Instant messsenger

A-level NEA Project by Caitanya Vaghela Durley

Candidate number: 4022

Centre number: 31260

Contents

[Analysis 5](#_Toc7094035)

[Project Outline 5](#_Toc7094036)

[Client 5](#_Toc7094037)

[Research 6](#_Toc7094038)

[Instant Messengers 6](#_Toc7094039)

[User Interface 7](#_Toc7094040)

[Client-Server and Peer-to-Peer 8](#_Toc7094041)

[Connections 8](#_Toc7094042)

[Protocols 9](#_Toc7094043)

[Threading 9](#_Toc7094044)

[Storage 9](#_Toc7094045)

[Defensive Programming 10](#_Toc7094046)

[Encryption 10](#_Toc7094047)

[Hashing 12](#_Toc7094048)

[Group Chats 13](#_Toc7094049)

[Data Volume 14](#_Toc7094050)

[Objectives 14](#_Toc7094051)

[Server-side 14](#_Toc7094052)

[Client-side 14](#_Toc7094053)

[Design 15](#_Toc7094054)

[Overview 15](#_Toc7094055)

[User Interface 17](#_Toc7094056)

[Main Form 18](#_Toc7094057)

[Friend Requests Form 19](#_Toc7094058)

[Admin Form 20](#_Toc7094059)

[Database Design 21](#_Toc7094060)

[Entity - tblUsers 21](#_Toc7094061)

[Entity - tblMessages 21](#_Toc7094062)

[Entity - tblFriends 22](#_Toc7094063)

[Entity-relationship diagram 22](#_Toc7094064)

[Connection 22](#_Toc7094065)

[Threading 23](#_Toc7094066)

[Encryption 23](#_Toc7094067)

[RSA 23](#_Toc7094068)

[AES 24](#_Toc7094069)

[Hashing 24](#_Toc7094070)

[Noteworthy Data Structures 25](#_Toc7094071)

[Port Queue 25](#_Toc7094072)

[Master Class Dictionary 25](#_Toc7094073)

[OOP Overview 26](#_Toc7094074)

[Common classes 26](#_Toc7094075)

[Client-specific 26](#_Toc7094076)

[Server-specific 26](#_Toc7094077)

[Server UML 27](#_Toc7094078)

[Client UML 28](#_Toc7094079)

[Class Design 29](#_Toc7094080)

[AES and RSA 29](#_Toc7094081)

[distAndAuth 29](#_Toc7094082)

[transmission 30](#_Toc7094083)

[instances 33](#_Toc7094084)

[SHA 34](#_Toc7094085)

[databaseIO 34](#_Toc7094086)

[Form1 36](#_Toc7094087)

[findFriendsForm 36](#_Toc7094088)

[adminForm 37](#_Toc7094089)

[Non-class code 38](#_Toc7094090)

[Server – Main 38](#_Toc7094091)

[Client – ConstantsModule 38](#_Toc7094092)

[Group messaging 39](#_Toc7094093)

[Server Algorithms 40](#_Toc7094094)

[Main 40](#_Toc7094095)

[distAndAuth.getPort 41](#_Toc7094096)

[newThread 42](#_Toc7094097)

[instances.initialise 43](#_Toc7094098)

[transmission.New 44](#_Toc7094099)

[transmission.communicate 45](#_Toc7094100)

[transmission.logOffSub 46](#_Toc7094101)

[Key Exchange 47](#_Toc7094102)

[Authentication 48](#_Toc7094103)

[userOnlineRead – Sending past messages to user 48](#_Toc7094104)

[Hashing – Compute Hash 50](#_Toc7094105)

[Hashing – compare 50](#_Toc7094106)

[databaseIO 51](#_Toc7094107)

[Client Algorithms 70](#_Toc7094108)

[Logging in/out 70](#_Toc7094109)

[Key exchange 71](#_Toc7094110)

[distAndAuth.getPort 72](#_Toc7094111)

[Setup transmission 73](#_Toc7094112)

[transmission.streamRead 74](#_Toc7094113)

[transmission.streamWrite 74](#_Toc7094114)

[Populate friends list 75](#_Toc7094115)

[Sending a message 77](#_Toc7094116)

[Profanity detector 77](#_Toc7094117)

[Adding messages to the list 79](#_Toc7094118)

[Receive message(s) 84](#_Toc7094119)

[Receive Populate Friends Response 85](#_Toc7094120)

[Receive Search Friends Response 86](#_Toc7094121)

[Receive Admin Account 86](#_Toc7094122)

[findFriendsForm Algorithms 87](#_Toc7094123)

[Admin form algorithms 90](#_Toc7094124)

