Software Design and Development Mini Major Work

# Problem Definition

In many professional settings, messaging applications are used to facilitate asynchronous, text-based communication within teams or entire businesses. The use of these applications has transformed the manner of communication in the workplace, increased productivity and allowed professionals to collaborate without the need to physically meet. However, asynchronous communication through messaging applications has also introduced some communication challenges which were not prevalent before. One such challenge is the challenge associated with receiving many messages and not being able to easily determine which messages are most important or timely. This can greatly hinder the efficiency of a team or company and while often the most important messages are naturally highlighted due to being the most recent, this is not always the case.

The purpose of this project is to create an asynchronous messaging application for professional and organisational communication, especially for organisations engaged in remote work, which partially solves this problem. It will achieve this in part with simple widgets that can reduce the number of messages that need to be sent in common situations, such as for scheduling a meeting. Users may choose to include a widget in a chat, or widgets may automatically replace messages to enhance clarity and declutter the user interface. Further, messages will be organised systematically using channels in a similar manner to Discord. This will significantly reduce the number of messages sent, hence decluttering the user interface.

## Functionality Requirements

In order to be a success, the application must contain all the core functionality of a standard messaging application with threads. This means that a user should be able to complete the following tasks.

* Create an organisation and add a given list of people to it.
* Create a group and add anyone in the organisation to that group.
* View a list of the groups to which they belong.
* Create a channel within a group.
* Send a message in a channel to which they belong and delete that message later if they choose.
* View messages sent in a channel
* Leave a group.
* Delete a group.

Further, the application must contain specific functionality in order to solve the problem outlined above. This means that the following criteria must be met:

* A group of users can use widgets to easily achieve the following tasks:
  + Schedule a group meeting or event by getting everyone’s availability
  + Hold a vote/survey (public, anonymous or result only)
* These widgets are automatically applied in simple cases, such as when keywords are used in messages, or can be selected by the user.

## Performance Requirements

The application must work efficiently. For the purpose of this project, this means that given a decent internet connection, the user should never have to wait for more than five seconds for any operation to be carried out, whether that be creating a group, sending a message to the server, retrieving a message from the server or any other task. The exception to this is to load the application, which may take up to ten seconds. Ideally, the performance will be significantly better than this, but this is simply a minimum standard for the project.

## Compatibility Requirements

The application must be compatible with both MacOS (V10 and above) and Windows (7 and above) operating systems. As this application is meant for professional collaboration, it is not important that the program can be used on mobile devices as almost all established workplaces would have individual laptops.

## Problem Boundaries

While there are many aspects of this project, there are also some components that lie outside of the scope of this project. Specifically, the implementation of a Graphical User Interface Engine is not required, as Kivy will be used. Further, the implementations of encryption and hashing algorithms is not in the scope of this project, and instead existing implementations of the Scrypt, AES and RSA algorithms will be used. This is partially because the task of designing and implementing encryption algorithms is a particularly large task, comparable in size to the rest of the project. However, the main reason is because security is a serious issue that is easily made vulnerable if not implemented properly, and thus is best left to experts in the field.

# Discussion of Relevant Issues

## Existing Solutions

There are many messaging applications currently available, and many approaches have been taken to solve the relevant problem. However, none of these provide a holistic solution.

### Slack

Slack is one of the most widely used applications for professional messaging. However, the application does not prioritise the reduction and organisation of messages, which is the problem which this project attempts to solve. There are certainly some optimisations made, such as the utilisation of threads for replies and the creation of reactions. Further, there are bots which could perform the tasks which the widgets in this project would perform, such as opening a survey. However, the bots are typically quite difficult to use, and often contribute to the problem by sending messages on chats. Therefore, while Slack has some tools that can contribute to the solution, they fall short of holistically solving the problem outlined above.

Piazza

Piazza is an application designed for collaboration in an educational setting, and specifically for supporting discussion in relation to answering student’s questions. The platform allows students to ask questions, and then for the students to collaborate on a *single* answer to that question, while the professors collaborate on a *single* answer to the question as well. This is a very interesting approach in general for reducing the number of messages and increasing clarity in the user interface. This design may be included as a widget in this project if time permits. However, the focus of Piazza is very limited, and thus does not render this project redundant.

Discord

Discord is an application originally focused on communication for gaming, however it has expanded dramatically from there and is used widely by professional corporations. This application has arguably come closest to solving the problem presented in this report. Within each group, users have channels to separate distinct conversations, which is an extremely valuable tool in sustaining a non-confronting, uncluttered user interface and will be replicated in this project. In addition, Discord contains voice channels, which are useful to reduce messages in synchronous communication, but are of no use for most communication which is asynchronous. Further, Discord has bots, much like Slack, which can provide similar widgets to those proposed in this project. However, the bots can be somewhat confusing to use and are interacted with primarily through messages, which can contribute to the clutter. Ultimately, bots in Discord and Slack are designed to automate and conduct a variety of tasks, rather than to improve the readability of chats. As a result, Discord does not holistically solve the problem which this project aims to solve.

