Daniel Wait| Candidate No.

[school] | Centre No.

Messenger App

Unit 3: programming project

For OCR Specification H446

Contents

[1 Analysis 2](#_Toc69405407)

[1.1 Problem Identification 2](#_Toc69405408)

[1.1.1 Problem outline 2](#_Toc69405409)

[1.1.2 How can the problem be solved by computational methods? 2](#_Toc69405410)

[1.1.3 Computational methods that the solution lends itself to 2](#_Toc69405411)

[1.2 Stakeholders 3](#_Toc69405412)

[1.2.1 Identification 3](#_Toc69405413)

[1.2.2 Questions and methodology 4](#_Toc69405414)

[1.2.3 Results analysis 4](#_Toc69405415)

[1.2.4 Conclusion 7](#_Toc69405416)

[1.3 Research 7](#_Toc69405417)

[1.3.1 Existing program – Discord 7](#_Toc69405418)

[1.3.2 Existing solution – Internet Relay Chat (IRC) 10](#_Toc69405419)

[1.4 Requirements 11](#_Toc69405420)

[1.4.1 Stakeholder requirements 11](#_Toc69405421)

[1.4.2 Limitations 11](#_Toc69405422)

[1.4.3 Software requirements 11](#_Toc69405423)

[1.5 Success Criteria 12](#_Toc69405424)

[2 Design 13](#_Toc69405425)

[3 Development 13](#_Toc69405426)

[4 Evaluation 13](#_Toc69405427)

[5 References 14](#_Toc69405428)

# Analysis

## Problem Identification

### Problem outline

Currently, the encryption in-transit system is widely used in instant messaging apps. This means that a user’s messages will be in plaintext in the service provider’s server, which allows the service provider to read users’ conversations breaching their privacy. These companies are then also susceptible to potential attackers accessing confidential information through this potential backdoor. Therefore, I will be making a solution to this problem of user privacy by making an end-to-end encrypted instant messaging program.

For this solution to work, the following features are required: only the endpoint users have the key for the symmetric encryption algorithm to qualify the system as end-to-end encryption; an easy-to-use GUI so that anyone can navigate and use the program; a login/sign-up system to authenticate users’ identities to the server to facilitate historic message viewing; networking capabilities in order send and receive messages from different networks with a low enough latency to be a viable ‘instant’ messaging platform.

### How can the problem be solved by computational methods?

This problem is well suited to a computational approach as it can be solved using computational methods. This is because the processes of encrypting, sending then decrypting messages over a network can be greatly abstracted for the user so that they do not need any prior technical knowledge to use the program. The solution will also have many algorithms to carry out processes without the need for complex input from the user.

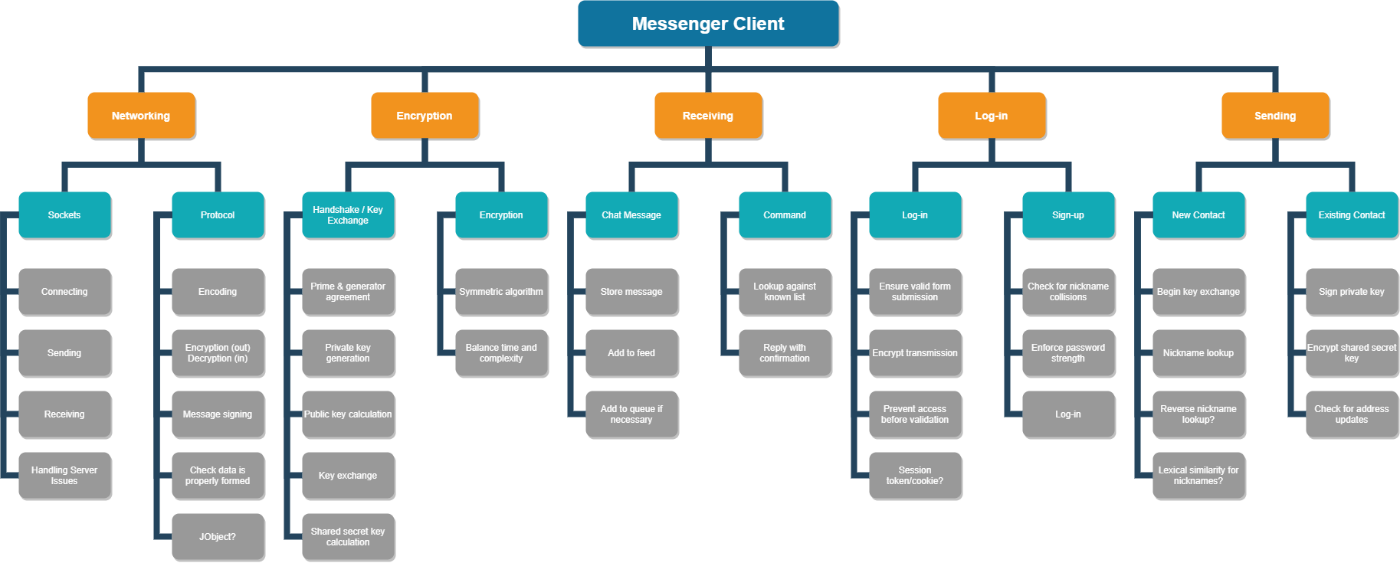
### Computational methods that the solution lends itself to