[Testing 93](#_Toc7094125)

[Outline 93](#_Toc7094126)

[Table 93](#_Toc7094127)

[Screenshots 98](#_Toc7094128)

[Evaluation 100](#_Toc7094129)

[Objectives 100](#_Toc7094130)

[Server-side 101](#_Toc7094131)

[Client-side 105](#_Toc7094132)

[Client feedback 109](#_Toc7094133)

[Potential extensions & improvements 110](#_Toc7094134)

[Conclusion 112](#_Toc7094135)

[Technical Solution 113](#_Toc7094136)

[Contents 113](#_Toc7094137)

# Analysis

## Project Outline

Networking comprises a large portion of modern computing. Communication unrestricted by geographical location improves productivity for workers, but also provides a platform for socialising, in the form of social media. In either case, exchanging messages over a WAN can be considered central, and this service is often extended to the public through an instant messenger. For my project, I wish to create an academically oriented instant messaging service that allows communication between two or more users over a WAN. My client for this project is Mr Deane, Head of Computing at Lawrence Sheriff School, who may allow students to use it to discuss subject related matters.

The project will be developed by first establishing the expected functionality of this project from Mr Deane, and then researching the methods by which this will be achieved. Similar systems to my project would be the likes of Pidgin for the instant messaging portion, and Google Classroom for the academia-oriented usage. These, and others, will be researched also. Once I have a sufficient understanding of the problem area, I will design the structure of the solution and the necessary algorithms. These can then be implemented in code. The research will begin by investigating current instant messengers and academic tools, and gradually become more specialised to suit the client’s needs.

## Client

As stated above, my client is Mr Deane, although likely end users include students in his classes. He believes an academically oriented messenger could improve productivity and allow students to collaborate on tasks both at home and in school. I interviewed Mr Deane to establish the desired aims of the project, a summary of which is shown below. Note these aims are numbered 1 to 5 to signify their importance, with 1 being essential and 5 being a luxury.

* Communication across computers via network - 1
* Encryption of messages and stored user passwords – 1
* Search for potential friends by username – 1
* The admin account approves friend requests before they are processed – 2
* Admin account facilities (monitoring messages, reset passwords, remove users) – 2
* Recommend friends, for example by recommending friends of friends – 3
* Word filter/profanity detection – 3
* Group messaging – 4
* Simple user interface – 5

Typical usage of this service was also discussed, with the following being suggested as most likely:

* Peer-to-peer communication to discuss subject related matters
* Teacher-to-teacher communication e.g. report behaviour, organize rooms etc.
* Teacher-to-peer(s) communication to set tasks, reprimand bad behaviour etc.

## Research

### Instant Messengers

An instant messenger is a means of transmitting messages over the internet in real time. Users do not have to be connected over a LAN, that is, users can be as geographically apart as desired and still use the service. Typically, messages are short pieces of text, sent from one user to another, however, modern implementations allow group messaging and transmission of images and other file types. Instant messenger services usually function through an installed piece of software or through browsers. Mr Deane would like a service similar to traditional instant messengers, but for academic use rather than for recreational purposes.

Pidgin is an example of a popular instant messaging client, with over three million users in 2007. It allows a user to log into various services, but centralises all messages to one application and friends list; supports end-to-end encryption, through the OTR plugin; provides file transfer, video chat and spell-checking facilities; and is based on the “libpurple” library. Many of these points are beyond the scope of my project but demonstrate the power of an advanced instant messenger.

Google Classroom is an online file sharing site for schools. Its main aim is to allow the distribution of assignments to students but it does support communication between students and teachers. “Comments” allow private conversations between a student and their teacher, while the “class stream” is the equivalent of a group chat. However, these equivalencies are not as streamlined as with Pidgin and feel more like an afterthought than a priority.

Popular systems used in schools are products from Impero Software, one of which is their flagship product Impero Education Pro. At its core, Impero supports staff by providing the ability to view students’ screens in real time, and take control in order to prevent distraction and complement traditional teaching. One feature pertinent to this project is their Messaging and Live Chat solution. It enables teachers to discreetly message students or other staff, through a simple live chat application. Of course, Impero is intended for use in a LAN, while this project is expected to function over a larger geographical area. Furthermore, it is exceedingly simple, providing very little besides a quick conversation between two devices. There is no memory of previous conversations, no concept of friends, and no authentication.

The most popular instant messengers are feature rich, seamlessly supporting abilities like synchronisation across devices, push notifications and group chats. School based solutions offer power to teachers to monitor students and maintain control.

### User Interface

I will adhere to the points made above for my own user interface; however, this is not a high priority for my project.

