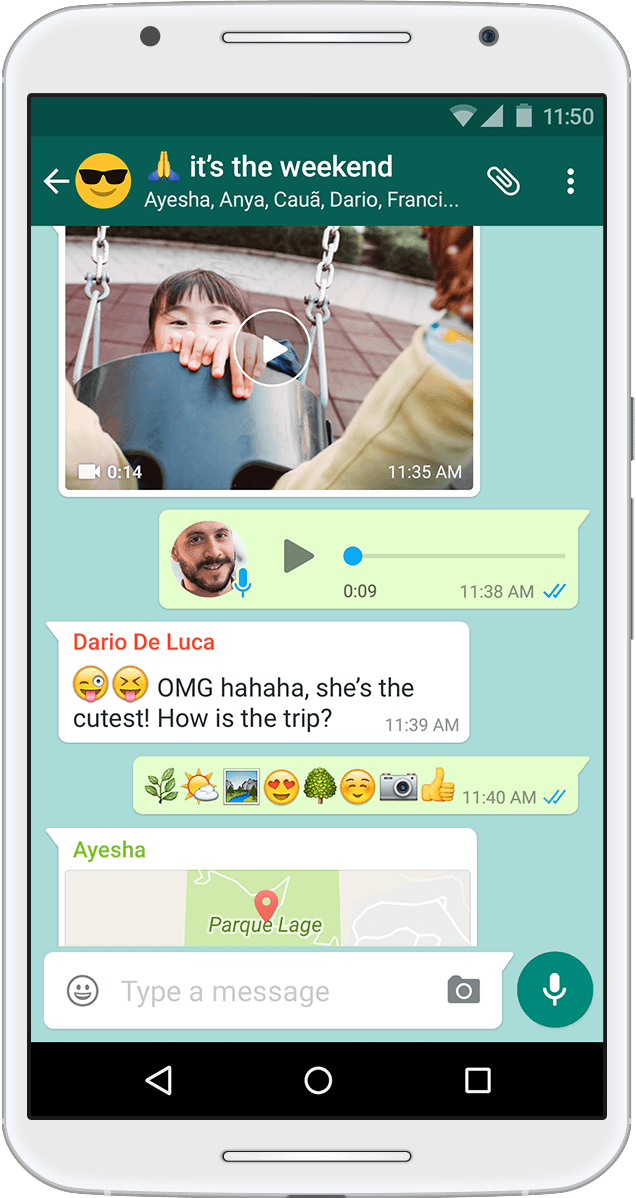


Figure 1.3: WhatsApp running on an Android Device

|  |  |
| --- | --- |
| Pros | Cons |
| Easy to use and set up. | Very little privacy or control over data – it is owned by Facebook, a company which is notorious for abuse of its users’ personal data. |
| Easy to connect to new ‘servers’ or group chats. | User interface is not designed for large groups of people or high rate of messages being sent. |
| Lots of features including voice chat, file sharing, direct messages etc. |  |

#### MD5 Hashing Algorithm

MD5 is one in a series of algorithms designed by Professor Ronald Rivest of MIT in 1991. It was designed as a cryptographic hashing function for use in encryption.

|  |  |
| --- | --- |
| Pros | Cons |
| Implementations exist already in many different programming languages already, in addition to pseudocode for the algorithm being widely available. | Suffers from extensive vulnerabilities, including a high chance of collisions (when two or more different input string produce the same output hash string) rendering it obsolete for encryption usage. |

#### 1.4.2.5 SHA-2 Hashing Algorithm

SHA-2 (Secure Hash Algorithm 2) is a set of cryptographic hash functions designed by the United States National Security Agency (NSA) in 2001. It utilises a Merkle–Damgård structure.

|  |  |
| --- | --- |
| Pros | Cons |
| When implemented correctly is virtually uncrackable. | Low support on older systems, for example Windows XP. This conflicts with the Client’s requests of running on all OSes |

## System Objectives

|  |  |  |
| --- | --- | --- |
| No | Objective | Performance Criteria |
| 1 | Login Screen User Interface must contain:   * Input for username and password * Login Button * Register Button * Error Label to display login issues | All of these UI elements must exist in an organised and easy to read layout. |
| 2 | Register User Screen UI must contain:   * Input for username, password and password confirmation * Register (submit) button * Error label to display register issues | All of these UI elements must exist in an organised and easy to read layout. |
| 3 | Main Window UI must contain:   * Box to contain message history * Input box for sending messages * Button to send message * General server information; who’s connected, who you’re logged in as etc | All of these UI elements must exist in an organised and easy to read layout. |
| 4 | Connecting to the server should be secure. | The initial handshake between client and server should be secure; ie no ‘fake’ clients attempting to connect should be accepted. |
| 5 | Client/Server Communication should be secure. | All packets to and from the server should be secure so that anyone intercepting them cannot read the information. |
| 6 | All messages and personal data should be stored securely. | The database should be encrypted; the whole thing in a two-way encryption unique to the server, and the passwords in a one-way encryption too. |
| 7 | User interface should be easy to use and read. | The UI should be intuitive to read, use and understand. |
| 8 | Client and server should be easy to set up and run. | The client and server should require minimal setup to get into full working/running order. |
| 9 | The server must be able to handle lots of users both consecutively and in general. | The server must be able to support 100 concurrently connected users and 2000 total registered users. |
| 10 | There must be a way for users to configure settings for the client. | The clients should be configurable via a settings menu which is saved between sessions. |
| 11 | The application must support most requested features by the client. | As many features requested by the client as possible should be implemented into the program. |

## Solutions

### Framework

|  |  |  |
| --- | --- | --- |
| Language | Pros | Cons |
| Python | Lightweight  Simple Syntax  Can be developed on school computers  Multiplatform  Decent documentation | Interpreted therefore slow and not memory efficient |
| C# | Powerful  Personally preferred language  Lots of good documentation  Very easy to debug | Low multiplatform support  Cannot be developed on school machines |
| JavaScript (via Electron) | Lightweight  Forgiving syntax  Multiplatform | Extremely loosely typed – easy to make bugs  Hard to debug |

### GUI Framework

|  |  |  |  |
| --- | --- | --- | --- |
| Language | Framework | Pros | Cons |
| Python | tKinter | Shipped stock with python, no extra installation | Extremely limited in terms of functionality  Hard to use |
| PyQt | Support for advanced GUI configuration  Easy to use | C++ Backend, some errors are hard to catch and result in silent crashes |
| C# | Winforms | Easy to use  Stock with C# | Lots of auto-generated code or lots of manual setup |
| JavaScript | Electron (with node.js, jQuery and npm) | Frontend developed in HTML/CSS so the GUI is very customisable. | Very memory inefficient  Uses lots of other people’s modules/code |

### Integrated Development Environments

|  |  |  |
| --- | --- | --- |
| IDE | Pros | Cons |
| Visual Studio | Advanced Debugging tools  Good at identifying programming errors before compiling/running  Easy to use | ‘Heavy’ piece of software – lots of overhead |
| Visual Studio Code | Advanced Debugging tools  Good at identifying programming errors before compiling/running  Easy to use  Lightweight | Less features than Visual Studio |
| IDLE | Comes stock with most python installations  Very lightweight  Simple Debugging tools | Feature lacking; little auto completion or pre-run bug finding |
| Notepad++ | Extremely lightweight | Very few features  Cannot compile many languages |

### Database Formats

|  |  |  |
| --- | --- | --- |
| Format | Pros | Cons |
| SQL | Very powerful  Lots of supported datatypes  Can create databases with complex structures | Complicated to set up  ‘Heavy’ – not very lightweight at all |
| SQLite | Lightweight  Reasonable amount of datatypes  Support for reasonably complicated structures | Less features than SQLite |
| MongoDB | Unique Solution; structure is based upon a tree rather than a table | Feature lacking; little auto completion or pre-run bug finding |