**Problem recognition** – The general problem is creating a program where sensitive information is not ever in plaintext whilst passing through the server. However, the more specific problems are creating a system of generating two symmetric private keys over a network connection that people monitoring the network are unable to recreate and sending and receiving the messages over that network. Once this is overcome the rest of the solution is using those keys to do the encryption/decryption and presenting the messages to the user in an intuitive form.

**Abstraction** – It is not necessary (or feasible) for the user to have hands-on control over every single process that happens when sending their messages. Therefore, abstraction will be used to hide most of the processes from the user to streamline their experience by only showing relevant detail. Some of these processes will be the following: encrypting and decrypting their message; segmenting their messages into packets; sending the packets to the server.

**Thinking ahead** – I have thought ahead by choosing C# to write the program is since I expect the backend to require a lot more focus than the frontend; the GUI capabilities in C# are much greater than Python for example (the language I am most familiar with), due to its support for WYSIWYG GUI builders in Visual Studio. I made this decision so that I can spend more time focusing on the development of the backend.

**Procedural thinking & decomposition** – The problem can be decomposed into a set of much smaller problems, using procedural thinking I will develop a solution to each of these in turn.



This set of problems will allow me to possibly use test harnesses during development to isolate certain parts of the program during the development process. This structure will make the overall solution easier to work with and will make the entire process more efficient.

**Thinking concurrently** – Through the use of concurrent processing, I will make the server-side program more efficient by processing each user’s requests on a virtual thread. This will mean that the program can deal with requests from multiple users and the same time, which is important as if they were acted on procedurally the bandwidth of the system would be dramatically reduced leading to very high latency.

**Performance modelling** – I will make heavy use of performance modelling to ensure that the program functions efficiently. On the smaller scale, I will use performance modelling to profile my encryption algorithms to make sure that they properly balance processing time and security since for this application an algorithm that sufficiently encrypted data in a few milliseconds is vastly superior to an algorithm that more complexly encrypts data in a few minutes. On a larger scale, I will use performance modelling to ensure that my server-side program has properly utilised threads to have a suitable bandwidth to support both average and peak throughput.

## Stakeholders

### Identification

The first group of potential stakeholders are the users who will prioritise privacy and security. The needs of this group are that all communications are encrypted so they cannot be read while passing through the server and that their sensitive information (e.g., passwords and keys) are protected while they are being stored. This group will most likely use the solution for everyday use as their main communication platform; for this reason, the solution must be robust enough for that use case. I have selected a user to represent this group of stakeholders: Ethan S. He is a student who believes that privacy is very important, especially online where he makes a conscious effort to minimise his digital footprint by limiting any personally identifiable information about himself. For these reasons, I believe that he will be a fair representative for this group’s needs.

The second group of potential stakeholders are the users who want a messenger app that is lightweight and easy to use. The needs of this group are that all components of the user interface are intuitive and clearly labelled and that the program requires no prior setup or configuration so the program can easily be installed and immediately used. This group will most likely only infrequently use the solution to keep in touch with friends and family; for this reason, the solution must have an easy system for finding contacts and have a low barrier of use as to not discourage them. I have selected a user to represent this group of stakeholders: [to be added]

### Questions and methodology

To help me better understand the requirements of the stakeholders, I have created a survey to send to them (since in-person interviews are not possible at this time).