## Modification of Existing Solutions

All of the applications outlined above are protected under a commercial licence, and are not available for modification or commercial redistribution. From relatively extensive research, there does not appear to be an adequate product that can be modified for the purpose of this solution. Even if there was a product under an open source licence that could be repurposed to create this application, it would likely be untrustworthy with the ethical handling of user data. Of course, there are some open source projects such as Android that do demonstrate adequate security practices, but many do not and it is sometimes difficult to know. Therefore, the most appropriate direction would be to create a new solution, rather than modifying an existing one.

## Social and Ethical Considerations

### Privacy and Security

In a messaging application, privacy and security are of utmost importance. This is because individuals send potentially confidential or private information to a server, which theoretically can be accessed by anyone. As a result, it is critical that the messages are encrypted with sufficiently complex algorithms, such as AES and RSA, and only ever decrypted offline. Further, all processing of messages or data should be done offline, on the user’s device.

Another important concern with privacy and security is the addition of people to groups after messages have already been sent. By default, an algorithm will not differentiate between messages sent before and after someone joins a group, and will display both types of messages to the user. This is the approach that services such as Discord and Slack have chosen, however it is likely not a great solution as information may be not appropriate for them to read. Further, if the wrong person is accidentally added to a group, then they have access to all of the confidential messages ever sent on the chat. This can be result using a simple conditional statement to check if the user was a member of the group when the message was sent. However, this is potentially incomplete as it may be possible for the user to decrypt the messages manually, given that they have the relevant key to decrypt the other messages. This issue is even greater when an individual leaves a group, for example when they have been fired, as they may still be able to decrypt and read messages on the chat.

The developer has brainstormed two solutions which avoid this problem, but each have drawbacks in terms of efficiency and storage. One such solution is to generate a new encryption key each time a user joins or leaves a group. However, this may not be viable in relation to efficiency or storage. A final alternative would be instead of using a common encryption key for an entire group, to encrypt the message individually with each member of the group. However, this increases the time complexity for sending a message from *O(1)* to *O(n)* where *n* is the number of members in a group. This may not be feasible with very large groups, especially if the encryption algorithm is complicated or the messages are large (eg. files were attached). However, the iMessages protocol does use this

### Safety

A final ethical concern is in relation to safety. One significant concern in this area is in relation to inappropriate or harmful messages being sent on the platform. It is important that users who receive this message have a safe and easy way to get support and stop receiving these messages. One common approach is to allow users to report other users, in which case relevant messages would be forwarded to the company, unknown to the sender. This approach seems appropriate for this application. In addition, users should be able to save and forward messages to others, in order to gain support. This feature will be supported by the application.

## Cost Effectiveness and Licencing Considerations

A typical software developer earns approximately $65-85/hour, and as a less experienced developer I would charge $65/hour for my services. Assuming I work for 80 hours over the course of the project, this expense would amount to $5200. The project will be coded in Python and Kivy, using the Python ‘sockets’ module to send messages to and from a server. All three of these are open source and so there would be no cost required to pay for them. However, a Microsoft Azure SQL server would be required, which is distributed on a shareware licence with a free-trial period, and requires payment thereafter. The costs for Azure are complicated and can vary significantly, but an appropriate upper bound would be $2/user/month. This would likely be a significant expense and would scale with the number of users. Another expense would be for personnel. Initially, a team of security experts would need to ensure that the software meets relevant security standards. After the product has launched, personnel would be required to maintain the code over time, to release updates, to manage safety reports and to provide customer service. This would also be a significant cost which would scale with the number of users, but not necessarily too significant. WhatsApp only hires fifty full-time employees, and thus it would seem appropriate that the number of employees for this application would also be limited. If this application had 50 full-time employees that were payed a salary of $100000, this would cost $5 million/year. If the project was to be launched commercially, it would be distributed on a Shareware licence with a Freemium Model. The plans would probably resemble the [pricing plans for Slack.](https://slack.com/intl/en-au/pricing) This would eventually compensate for the expenses.

# Design Specifications

## Users’ Perspective

### User Interface Design

The user interface must resemble the standard user interface for any messaging application. More specifically:

* The application must contain a login screen, which allows contains a username field and a password field, as well as a sign in button and a ‘forgot my password’ link to online help.
* Once logged in, the application must contain a screen showing the user each of their groups, which they can select to go onto.
* For each group, there should be a page which has a sidebar consisting of each of the channels, and on the main page a list of the messages and widgets on the selected channel, as well as an input field to write a message, with a button to send that message to the desired channel.
* Finally, there should be a settings page which can be accessed using a standard ‘toggle’ settings icon. This will allow users to change their name and password, modify the font size to enhance readability and change other settings.