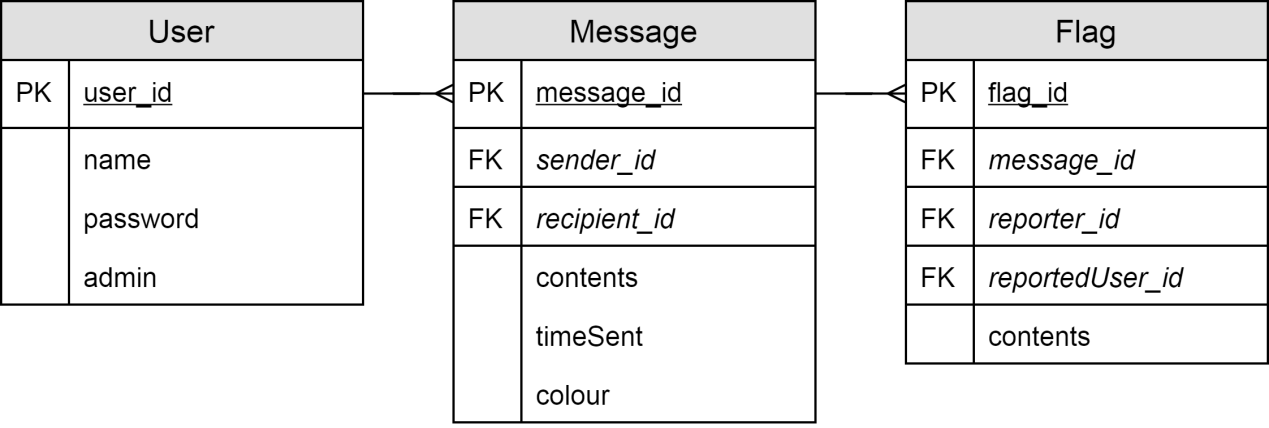
### Solution Decisions

I plan to write my program in Python, as it runs on multiple platforms easily and can be debugged without too much trouble. I will use IDLE to develop the project as it has all the features I need, and is nice and lightweight without too much bloat. I will use PyQt for my GUI as tKinter is not advanced or powerful enough to meet the client needs. I will use SQLite for my database as it will likely be quite simple; SQLite is a good compromise between features and functionality.

# Documented Design

## 2.1 Database Structure & SQL Statements

### 2.1.1 Entity Relationships and Attributes



### 2.1.2 Normalised Data Structures

User(user\_id, name, password, admin)

Message(message\_id, *sender\_id*, *recipient\_id*, contents, timeSent, colour)

Flag(flag\_id, *message\_id*, *reporter\_id*, *reportedUser\_id*, contents)

### 2.1.3 Data Dictionaries

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table Name | User | | | |
| Primary Key | user\_id | | | |
| Foreign Keys | N/A | | | |
| **Data Item** | **Data Type** | **Validation** | **Sample Data** | **Description** |
| name | String | Not null | “user” | The username. |
| password | String | Not null | “O.L&ZXJL3=!VK&\>@,!” | The (one way) hashed password. |
| admin | Boolean | Not null | False | Indicates whether this account has admin privileges. |

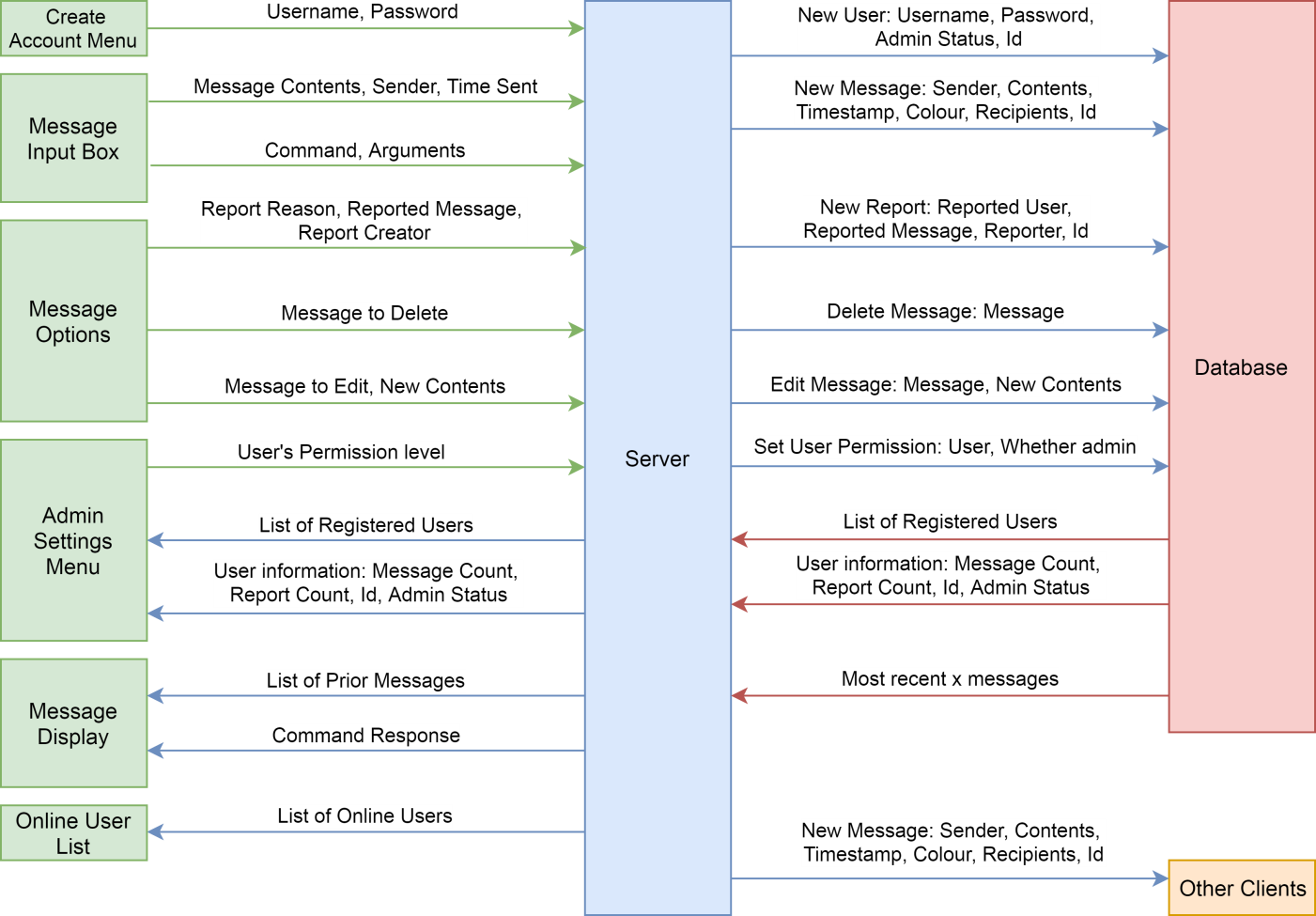
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table Name | Message | | | |
| Primary Key | message\_id | | | |
| Foreign Keys | sender\_id, recipient\_id | | | |
| **Data Item** | **Data Type** | **Validation** | **Sample Data** | **Description** |
| sender\_id | Integer | Not null | 2 | The id of the user who sent the message. |
| recipient\_id | Integer | Not null | 1 | The id of the recipient of the message (1 is the server which means everyone). |
| contents | String | Not null | “Hello World” | The content of the message. |
| timeSent | String | Not null | “26-10-00 11:10” | The date/time the message was sent. |
| colour | String | Not null | “#000000” | The hex colour value to display the message as. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table Name | Flag | | | |
| Primary Key | user\_id | | | |
| Foreign Keys | message\_id, reporter\_id, reportedUser\_id | | | |
| **Data Item** | **Data Type** | **Validation** | **Sample Data** | **Description** |
| message\_id | Integer | Not null | 1 | The id of the message that was reported. |
| reporter\_id | Integer | Not null | 2 | The id of the user who reported the message. |
| reportedUser\_id | Integer | Not null | 3 | The user who’s message was reported. |
| conents | String | Not null | “They insulted me” | The reason why the message was reported . |