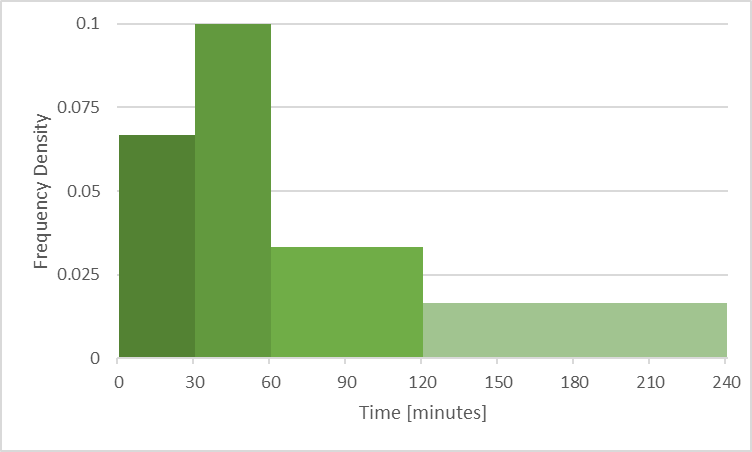
**Aims for the survey:**

Firstly, I want to investigate the stakeholders’ current use patterns with messaging apps. I believe this will give me a good insight into standards and expectations. This is especially relevant for my stakeholders whose needs involve an easy-to-use platform as I will research some of the most used platforms and see what makes their user interface so accessible.

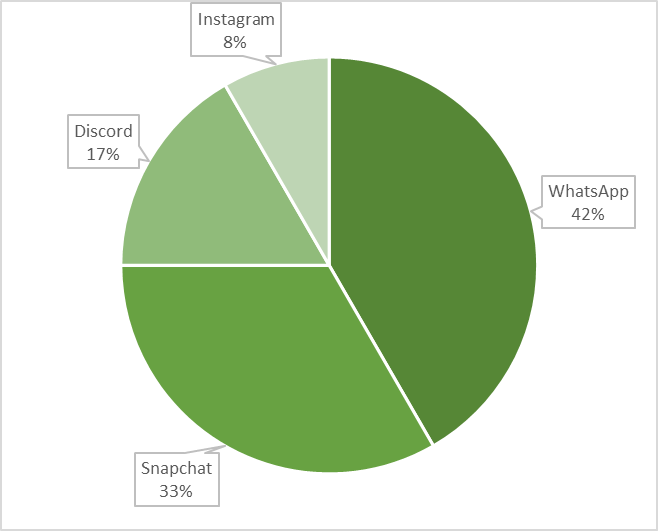
Secondly, I wanted to hear the stakeholders’ opinions on some common features of messaging apps. This information from the stakeholders will be crucial as I will use it to inform my decision of whether a feature is worth including in my final solution.

### Results analysis

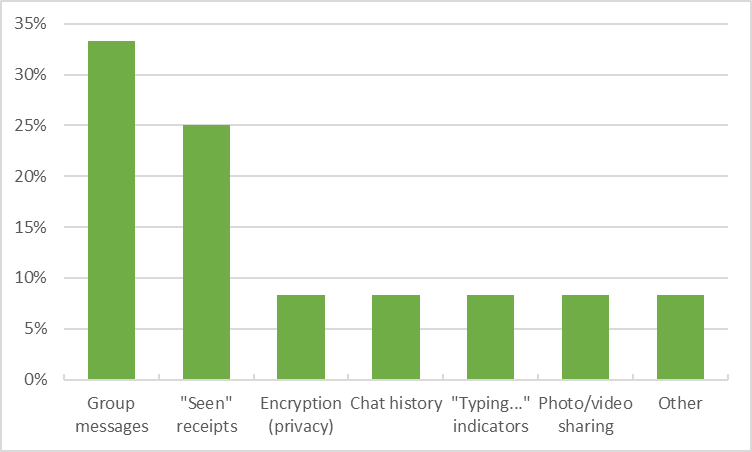
**Question 1 – “How much time do you spend on messaging apps each day?”**