### Client-Server and Peer-to-Peer

The two main networking architectures are client-server and peer-to-peer. In this section, I will outline the advantages and disadvantages of each in order to reach a decision on which model is most appropriate for my project.

P2P networks are easy to set up and require no central server. I would only have to create one program, the client, which would have the capability to interact with other clients. However, authentication would be virtually non-existent and there would be no hierarchy in case of data corruption. A single client could cause data loss if they decide to remove the program, so every client would have to store all information, leading to privacy issues, data redundancy and unrealistic storage usage. I also fail to see how a client could listen for connections over a WAN, without first sending all other clients their public (and likely dynamic) IP and port forwarding their personal router.

A dedicated server would control authentication securely and maintain data integrity by storing all data. Hence, reliance on clients is all but eliminated: a client does not have to store any information, there are no privacy or data redundancy issues and a client only has to connect to the server, not act as a server itself. Furthermore, should a change be required then the server can be updated without issuing a client update, which every user would have to download before the system could function again. A client-server model would, however, require two programs to be created: the client and the server, and traffic congestion could become an issue should the number of clients increase. Finally, if the server failed, the entire system would be rendered unusable. However, this is why hardware redundancy exists and in a low priority application such as this project, a little downtime should not cause excessive inconvenience.

For my project, I will use the client-server model. Messages will be sent from a client to the server, where they will be stored, available for retrieval by the intended recipient. As the server will only be running on one computer, I will make the client reasonably thick.

### Connections

The language I will use for this project is VB.NET, and upon research, I found suitable classes for this project already exist: TcpListener and TcpClient.

TcpListener listens for connection requests and so would be suitable for the server. However, its documentation states that it waits in blocking synchronous mode, so I predict a listener will have to be run in a separate thread so the main thread can still perform operations. Furthermore, they are bound to a port number so, if I am to use this class, then one listener will be required per connection, and each connection will need to connect to a different port. I am as yet unsure how the client would ‘know’ which port it should connect to, however.

TcpClient allows connections to be attempted and data to be sent and received (enabling two-way communication with the server as required). Similarly to the server, the client must always be listening for data sent from the server. Therefore, to allow the user to perform additional tasks, the method reading from the connection will be run in a separate thread.

### Protocols

Many well-established protocols exist for instant messaging, Pidgin, for example, supports many. XMPP is a very popular solution, which has many features, and servers ready to download and run. However, I believe these would detract from the purpose of this project, which is to understand how these services were developed. I would like to investigate the server-side aspects of an instant messenger beyond merely installing an XMPP server. So, instead, I will simply use the TCP protocol and build my own server from the ground up.

### Threading

VB.NET also has classes to enable multithreaded programming. To accomplish this one merely has to use the Thread class to define the process which should be run in the thread, and then execute the thread. Once processing is finished the thread should close automatically. Each user’s connection to the server could be handled in a separate thread.

### Storage

The server will need to store user authentication details, sent and received messages, and every user’s friends. For the first two aims, a database would be ideal because a large amount of data will be stored, and it needs to be easily queried and manipulated.

For representing how different users are related (by being friends) I first considered the use of a graph database, as recommended to me by Mr Deane. These databases are oriented towards situations where the data is highly connected, and the relationships between data are of equal importance as the data itself. Many social media platforms, including Facebook, use graph databases to store who are friends, who are friends of friends, who are friends of friends of friends, and so on. In a relational database, establishing relationships is computationally expensive, and the problem would only worsen as the number of users increases. In a graph database, these relationships that relational databases struggle to calculate are stored along with the nodes, like edges. Hence, retrieving relationships is no different to retrieving data.

However, the NoSQL nature of graph databases and the, compared to relational databases, complexity and lack of documentation currently available signals that their incorporation into my project may result in a disproportional use of my time. Furthermore, a relational database would suffice in this situation since my instant messenger is not intended for a large volume of users. The design of the relational database will be covered in the design section.

Upon researching encryption (see later section), I have found encrypted data is often stored in binary. This does not present a problem for volatile storage but storing encrypted data in the database will require some changes. The two main data types which support this are “binary” and “varbinary”, with the only difference being that binary is suitable for where all the data is of the same size. The binary data type will always use the allocated number of bytes of storage, even if the length of the data does not require it.