### 2.1.4 SQL Statements

|  |  |
| --- | --- |
| Statement | Use |
| SELECT \* FROM User | Extracts all users in order to verify logins. |
| SELECT \* FROM Message limit ? offset (SELECT count(\*) FROM Message)-? | Selects x most recent message from the database where ? is the amount to select. |
| SELECT name FROM User WHERE user\_id == ? | Gets the username of a user from their id. |
| SELECT name FROM User WHERE name == ? | Used to check if a user with a certain name exists, returns null if they do not. |
| SELECT user\_id, name, admin FROM User WHERE name != 'SERVER' | Gets a list of all registered users apart from the ‘SERVER’ user as that is not a true user. |
| SELECT user\_id, admin FROM User WHERE name == ? | Gets a user’s user id and whether they’re an admin from their name. |
| SELECT count(\*) FROM Message WHERE sender\_id == ? | Returns the amount of messages a specific user has sent. |
| SELECT \* FROM Flag WHERE reportedUser\_id == ? | Selects all the reports corresponding to a specific user. |
| SELECT contents FROM Message WHERE message\_id == ? | Gets the contents of a message from its id. |
| INSERT into User(name, password) values (?, ?) | Creates a new non admin user, as ‘admin’ has a default value of False. |
| INSERT into Message(sender\_id, contents, timeSent, recipient\_id, colour) values (?,?,?,?,?) | Creates a new message. |
| SELECT last\_insert\_rowid() | Gets the Id (primary key) of the most recent insert. |
| SELECT sender\_id FROM Message where message\_id == ? | Gets the sender id of a message from the message Id. |
| INSERT into Flag(reportedUser\_id, message\_id, reporter\_id, reportReason) values (?,?,?,?) | Creates a new message report. |
| UPDATE Message SET contents=? WHERE message\_id=? | Changes a message’s contents by Id. |
| UPDATE Message SET contents=?,sender\_id=?,colour=? WHERE message\_id=? | Changes a message’s contents, who sent it and its colour. Used to ‘delete’ messages. |
| UPDATE User SET admin=? WHERE user\_id=? | Sets a users’ privelidges. |