****

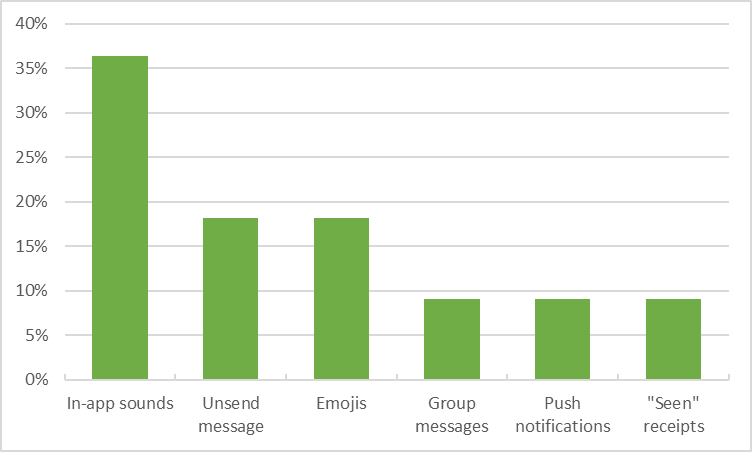
**Question 2 – “What is your most used messaging app?”**

****

**Question 3 – “What is your favourite feature of messaging apps?”**



**Question 4 – “What is your least favourite feature of messaging apps?”**

****

**Question 5 – “How important are the following features?”**



The data from this question showed me that the stakeholders do not think that the following features make a messaging app: in-app sounds, emojis, “typing…” indicators, and unsending messages. From this I can see that these features are non-essential so I will not consider these for the list of necessary features for my solution.

The data also showed me that there are four features that are generally deemed core features: group messages, individual messages, encryption, and file sharing. As these are important for the stakeholders, I will consider these for the list of features that must be in the final solution.

### Conclusion

## Research

### Existing program – Discord

Discord is a free instant messaging and VoIP platform created in 2015 centred around enabling communities to connect through guilds: collections of chat rooms and voice channels. In 2019 the platform saw 250 million users with a total of 25 billion messages being sent per month [1] making it one of the largest gaming-focused communications platforms available.



**Features:**

Each guild is managed by users with varying levels of permissions which can be as general or customised as desired by the guild owner. In the text channels, you can send files, links, and text with basic mark-up. In voice channels, as well as streaming audio you can also use webcams and stream your desktop. The purpose and scope of a guild very flexible as there is no predefined structure: they can be created for a small group of friends or as large official game hubs.

**Differences:**

Discord is not a privacy-focused platform and users are expected to forfeit their privacy in exchange for ease of use and versatility. Discord uses the encryption in-transit system meaning that all traffic is decrypted on the server-side; for non-audio/video data, the HTTPS protocol is used which is encrypted using TLS or SSL. It is also known that Discord inspects all user traffic whilst it passes through their server. However, the reason why is unknown.

**Parts I can apply to my solution:**

1. The idea of having a central area consisting of multiple ‘channels’ for different conversations or topics would potentially be worth including in my solution. This is because it would be an intuitive structure for larger chat rooms, where having a singular area for everyone to talk in would not be feasible.
2. I also think that Discord’s feature of activity statuses for each user would be a good addition to my solution. This would not be an essential feature but, if I have enough time to add it, it will improve the overall experience of the user by enabling them to see who is also online while they are using the program.