For a more detailed specification of the user interface, see the storyboard. It is important to note that this user interface is very similar to user interfaces for similar products. This is important in order to minimise the learning curve for using this product, and to encourage the transition towards this product from other products.

### Social and Ethical Considerations

In addition to the social and ethical issues outlined in the section above, there are many social and ethical issues related specifically to the design specifications from the user’s perspective. In essence, messaging applications are fairly consistent in their structure, and it is important that this messaging application, at least in majority, conforms to the design choices that have been used frequently in applications like Discord and Slack. These standard designs are related to the user interface and have been described above.

## Developers’ Perspective

### Data Structures

While there are many data structures in the project, these are the main ones:

* Members: A sorted array consisting of records for all members of the platform. The records contain their name, email address, an array of groups they belong to, their organisation name, their double-encrypted password, their AES-encrypted RSA secret key, their RSA Public Key and their Salt.
* Inbox: An array consisting of strings for each item that a user receives. Each string encodes the sender, the message and any widgets for the given item.
* Groups: A sorted array consisting of records for each group that the user is a part of. Each group is a record with its name, its members and the names of its channels.

### Data Types

While there are many data types, these are some of the main ones:

* AES/RSA key – A string that acts as an AES or RSA Key
* Organisation Members Count – An integer representing the number of members in the organisation
* Total Members Count – An integer representing the number of members in the organisation
* Passwords Match – A Boolean representing if the password is the same as the confirm password

### Algorithms

Please see separate algorithms section

# System Documentation

## IPO Chart

|  |  |  |
| --- | --- | --- |
| **Input** | **Process** | **Output** |
| Sign Up: Organisation name and emails | Add organisation to Database  For each email:  Add email to the Database under organisation | ‘Success’ message |
| Add Organisation Member: Member’s email | Add email to the Database under organisation | ‘Success message’ |
| Remove Organisation Member: Member’s email | Remove email from Database under organisation |  |
| Log In: Email | Check for existence of email in database  If no email present, raise error  If email present, check if a password has been chosen  If password chosen, ask for password  If no password chosen, randomly generate salt. Then, ask for new password and confirm password | Input box for Password  OR  2 input boxes for new password and confirm password  OR  Error message “There is no account with this email” |
| Log In: New password and confirm password | Compare passwords  If passwords match, randomly generate a salt and hash one of them with salt using Scrypt. Store as AES Key variable.  Create RSA Public/Private Key pair. Store Public Key in database with email. Hash Private key using AES and store as in database with email.  Hash AES Key with salt and store in database as stored hash.  Show success message and send user to homepage  If passwords don’t match, raise error | Success message and homepage  OR  Error message |
| Log In: Password | Hash password with salt using Scrypt and store as AES key in variable  Hash key with salt using Scrypt and compare with stored hash  If they are the same, use AES key to decrypt RSA private key, store as variable and send user to homepage  If they are different, return error and send user to Log In: Password page | Homepage  OR  Error Message and Log In: Password Page |
| No Input: Occurs every few seconds when logged in | Receive new messages and widgets from database and perform data validation on them  Decrypt them using your private key and perform data validation again.  Remove the ends of the new messages, which will say who the message or widget was sent from. Then, decrypt with their public key and perform data validation a third time.  Store messages and widgets on system.  If on homepage, group page or channel, update user interface to reflect new messages  If any data validation fails, simply ignore. | Possible updated interface |
| Group Selected | --- | List of channels from group with the number of messages in each channel displayed next to it |
| Channel Selected | --- | List of messages and widgets from channel |
| Message/Widget Sent | Perform data validation – if invalid, raise error  Convert Widget/Message into plain text so that it can be sent  Encrypt using user RSA private key and concatenate user ID to the end  For each recipient, encrypt with their RSA public key and send to database | Their message/widget shown above in the list of sent items, and message box cleared. |

# Algorithms

## Algorithms to be Implemented

BEGIN sign\_up\_organisation

Get organisation\_name, member\_names, member\_emails, org\_member\_count

Let members(org\_member\_count) as record

FOR index = 0 TO org\_member\_count STEP 1

members(index) = record:

name: member\_names(index)

email: member\_emails(index)

organisation: organisation\_name

groups: group\_array(128) as record

salt: “”

double\_hashed\_pwd: “”

PK: “”

encrypted\_SK: “”

NEXT index

record\_bubble\_sort(members, org\_member\_count, email)