### 2.1.5 Data Flow Diagram

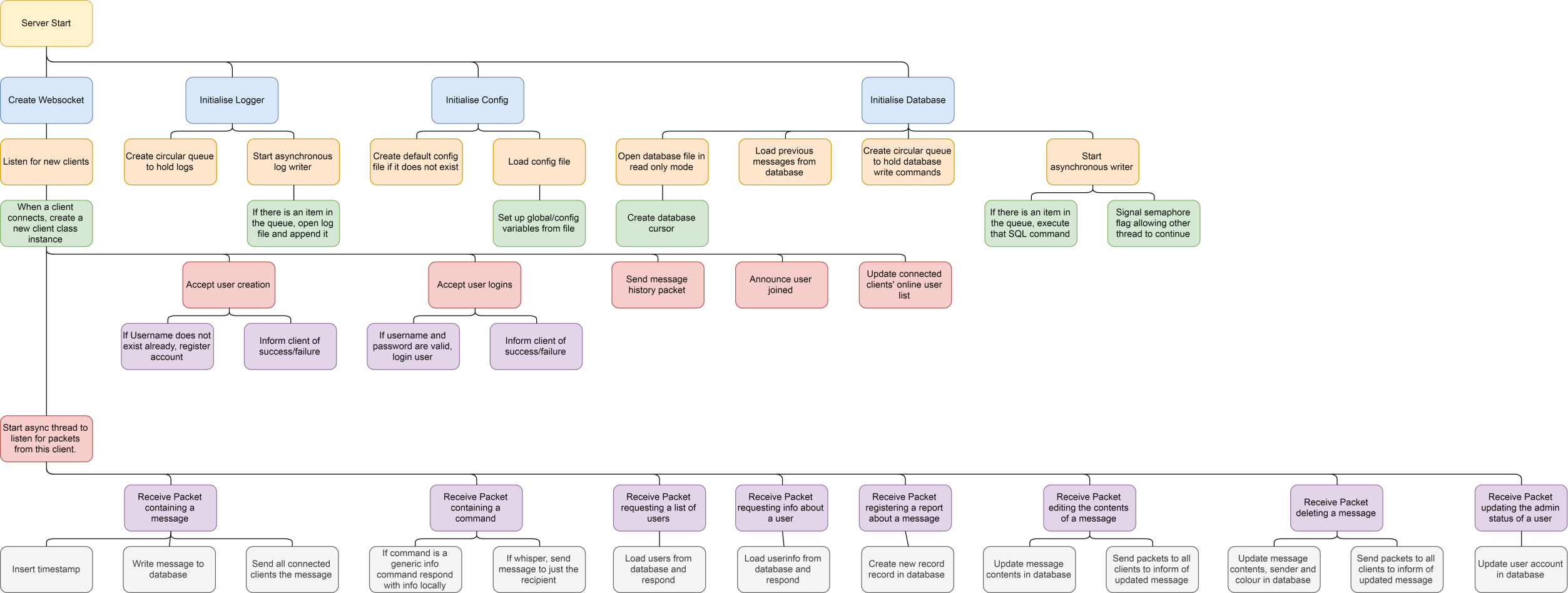


## 2.2 System Design

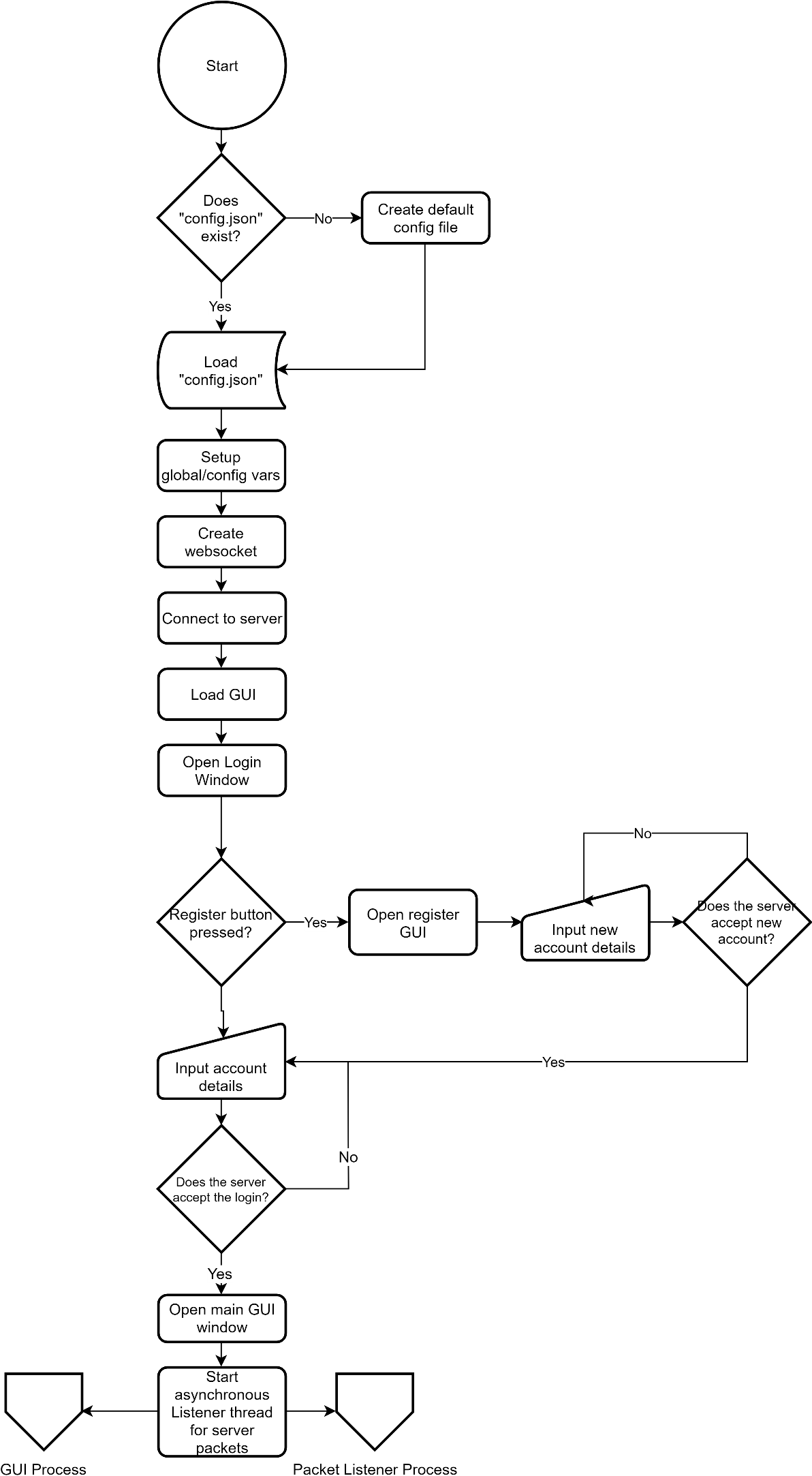
### 2.2.1 IPSO Chart

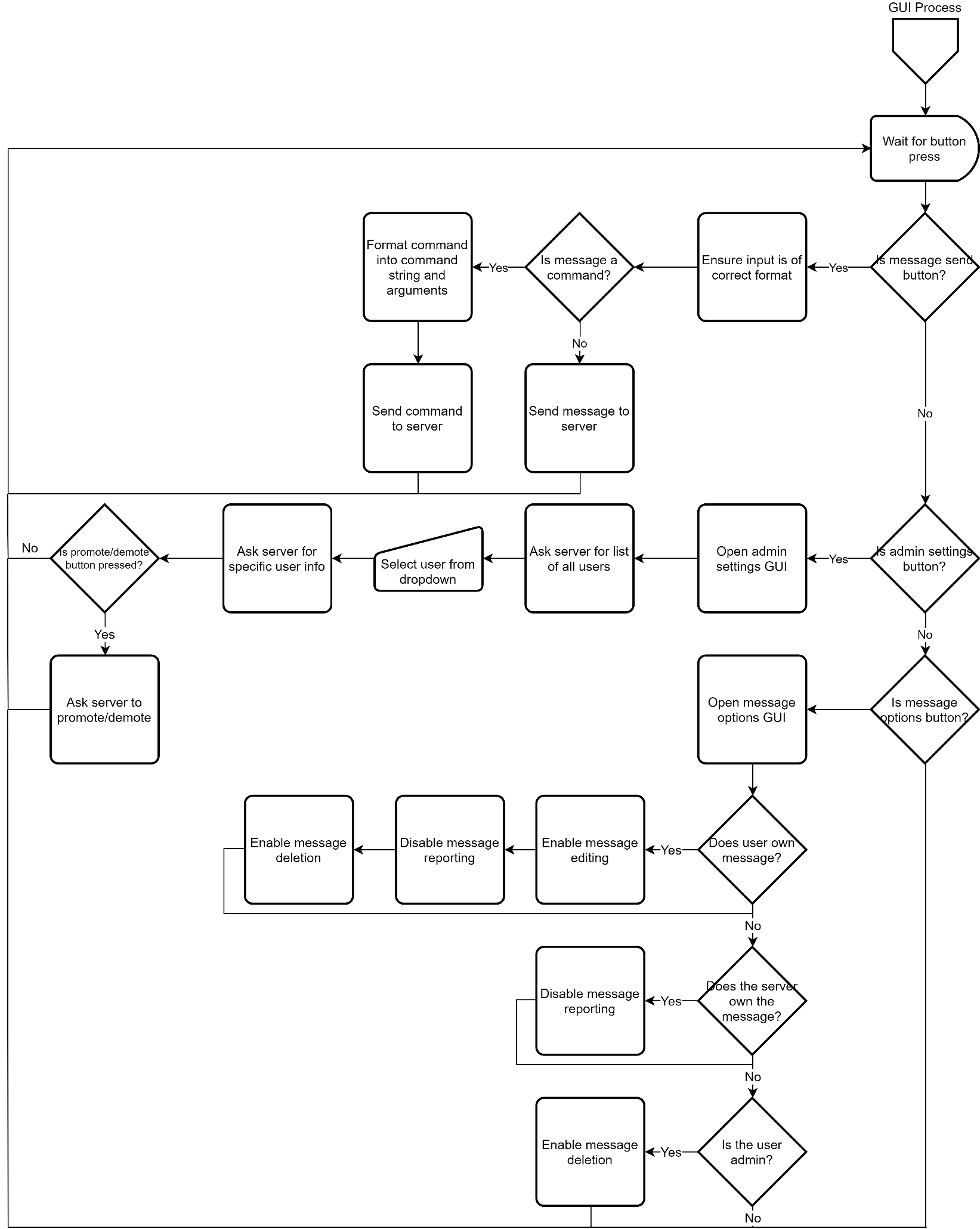
|  |  |  |
| --- | --- | --- |
| IPSO | Program Section | Item |
| Input | Registering User | Username  Password  Confirm Password |
| Logging in | Username  Password |
| Sending Message | Message Contents |
| Reporting User | Report Reason |
| Processing | Registering User | Check if username is available  Check passwords match |
| Logging in User | Check username and passwords match |
| Sending Message | Check if command  Insert username  Insert timestamp  Attach colour  Designate recipients  Send to other clients |
| Process command | Determine if valid command  Generate corresponding response  Return response to correct recipients |
| Reporting User | Attach message id  Attach sender id |
| Storage | Register User | Create new record in User table in database |
| Sending Message | Create new record In Message table in database |
| Reporting User | Create new record in Flag table in database |
| Promoting user to admin | Update user record in User database table |
| Editing message | Edit message record in Message database table |
| Deleting message | Edit message record in Message database table |
| Output | Display Message | Create new message widget  Update text elements |
| Display Online Users | Update online users widget |

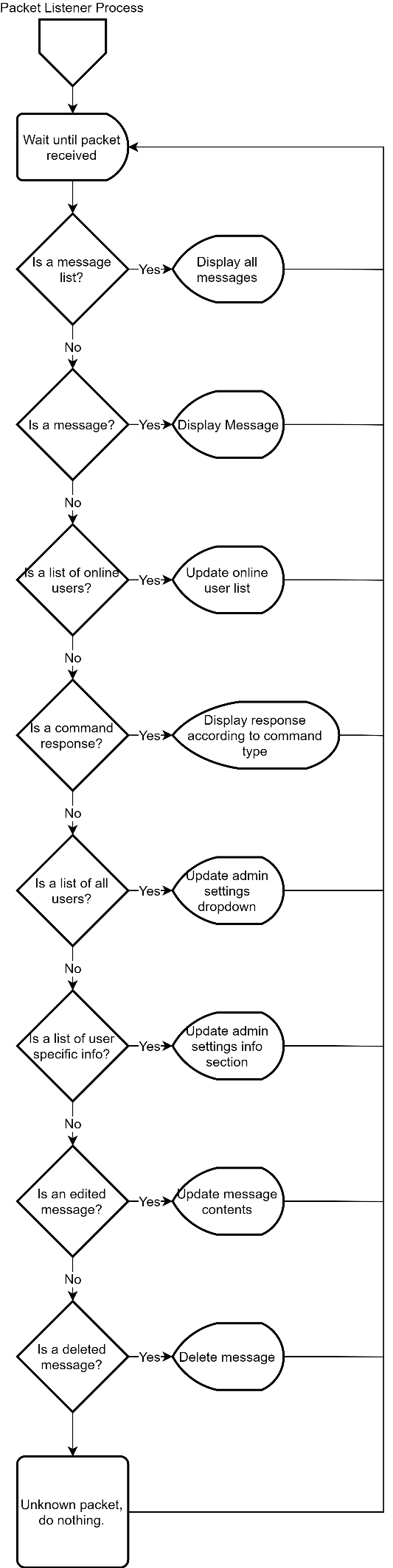
### N:\A2\NEA\Writeup\Diagrams\top-down-client.png2.2.2 Top-down diagrams