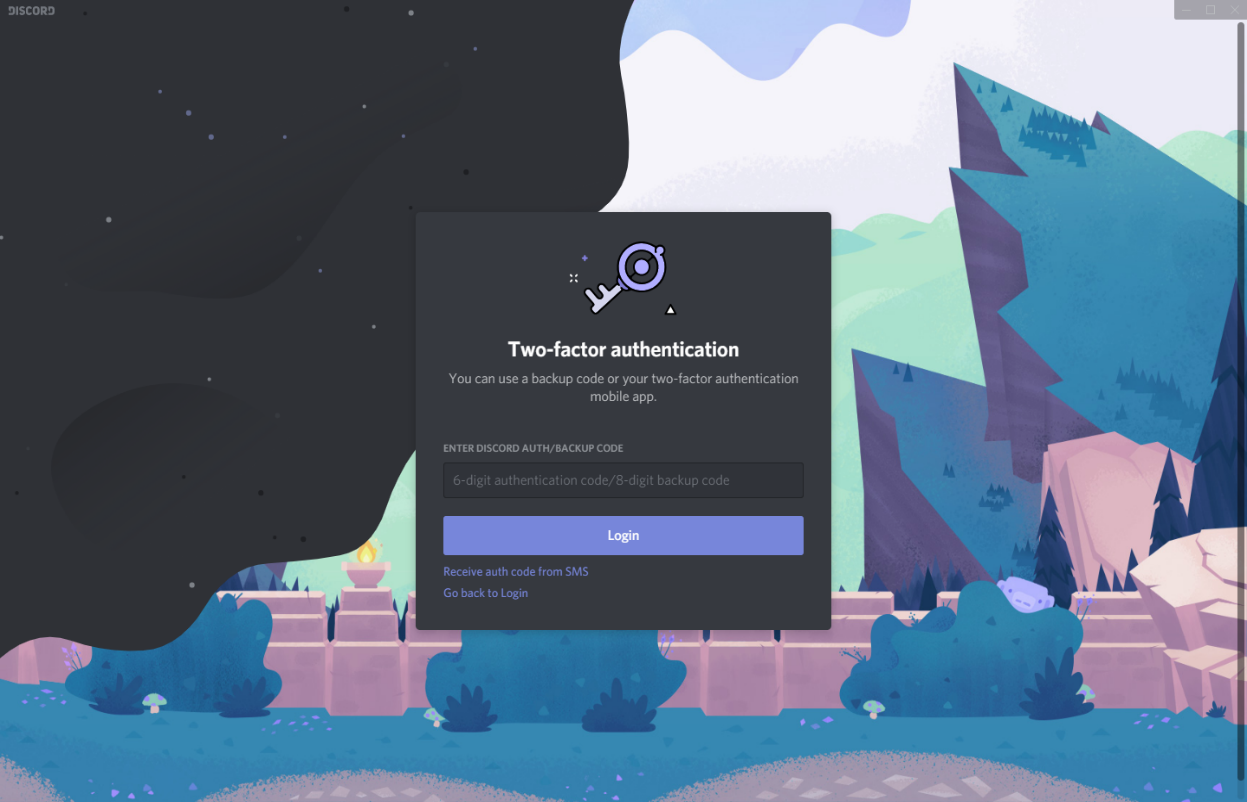
**Splash Screen:**

When starting Discord, a splash screen appears showing the user that processes are occurring behind the scenes. This is useful as it gives visual feedback to the user and gives the program time to connect to the server and load all the required resources into the RAM.

I will consider this feature for my solution as it will give my program time to establish a server connection before the user can try to log in without appearing to the user as the program has frozen.

**Logging In**

To use Discord, users must first log in. This can be done with an email and password or by scanning a QR code from the Discord mobile app. In my solution, I will also require the user to log into an account before they can use the program. This is because protecting messages behind a password improves privacy which is one of the requirements for the program. A login system will also be useful as it provides each user with a unique identifier that can be used when addressing messages and viewing contacts.

**Two-factor authentication**

Discord has the option to enable two-factor authentication. This means that when logging in to your account you need both the correct credentials and access to another method of proving your identity. Common methods are SMS message, email or a dedicated authenticator app, Discord chose the latter.

I will consider this feature for my program as it will add more security to the login process which would further satisfy the stakeholders’ requirements.

**Creating an account**

When creating a Discord account, usernames are case sensitive and are automatically postfixed with a number after a ‘#’ called a discriminator. This robust naming system is done to allow up to ten thousand people to have the same username. I will consider implementing this feature in my solution as preventing username collisions improves the experience for the user by allowing them to continue using their username from any other service.

Discord’s password requirements are on the other hand very lenient: “Must be between 6 and 128 characters long” [2]. Therefore, it is up to the user to choose a sufficiently broad character pool for their password which many users will forgo in exchange for convenience. So, for my solution, I will consider implementing some further requirements for the strength of passwords. This is since the privacy given by the end-to-end encryption will be undermined by an easily guessed password.

**Adding friends**

To add a friend (contact), you enter their complete Discord Tag (username) and send them a Friend Request. A complete username is needed as there are many users of Discord who have the same username so the only thing that differentiates these users is their discriminator. The recipient of this request can then accept or decline this request.

In my solution, I will consider a similar method of adding contacts by searching their username. However, since my user base will be much smaller than that of Discord, I may give the user feedback to similar names to what they entered by pattern matching the string they want to search and a list of all registered accounts.



**Home page**

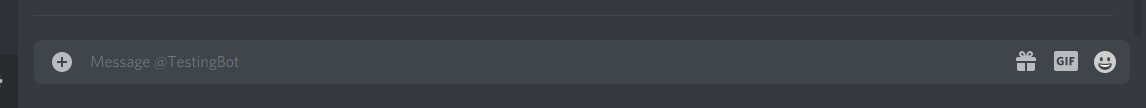
The default home page for Discord is a list of the user’s online friends. This page consists of four tabs: “Online” the default; “All” which includes offline friends; “Pending” which are accounts that the user has sent friend requests to and “Blocked”. Upon clicking on a friend in either of the first two tabs, a DM (direct message) thread is opened. This gives users quick access to all their DM threads as well as giving them a helpful overview of who is currently online.