### Defensive Programming

An interesting topic I encountered while researching error handling is defensive design. The aim of this is to write source code in such a way that it is well protected in the case of an unforeseen error in the future. The program should, if at all possible, continue operating at all times. Clearly, this technique would be applicable to my server, as the clients are useless without it. Some common guidelines to achieve defensive design are given below:

* Source code should be high quality, comprehensible and behave predictably
* If bug-free code exists, it should be reused, to reduce the chance of introducing bugs
* User input should be scrutinized

### Encryption

My client specified that the encryption of messages and stored user passwords is a high priority. Some instant messenger providers offer end-to-end encryption, but many also do not enable this by default, suggesting they would prefer it not to be used.

End-to-end encryption is a system whereby only the two users communicating, that is the two instances of the client, can decrypt the messages sent between them. Eavesdroppers, or even the server, should not, in theory, be able to decrypt the messages for the simple reason that they do not have access to the required keys. The server would store the messages as usual, just in ciphertext rather than plaintext. The initial exchange of keys between the clients must be secure and known only to those clients for this system to work. The main advantage of end-to-end encryption is the absolute privacy possible. Data cannot be decrypted, even under the ruling of a court order, and so is often used in critical applications such as whistleblowing, or when the data must be assumed to be unedited.

Most instant messenger providers instead offer encryption only from the client to the server. In principle, this is also secure but raises the possibility of any third party running the server to have access. The client must trust the server to be secure, and the owners of the server to respect their privacy. However, allowing the server to view messages in plaintext offers some advantages. The client can request features such as searching messages from the server, and the server can detect illegal or banned usage.

Two of my client’s requests were admin facilities and a profanity filter. Both of these would require access to the messages sent so I will not be using end-to-end encryption in my project. I will, however, be encrypting messages from the client to the server, and vice versa.

I first considered rolling my own encryption algorithm, however, upon research, it seems this is almost universally frowned upon, with the general consensus being that the result will almost certainly be intrinsically flawed. An open-source solution will have been tested by experts for years, however, and hence can be assumed to be secure.

There are two types of encryption: symmetric and asymmetric. Symmetric algorithms use the same key for encryption and decryption whereas asymmetric algorithms use different keys for encryption and decryption.

With asymmetric encryption, every person owns a public key and a private key. The public key is available to everyone; however, the private key must be only known to that user. A message is encrypted using the public key and sent to another user. While this encrypted messagecan be intercepted, and the interceptor could have the public key, only those holding the private key can decrypt it. The public and private keys are mathematically linked; the public key can be derived from the public key but not vice versa.

With symmetric encryption, a single key is used, which must only be known to the two users communicating. It is much faster than asymmetric encryption but suffers from the key distribution problem.

Pretty Good Privacy (hereafter PGP) is an encryption program which makes use of both of these algorithms to achieve the sending of messages confidentially. PGP follows these steps:

1. The server generates a session key (symmetric), which will eventually be used to encrypt messages between the client and the server. However, the session key needs to be securely sent to the client.
2. The client generates a public and private key pair (asymmetric), which will be used to facilitate the sending of the session key.
3. The client sends the public key, in plaintext, to the server. So, everyone has this key.
4. The server encrypts the session key with the public key. So only the user holding the correct private key can decrypt the session key.
5. The server sends the encrypted session key to the client, who decrypts it using their private key.
6. Both the client and server now have the session key, and so can communicate securely.

PGP uses asymmetric encryption to work around symmetric encryption’s key distribution problem and hence allows for a best of both worlds approach. PGP also includes methods to prevent public key tampering, such as identity certification and their web of trust.

Unfortunately, I have found that to utilise PGP in VB.NET a third-party library must be installed, namely “DidiSoft.Pgp”. My school forbids installation of software, so were I to use this method, I would only be able to work on my project at home. This is unfeasible so I will look into using a library provided with VB.NET. The “Security.Cryptography” library contains multiple encryption classes such as RSA, TripleDES and AES.

RSA is an asymmetric encryption algorithm and is commonly used for the asymmetric portion of PGP. 2048 bit keys are recommended for RSA, however, and RSA is perceived to be very slow.

TripleDES is based on the Data Encryption Standard but performs three times as much encryption. It is a symmetric stream cipher, requires a 24-byte key, and is considered secure.

A block cipher encrypts plaintext in blocks, as opposed to, for example, TripleDES, which encrypts bitwise. Each block is encrypted multiple times using different sub-keys derived from the main key, and each block operation affects the next. The first block does not have a preceding block, so a randomly generated IV (initialisation vector) is used. AES is an example of a block cipher, and is considered one of the most secure algorithms available due to its block size of 128 bits and its support of key size up to 256 bits. Using CBC mode, each block is XORed with the previous block (with the first XORed with the IV). Even with AES-128, and a supercomputer, it would take years to brute force.

For my project I plan to use AES-256 and RSA-2048, using the steps from PGP.