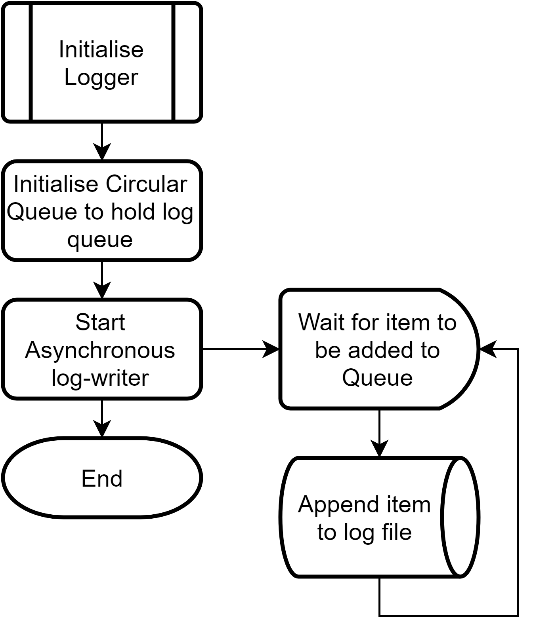
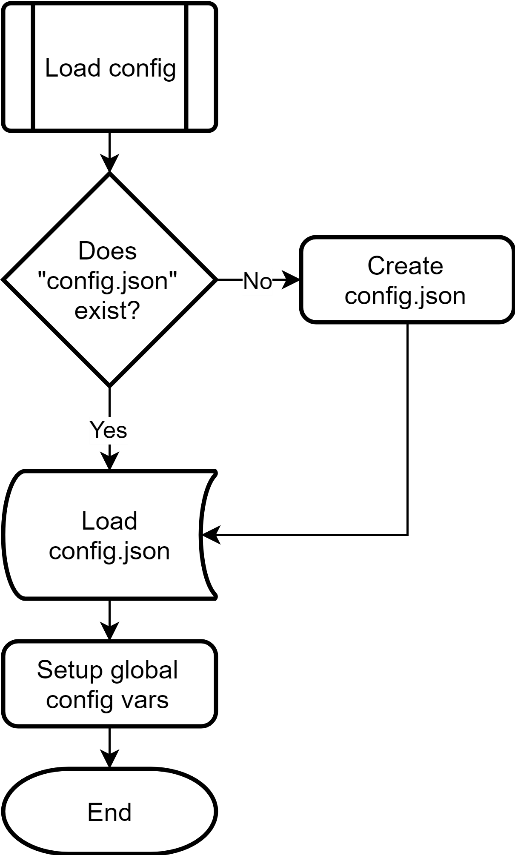
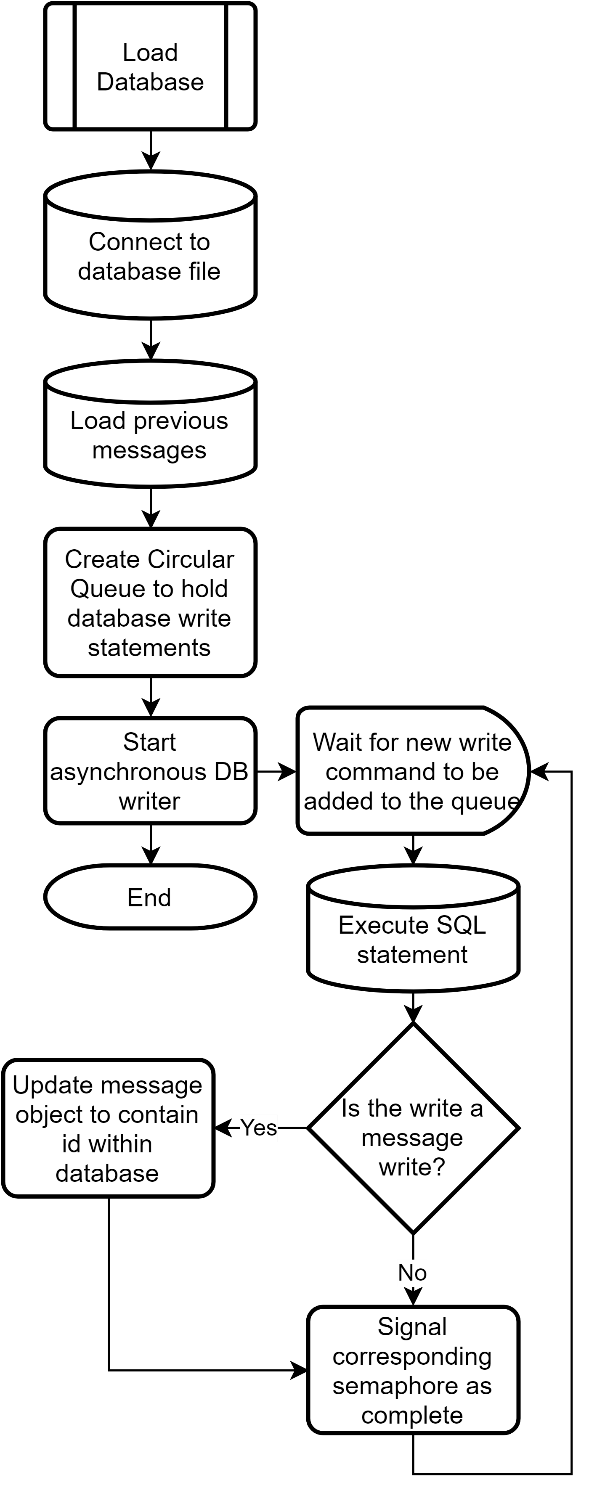
### 2.2.3 Client Flowchart