I will consider having a similar homepage in my solution as immediate, easy access to conversations will greatly improve the flow of the program for the user. The online statuses and overview of online friends will also be a helpful addition to the program.

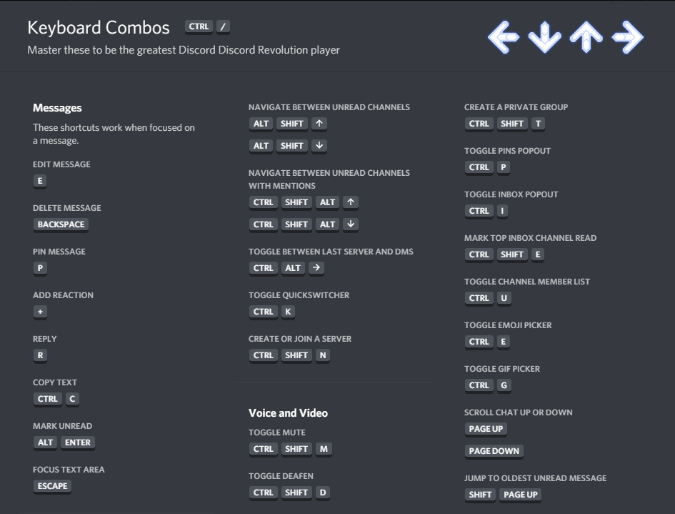
**Direct Messages**

DM threads on Discord are the conversations between users. These can show historic conversations if the users have sent each other messages in the past or empty when beginning a conversation. The messages that users send each other can be generalised as one of two formats:

1. Text, up to 2000 characters
   * Emojis
   * Embeds
   * Colouring
   * Code blocks with syntax highlighting
   * All the other Markdown formatting features
2. Files, up to 8MB (100MB with subscription)
   * Images (including GIFs) with previews
   * Playable audio files and videos with picture-in-picture
   * All other file types must be downloaded
     + Executable files cannot be sent for security reasons



In my solution, I will consider including the feature of viewing historic messages since if messages were lost after being viewed this would not make the program very helpful for the user. Secondly, I will consider including the ability to send multiple types of messages. This is because limiting the program to only text would be restrictive for the users when compared to alternative programs.

**Navigation**

Discord has many shortcuts which are helpfully listed on a dedicated help screen in the program. This streamlines the experience for power-users of the program while allowing regular users to continue using their normal cursor orientated navigation.

Many parts of the Discord UI are divided into groups of tabs: servers, DM threads and channels are all formatted as such. This makes the program very friendly for most users as tab-based navigation is a common UI style in websites, mobile apps, and desktop applications.

For my solution, I will consider also using tab-based navigation as it is well established and fits well with messaging apps and their lists of contacts/conversations. I may also consider giving the user the option to operate some features in my program via keyboard shortcuts, albeit on a smaller scale to Discord as complete keyboard navigation is not a requirement.

### Existing solution – Internet Relay Chat (IRC)

Internet Relay Chat is an internet protocol created in 1988 to allow group plaintext conversations with channels working on a client-server model or to individuals with private messages using the Direct Client-to-Client protocol (DCC). In February 2005, at the height of IRC the largest network – QuakeNet – saw a peak user count of almost a quarter of a million users [3]. This has dramatically reduced since then and is now at an average of 10 thousand users [4]. However, the protocol is still used by some services today as a means of lightweight communication typically attached to a larger service: The Twitch IRC network is responsible for the live chat in a Twitch stream and some games such as Tabletop Simulator, StarCraft, and Unreal Tournament use IRC for their in-game chat.



*Image via* [*WeeChat.org*](https://weechat.org/about/screenshots/)

**Networking:**

The structure of an IRC network is a spanning tree, in which clients will connect to one of the multiple servers that all share the same state. This introduces the first limitation of IRC: the fact that the networks are distributed becomes extremely inefficient with large networks as all the servers need to know about all the other servers, clients and channels every time something happens. The second limitation is that if one of the server-server connections was to go down, the network would split in half and many users will appear to have disconnected in what is called a netsplit.

**Difference:**

In my solution, I will use a centralised network. This means that I will not have to constantly share state information between servers like in an IRC network so the configuration and maintenance will be easier. However, the solution will be limited with scalability as the maximum throughput of the network is limited to the bandwidth of the one machine and the only way to scale up the network is to upgrade the parts in that machine.