Read total\_member\_count from online server

total\_member\_count = total\_member\_count + org\_member\_count

Write sorted\_members to online server under members

Write “total\_member\_count = “, total\_member\_count to online server

END sign\_up\_organisation

BEGIN login

Get email

Read members, total\_member\_count from online server

FOR index=0 TO total\_member\_count STEP 1

IF members(index).email = email THEN

IF members(index).salt = “” THEN

login\_password\_new

ELSE

login\_password\_returning(members(index).salt, members(index).double\_hashed\_pwd,

members(index).encrypted\_SK)

END IF

END IF

NEXT index

END login

BEGIN login\_password\_new

Get password, confirm\_password

WHILE password ≠ confirm\_password

Raise Error

Get password, confirm\_password

END WHILE

salt = random\_salt()

scrypt(password, salt, AES\_key)

scrypt(AES\_key, salt, double\_hashed\_pwd)

RSA\_key\_generation(PK, SK)

AES(SK, AES\_key, encrypted\_SK)

Global SK

Update members on online database with PK, encrypted\_SK, salt, double\_hashed\_pwd

END login\_password\_new

BEGIN login\_password\_returning(salt, double\_hashed\_pwd, encrypted\_SK)

REPEAT

Get password

scrypt(password, salt, AES\_key)

scrypt(AES\_key, salt, pwd\_candidate)

UNTIL pwd\_candidate = double\_hashed\_pwd

AES(encrypted\_SK, AES\_key, SK)

GLOBAL SK

END login\_password\_returning

BEGIN send\_item(name, SK, item, recipients, recipients\_len)

RSA(item, SK, item\_encrypted\_once)

item\_encrypted\_once = name + “XXX” + item\_encrypted\_once

FOR index=0 TO recipients\_len STEP 1

RSA(item\_encrypted\_once, recipient.PK, item\_encrypted\_twice)

Write item\_encrypted\_twice to online database under recipient’s inbox

NEXT index

END send\_item

BEGIN receive\_items(SK, items)

WHILE inbox in database isn’t empty

index = 0

RSA(inbox(0), SK, item\_encrypted\_once)

sender = “”

REPEAT

sender = sender + item\_encrypted\_once(0)

remove the first character from item\_encrypted\_once

extract the 0th character (for 3 characters) of sender into three

UNTIL three= “XXX”

remove the last three characters from sender

Read PK of sender from online database under members

RSA(item\_encrypted\_once, SK, item)

items(index) = item

ENDWHILE

END receive\_items

BEGIN record\_bubble\_sort(array, length, key)

swapped = True

pass = 0

WHILE swapped = True

swapped = False

comparison = 1

WHILE comparison < length – pass

IF array(comparison).key > array(comparison+1).key THEN

temp = array(comparison)

array(comparison) = array(comparison+1)

array)comparison+1) = temp

swapped = True

END IF

comparison += 1

ENDWHILE

pass += 1

ENDWHILE

END record\_bubble\_sort

BEGIN random\_salt(salt)

hex\_digits(16) as int

hex\_digits = (‘0’,’1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’,’a’,’b’,’c’,’d’,’e’,’f’)

salt = “”

FOR index = 0 TO 16 STEP 1

random\_choice(hex\_digits, choice)

salt += choice

NEXT index

END random\_salt

## Standard Algorithms to be Imported

**scrypt(content, salt, hash)**

A subroutine that takes as input a string, commonly a password (content) and a randomly generated string (salt). It then deterministically calculates and returns a hash which is based on the content and the salt. It is a consistent measure that can be used to determine whether two passwords entered at different times were the same, without storing the original password. The distinguishing feature of scrypt, in comparison to something like SHA, is that scrypt is deliberately slow and thus it takes longer for an attacker to guess a user’s password using brute force or a dictionary attack. It also inherently incorporates a salt, which prevents against rainbow table attacks as common passwords won’t all be hashed to the same value due to differing salts.

**RSA\_key\_generation(pk, sk)**

A subroutine that non-deterministically produces a public key and a secret key, to be used in RSA encryption. Typically (and in the case of this project), the public key is made public for all and the secret key is kept privately.

**RSA(content, key, encrypted\_content)**

A subroutine that takes as input content to be encrypted and an encryption key, and returns the content encrypted with that key. The key can be either key from a pk-sk pair. Importantly, content that is first encrypted with a given public key and then encrypted with the corresponding secret key, or vice versa, simply reveals the original message. This means that messages can be encrypted with someone else’s public key to make the message only readable to them, and can be encrypted with your own private key to prove that you were the true recipient.

**AES(content, key, encrypted\_content)**

A subroutine that takes as input content to be encrypted and an encryption key, and returns the content encrypted with that key. Importantly, the AES algorithm is its own inverse, and so performing it twice simply returns the original content. This means that private data can be encrypted and only the person with the key can decrypt the content, even if the encrypted version is made public.

**random\_choice**(list, choice)

A cryptographically secure algorithm that takes as input a list and, using pseudorandom processes, returns a single choice from that list with a uniform distribution.