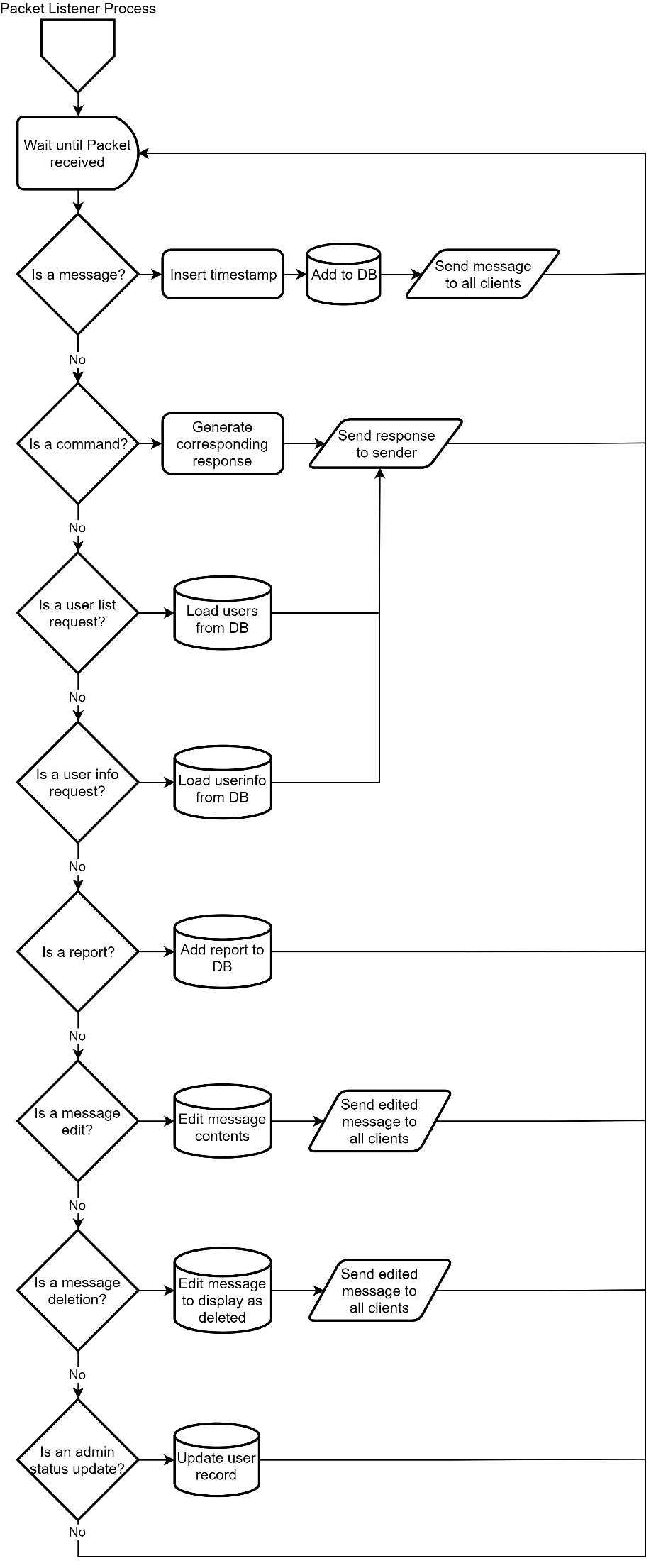






### 2.2.4 Server Flowchart





## 2.3 Data Structures & Dictionary

### 2.3.1 Circular Queue

Both the database writer and logger will operate using a circular queue to ensure there is never an error due to two threads attempting to write at once. The queue will hold each write to be made chronologically, with the oldest item at the front.

Variables will be needed for the FrontPointer, RearPointer and Length in order to keep track of the size. Custom methods to EnQueue and DeQueue will be written rather than accessing the data array directly.

|  |  |  |
| --- | --- | --- |
| Index | Data Example |  |
| 0 | **INSERT into Message ...** |  |
| 1 | **Update User ...** |  |
| 2 |  |  |
| 3 |  |  |
| ⋮ |  |  |
| n |  |  |

### 2.3.2 Client Data Dictionary

|  |  |  |  |
| --- | --- | --- | --- |
| Data Item | Data Type | Validation | Sample Data/Info |
| \_app | QApplication |  | The instance of the PyQy app |
| \_mainGui | MainWindow |  | The main GUI window |
| \_serverSocket | socket |  | The websocket that all server connections will go through |
| \_username | String | Not Null | “admin” |
| \_userId | Integer | Not Null | 1 |
| \_admin | Boolean | Not Null | True |
| NONPRINTINGCHAR | Char (1 length string) | ==“\u200B” | “\u200B” |
| MAXTRANSMISSIONSIZE | Integer | Not Null | 40960 |
| COMMANDCHAR | Char (1 length string) | Not Null | “/” |
| DEBUG | Boolean | Not Null | False |
| New message | String | Not Null | “Hello World” |
| New Username | String | Not Null | “Sam” |
| New Password | String | Not Null | “secret” |
| Confirm Password | String | == New Password | “secret” |
| Username Login | String | Not Null | “User” |
| Password Login | String | Not Null | “shhhhh” |
| Report User | String | Not Null | “Said a rude word” |
| Edit Message | String | Not Null | “Hello World!” |

### 2.3.3 Server Data Dictionary

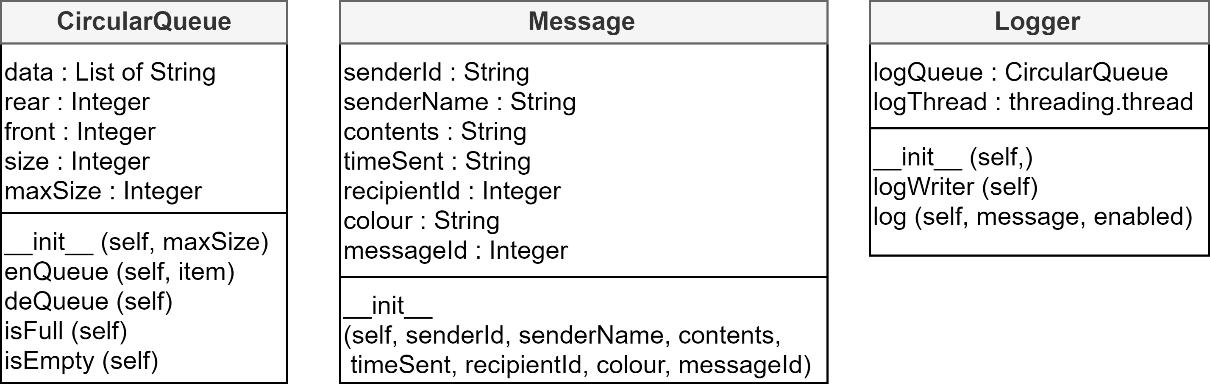
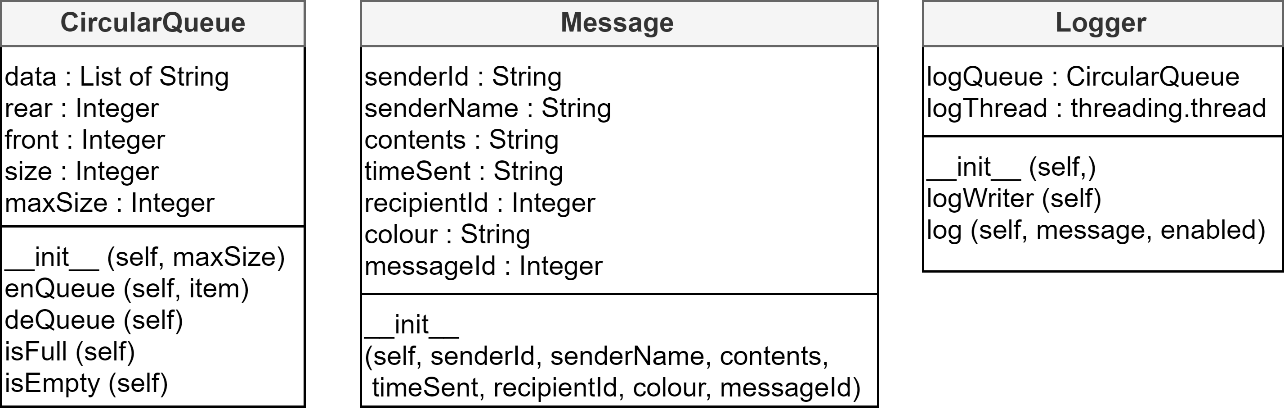
|  |  |  |  |
| --- | --- | --- | --- |
| Data Item | Data Type | Validation | Sample Data/Info |
| \_clients | List of Client | Not Null | A list containing all connected clients |
| \_database | Database | Not Null | Database management class |
| \_logger | Logger | Not Null | Logger Class |
| \_configManager | ConfigManager | Not Null | Config Manager Class |
| INFOLOGGINGENABLED | Boolean | Not Null | False |
| MAXTRANSMISSIONSIZE | Integer | Not Null | 40960 |