**Group Messaging:**

To access IRC channels, users must first install an IRC client and select the domain name for the network they want to connect to. Once connected they will have to choose a display name. This is because users do not need to register to use IRC, only supply a short identifier in the form of a nickname. Finally, once they join a network’s channel, the network’s server they are connected to will relay all the messages they send to all the other users connected to the channel, and vice versa.

**Difference:**

In my solution, I will require users to register before they can use the program. This will help to prevent the nickname collisions which occurred with IRC where multiple people had/wanted the same nickname. This will ensure the all the messages are sent to the correct users and data validation can be used to make sure no two people accidentally share an identifier.

**Offline Messages:**

Some IRC networks support bouncers that enable offline messages, these are daemons on a server that act as a proxy for the client. When a client connects to a bouncer, the bouncer simply relays all the traffic to and from the server. However, in the event the client disconnects, the bouncer can store the messages that the client would have received if they were still connected. These will then be sent to the client once they reconnect. A similar implementation for offline messages is having a client run on an always-on server to which users connect to via SSH for their session. This also allows users who do not have an IRC client installed to connect.

**Part I can add to my solution:**

In my solution, I could include a way of archiving messages for users when they are not online. This could be implemented in a similar way to the bouncer where if the server detects that the client is no longer connected it will reroute the messages to a daemon. However, I will need to find a way of securely storing the user’s messages as security is a focus point of the solution.

**Typical Client User Interface:**

The layout of the UI for many IRC clients is the following: channels on the left, a nickname list on the right and the chat in the middle (this has become a common chat program layout as can be seen in Discord’s GUI in the section prior). In the past and in addition to standalone programs, Opera came with an IRC client attached to Opera Mail and there was an IRC client for Firefox called ChatZilla. This proved that IRC was a very lightweight protocol with not many needs besides a socket to run off.

**Part I can include in my solution:**

In my solution, I will also try to create a lightweight protocol that only requires a single socket as it is a requirement for my solution to create a lightweight program. I also think that sticking to the tried and tested chat program layout shown in many IRC clients will be a good inclusion into my program; since it will make using the program a lot easier for users who have used other chat programs in the past and it seems like an intuitive design for new users.

## Requirements

### Stakeholder requirements

### Limitations

**Hardcoded server IP:**

When a client tries to connect to the server, it will use a hardcoded IP address as its target. This means that the IP address of the server must be static and cannot be moved onto another network. For this limitation to be fixed the server would have to be added to a DNS server so that the domain can dynamically point to the server. However, this is beyond the scope of the project.

**Group messaging:**

Group messaging – the most chosen “favourite feature” and rated the third most important feature from the stakeholders’ survey – will not be implemented in the solution. This is down to the vast increase in complexity from individual end-to-end encrypted messaging to group end-to-end encrypted messaging; implementing such a feature will take up too much time and would require the redesign of many of the procedures of the solution. For these reasons, I will be unable to implement the feature.

### Software requirements

For simplicity, I will only be building a Windows x86 version of the solution for development and the final evaluation. However, using the dotnet compiler, executables for all the following operating systems can be built from the source.

|  |  |  |
| --- | --- | --- |
| OS | Version | Architectures |
| Windows | 7 SP1+, 8.1 | x64, x86 |
| Windows 10 | 1607+ | x64, x86 |
| Mac OS X | 10.13+ | x64 |
| Fedora | 32+ | x64 |
| Debian | 9+ | x64, ARM32, ARM64 |
| Ubuntu | 20.10, 20.04, 18.04, 16.04 | x64, ARM32, ARM64 |

Information from the .NET Core GitHub repository [5]

The final user of the program will not be required to install the .NET runtime as the solution will be published self-contained. This means that the download will be larger as it will contain the .NET libraries, runtime and dependencies needed.

Internet access will be required to run the program as the client program needs to communicate with the server.