## 2.4 Class Design

### 2.4.1 Client Classes

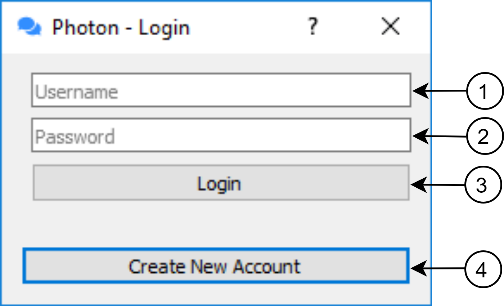
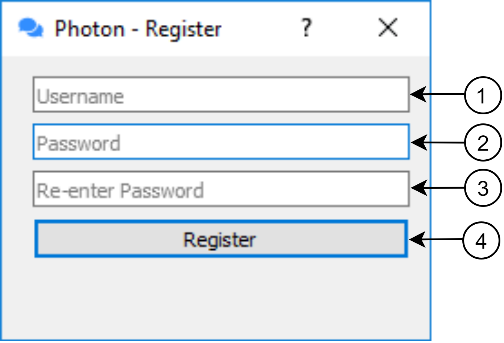
### 2.4.2 Server Classes

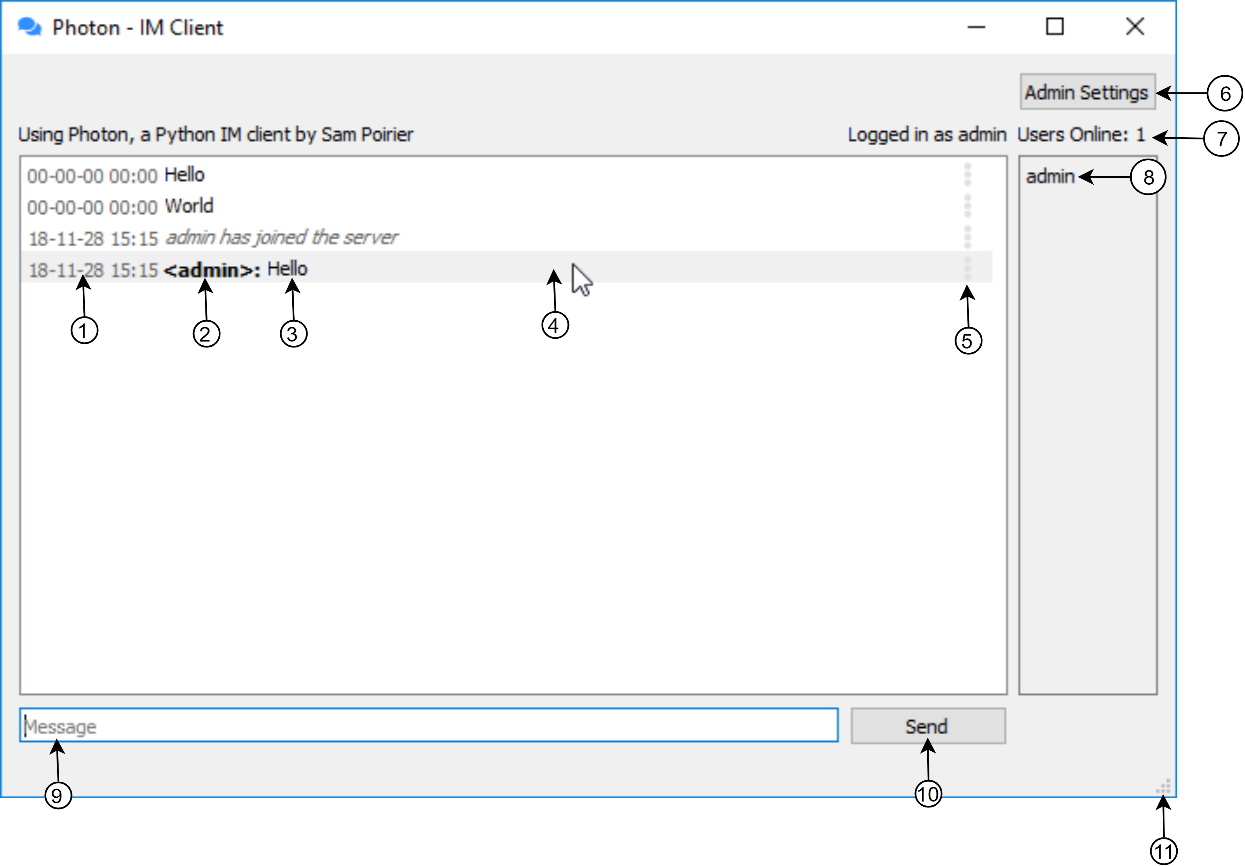
### 2.4.3 Generic Classes



## 2.5 User Interface Design

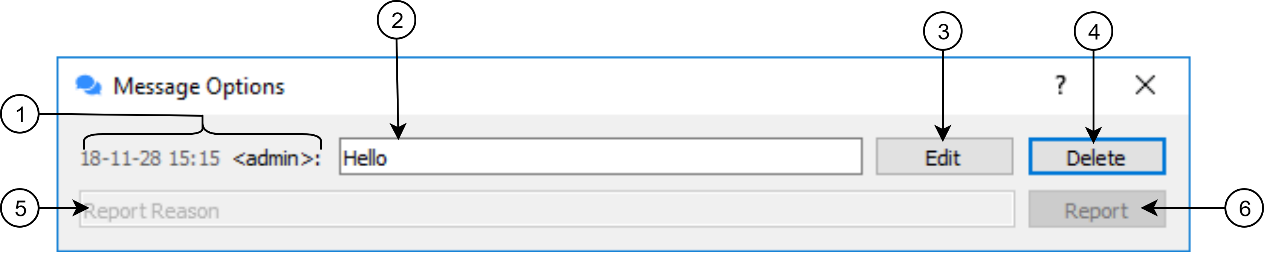
1. Input for the username of the account to login to.
2. Input for the password of the account. Displays text as ‘•’ characters.
3. Login button sends login request to server, taking the input from the text boxes.
4. Create new account button opens up the register window, allowing users to create a new account.

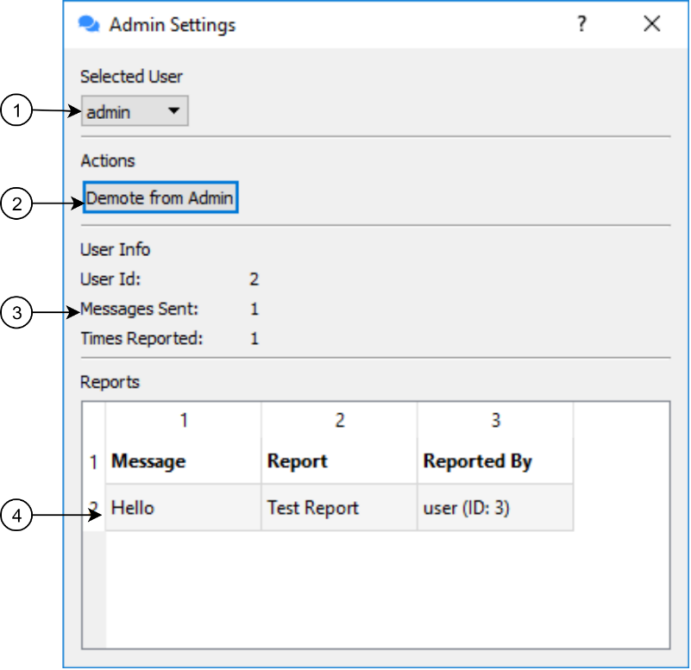


1. Input for the username of the new account.
2. Input for the password of the new account. Displays text as ‘•’ characters.
3. Input for the confirm password of the new account. Displays text as ‘•’ characters. Must equal the box above.
4. Submits the account details to the server, attempting to create a new account
5. Label to display message timestamp.
6. Label to display message sender username.

Part of the message widget

1. Label to display actual message contents.
2. When hovered, the message widget darkens for ease of reading.
3. Button to open message options widget.
4. Button to open admin settings window. Only visible if logged in as admin.
5. Label to display count of online users.
6. Multiline label to display list of online users.
7. Text input to write new messages in.
8. Button to send messages. Also empties Message text input.
9. Window is fully resizable, with all elements scaling properly.



1. Timestamp & Sender display labels.
2. Text input to edit messages. Only enabled if user sent the message.
3. Submits message edit. Only enabled if user sent the message.
4. Deletes message. Only enabled if admin or sender of message.
5. Text input to register a report. Disabled on own or server messages.
6. Button to submit report.
7. Dropdown contains all registered users. Used to select user to edit.
8. Promotes/Demotes a user to/from admin.
9. Labels to display statistics/information about the selected user.
10. If a user has been reported, reports are loaded into this table for admins to look over.

## Algorithms

### Circular Queue

The Circular Queue Class is used for the database write queue and the logger in the server file.

|  |  |
| --- | --- |
|  |  |
| 1 | class CircularQueue(): |
| 2 | def \_\_init\_\_(self, maxSize): |
| 3 | if maxSize < 1: |
| 4 | raise ValueError("Queue size must be at least 1") |
| 5 | self.data = [''] \* maxSize |
| 6 | self.rear = -1 |
| 7 | self.front= 0 |
| 8 | self.size = 0 |
| 9 | self.maxSize = maxSize |
| 10 |  |
| 11 | def enQueue(self, item): |
| 12 | if self.size == self.maxSize: |
| 13 | raise ValueError("Cannot enqueue when the queue is full") |
| 14 | else: |
| 15 | self.rear = (self.rear + 1) % self.maxSize |
| 16 | self.data[self.rear] = item |
| 17 | self.size = self.size + 1 |
| 18 |  |
| 19 | def deQueue(self): |
| 20 | self.front += 1 |
| 21 | self.size -= 1 |
| 22 | return self.data[self.front-1] |
| 23 |  |
| 24 | def isFull(self): |
| 25 | return self.size == self.rear |
| 26 |  |
| 27 | def isEmpty(self): |
| 28 | return self.size == 0 |

### Hashing Algorithm

The hashing algorithm is one-directional and is used for encrypting passwords before storing them in the database.

|  |  |  |
| --- | --- | --- |
|  |  | |
| 1 | def hashString(string): | |
| 2 | bitValueChunk = "" | |
| 3 | bitSum = 0 | |
| 4 |  | |
| 5 | for char in string: | |
| 6 | bitSum += ord(char) | |
| 7 | bitValue = format(ord(char), 'b') # Convert char to binary | |
| 8 | bitValueChunk += bitValue # Append to 'binary chunk' | |
| 9 |  | |
| 10 | n = 9 | |
| 11 | bitValues = [bitValueChunk[i:i+n] for i in range(0, len(bitValueChunk), n)] # Split 'binary chunk' into list of 9 bit binary numbers | |
| 12 |  | |
| 13 | moddedBitChunk = "" | |
| 14 | for bitValue in bitValues: | |
| 15 | bitValue = int(bitValue) | |
| 16 | moddedBitValue = bitValue + (bitValue % 37) # Modulo is a one way function, so we modulo by a prime as the core of the hash. This is the step that ensures the hash is unidirectional | |
| 17 | moddedBitValue = moddedBitValue \* bitSum # Multiply by sum of the ascii values of the chars to ensure similar input strings look different | |
| 18 | moddedBitChunk += format(moddedBitValue, 'b') # Convert into binary again | |
| 19 |  | |
| 20 | n = 6 # 6 bit chunks to avoid strange characters | |
| 21 | moddedBitValues = [moddedBitChunk[i:i+n] for i in range(0, len(moddedBitChunk), n)] # Split modded 'binary chunk' into list of 8 bit binary numbers | |
| 22 |  | |
| 23 | hashed = "" | |
| 24 | for moddedBitValue in moddedBitValues: | |
| 25 | hashed += chr(int(moddedBitValue, 2) + 33) # Convert binary value into decimal value then into the corresponding character, skipping the first 33 as they are non-printing/whitespace | |
| 26 |  | |
| 27 | return hashed | |
|  |  | |

### Integer Merge Sort

The integer merge sort is not actually used anywhere in the program, but was written as a prototype in order to get the string merge sort working, as they are very similar algorithms.

|  |  |
| --- | --- |
|  |  |
| 1 | def integerMergeSort(mergelist): |
| 2 | if len(mergelist) > 1: |
| 3 | mid = len(mergelist) // 2 # Perform integer division |
| 4 | lefthalf = mergelist[:mid] # Left half of merglist into lefthalf |
| 5 | righthalf = mergelist[mid:] # Right half of merglist into righthalf |
| 6 | lefthalf = MergeSort(lefthalf) |
| 7 | righthalf = MergeSort(righthalf) |
| 8 |  |
| 9 | i = 0 |
| 10 | j = 0 |
| 11 | k = 0 |
| 12 | while i < len(lefthalf) and j < len(righthalf): |
| 13 | if lefthalf[i] < righthalf[j]: |
| 14 | mergelist[k] = lefthalf[i] |
| 15 | i += 1 |
| 16 | else: |
| 17 | mergelist[k] = righthalf[j] |
| 18 | j += 1 |
| 19 | k += 1 |
| 20 |  |
| 21 | # Check if left half has elements not merged |
| 22 | while i < len(lefthalf): |
| 23 | mergelist[k] = lefthalf[i] # If so, add to mergelist |
| 24 | i += 1 |
| 25 | k += 1 |
| 26 | # Check if right half has elements not merged |
| 27 | while j < len(righthalf): |
| 28 | mergelist[k] = righthalf[j] # If so, add to mergelist |
| 29 | j += 1 |
| 30 | k += 1 |
| 31 | return mergelist |

### String Merge Sort

The String Merge Sort is used to sort the list of online users alphabetically.

|  |  |
| --- | --- |
|  |  |
| 1 | def stringListMergeSort(mergelist): |
| 2 | if len(mergelist) > 1: |
| 3 | mid = len(mergelist) // 2 # Perform integer division |
| 4 | lefthalf = mergelist[:mid] # Left half of merglist into lefthalf |
| 5 | righthalf = mergelist[mid:] # Right half of merglist into righthalf |
| 6 | lefthalf = stringListMergeSort(lefthalf) |
| 7 | righthalf = stringListMergeSort(righthalf) |
| 8 |  |
| 9 | i = 0 |
| 10 | j = 0 |
| 11 | k = 0 |
| 12 | while i < len(lefthalf) and j < len(righthalf): |
| 13 | l = 0 |
| 14 | while ord(lefthalf[i][l]) == ord(righthalf[j][l]) and l < len(lefthalf[i])-1 and l < len(righthalf[j])-1: # If the charachers are the same, we must look at the next one until the end |
| 15 | l += 1 |
| 16 | if ord(lefthalf[i][l]) < ord(righthalf[j][l]): |
| 17 | mergelist[k] = lefthalf[i] |
| 18 | i += 1 |
| 19 | else: |
| 20 | mergelist[k] = righthalf[j] |
| 21 | j += 1 |
| 22 | k += 1 |
| 23 |  |
| 24 | # Check if left half has elements not merged |
| 25 | while i < len(lefthalf): |
| 26 | mergelist[k] = lefthalf[i] # If so, add to mergelist |
| 27 | i += 1 |
| 28 | k += 1 |
| 29 | # Check if right half has elements not merged |
| 30 | while j < len(righthalf): |
| 31 | mergelist[k] = righthalf[j] # If so, add to mergelist |
| 32 | j += 1 |
| 33 | k += 1 |
| 34 | return mergelist |