## Success Criteria

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Requirement | Justification | Reference |
| 1 | Client sockets connect to server at start-up of the program | The app needs a connection to the server so it should connect while the app starts to minimise waiting time | Discord (splash screen) |
| 2 | Client socket tells the server it is closing before the app is closed | Prevents any errors from occurring and begins the client disconnect procedure |  |
| 3 | The socket sends heartbeats to the server to show that it is still open | Stops the possibility that a client has disconnected without the server realising which will lead to errors | IRC (PING) |
| 4 | User must log in to their account to access the program | Ensures that only people with valid credentials can view an account’s messages | IRC (?) or Stakeholders |
| 5 | New users can create an account | New users need a way of accessing the app | IRC (?) or Stakeholders |
| 6 | Users are not allowed to try to log in if the socket cannot connect | With no connection, logins cannot be authorised so the login process cannot be done |  |
| 7 | Usernames must be unique | Prevents situations where two people can accidentally share credentials | Existing solution - Internet Relay Chat (IRC) |
| 8 | Password must be of a minimum strength (upper, lower, digits, special characters) | Makes sure the password is not a security flaw for the user | Discord (account creation) |
| 9 | Users can type out a message and send it with a "Send" button | Intuitive button to send the message | Discord (UI) |
| 10 | Users can type out a message and send it with the Enter key | Enter is a common key to press to send a message | Discord (controls) |
| 11 | Users can see a list of their contacts | Users can easily see who they are sending the message to | Existing program - Discord |
| 12 | Users can click on a contact to message with them | No need to manually address each message they send like an email | Discord (UI) |
| 13 | Users can see past conversations with a contact | Users do not have to worry that old messages will be lost | Stakeholders survey |
| 14 | Users can be added to contacts by searching their username | Usernames are easier for a user to remember than an IP address | Discord (add friend) |
| 15 | Messages are encrypted with the recipient's public key | The recipients public and private keys are a keypair as it is asymmetric encryption |  |
| 16 | Messages are decrypted with the recipient's private key | Only the intended recipient can view the message as the decryption key is private |  |
| 17 | Messages are signed with the sender's private key | The sender can prove their identity by using a key that only they know |  |
| 18 | Signatures are checked with the sender's public key | The recipient must be able to prove that a message came from the user it claims to be from |  |
| 19 | Messages cannot be read while in the server | End-to-end encryption requires messages to not be able to be read during transit |  |
| 20 | Key pairs can be generated | Public and private keys are needed for the encryption of all messages |  |
| 21 | Public keys can be accessed by anyone | Public keys are used to check signatures and encrypt messages so they must be visible to everyone |  |
| 22 | The server can accept incoming client connections | Allows clients to connect to the server when they are opening the app |  |
| 23 | Each connected client is handled a separate thread | Concurrency allows for greater scalability in the number of clients and reduces their response time |  |
| 24 | The server authorises logins against its database | Only clients supplying valid credentials can view a user's messages |  |
| 25 | The server can create new accounts | New clients must be issued an account to start using the app |  |
| 26 | Messages are routed to the intended recipient | Users must only receive messages addressed to them | Existing solution – Internet Relay Chat (IRC) |
| 27 | Messages are held in the server if the recipient is not connected | Users may not be connected when a message is being sent so that must be accounted for | Existing solution – Internet Relay Chat (IRC) |
| 28 | Undelivered messages are sent the next time the recipient connects | Users should no miss out on messages because they were not online | Existing solution – Internet Relay Chat (IRC) |

# Design

# Development

# Evaluation

# References

|  |  |
| --- | --- |
| [1] | C. Corberly, “Discord has surpassed 250 million registered users,” TechSpot, 13 May 2019. [Online]. Available: https://www.techspot.com/news/80064-discord-has-surpassed-250-million-registered-users.html. [Accessed 12 February 2021]. |
| [2] | A. Schmelyun, “Password requirements for Discord,” [Online]. Available: https://passhints.co/discord/. [Accessed 13 April 2021]. |
| [3] | A. Gelhausen, “IRC Networks - Top 10 in the annual comparison,” Netsplit, 2005. [Online]. Available: https://netsplit.de/networks/top10.php?year=2005. [Accessed 19 February 2021]. |
| [4] | A. Gelhausen, “IRC Network QuakeNet,” Netsplit, [Online]. Available: https://netsplit.de/networks/QuakeNet/. [Accessed 19 February 2021]. |
| [5] | Collaborative, “.NET Core 3.1 - Supported OS Versions,” 15 October 2019. [Online]. Available: https://github.com/dotnet/core/blob/main/release-notes/3.1/3.1-supported-os.md. [Accessed 18 March 2021]. |
| [6] | C. Kalt, *Internet Relay Chat: Architecture,* RFC 2810 ed., 2000. |
| [7] | J. Oikarinen and D. Reed, *Internet Relay Chat Protocol,* RFC 1459 ed., 1993. |