### Hashing

Hashing is similar to encryption in that plaintext is encrypted, however, hashing is a one-way function. The output of a hashing function, known as the message digest, cannot then be decrypted. Another crucial property of a hashing function is that the message digest is unique; two distinct strings will never be hashed to the same output, in theory.

At first glance, this may seem less than useful, however, they are often used for storing user passwords. Even if the server’s database is compromised, the user’s original password, which they may use in other applications, is safe. Typical usage of hashing is as follows:

1. The user enters their plaintext password, which is sent to the server.
2. A hashing algorithm is applied to the password, to output a message digest.
3. The message digest is compared to that stored in the database, from when the user created their account.
4. If they match the server knows that the user entered the correct password, since message digests are unique.
5. If the database is compromised, attackers can only yield the hashes, which cannot be reversed to obtain the plaintext password.

To make the hash harder to crack, for example by running a dictionary attack with a list of common passwords that have been pre hashed, a salt is often used. A salt is random data appended or prepended to the password before applying the hashing algorithm. This salt is then stored alongside the message digest in the database. Salting greatly reduces the effectiveness of dictionary attacks since the ‘new’ plaintext password is now likely much more complex and unlikely to be found in a common list of passwords. Provided the salt is sufficiently large (usually at least as large as the hashing algorithm’s output), a rainbow table (pre-computed table of plaintext and message digest) would have to be unfeasibly large due to the number of possible salts in addition to the possible passwords in the first place!

If a salt is used, typical usage would be as follows:

1. When the user first creates an account and sends the server their password, the server randomly generates a salt, which is combined with the password, hashed and stored.
2. Whenever a user authenticates the salt is retrieved, combined with the password, hashed and compared to that stored in the database.
3. If the database is compromised, and the hash and salt obtained, the attacker still has to generate a new rainbow table with the salt.

Perhaps the most widely vetted and trusted hashing algorithm to date is BCrypt. Unlike other (much faster) hashing algorithms such as SHA-1, BCrypt is very slow and can be configured to take even longer. The advantage of this is the significant setback to brute force attacks, as hashing each password is much more time consuming than usual. However, similarly to PGP, a third-party library must be installed, so instead I will use SHA-512 with a 64 byte (64\*8=512 bits) randomly generated salt from the RNGCryptoServiceProvider class.

### Group Chats

Group chats allow one user to communicate with several other users at once. Instead of the user selecting a partner they wish to communicate with, they select a group or conversation. From the user’s perspective, a temporary chat room has been created alongside their individual conversations. The server will have to store these messages separately, so that group messages do not ‘leak’ into private messages.

## Data Volume

I would estimate that anywhere up to 500 users could have accounts at any given time. The number of active users at any one time would be much smaller however, likely less than 100. Working off the 500 figure, and some very rough approximations, I highly doubt the database would ever surpass 1GB in size. Therefore, I see the use of a database as feasible for this project.

## Objectives

These objectives are based mainly on Mr Deane’s desired aims for the project, but also the research I have conducted on similar services.

### Server-side

* Accept connections from clients
* Verify authentication details by consulting the database
* Facility to create a new user
* ‘Distribute’ client connection to a free port
* Keep track of free/used ports
* Receive ASCII messages from clients
* Forward to the relevant partner
* Store messages in a database
* Allow users to have ‘friends’ and search for potential friends
* Handle multiple connections at once
* Recommending friends to the user
* Admin facilities: monitor messages, reset passwords, approve/reject friend requests, remove users
* Encryption and hashing

### Client-side

* Connect to server
* Supply username and password to the server
* Send ASCII messages to the server
* Choose desired ‘partner’ from list
* Ability to see if users are online
* Retrieve all messages in the current conversation
* Search for friends and send friend requests (to be approved by the admin account)
* Simple user interface
* Profanity detection
* Encryption
* Group messaging

# Design

## Overview

To reiterate, my project is an academia-oriented messaging service that allows communication between two users over a WAN. The whole service can be broken down into two separate solutions:

1. The client-side program capable of connecting back to the server, and providing the interface for all user functionality. This includes logging in, sending messages, viewing conversations, adding friends and encrypting transmissions.
2. The server-side program which services all the aforementioned user requests and deals with maintaining data integrity and managing multiple live connections simultaneously.

The administrative facilities are included in the client program, but will be covered later.

The server will store messages, friends and user details in a relational database. User passwords will be hashed with SHA512 and stored along with their respective salts. Transmission over the internet will use AES-256, with the exception of the initial key exchange, which will be secured with RSA-2048. The typical order of operations for a user, and the corresponding actions of the server, could be as follows:

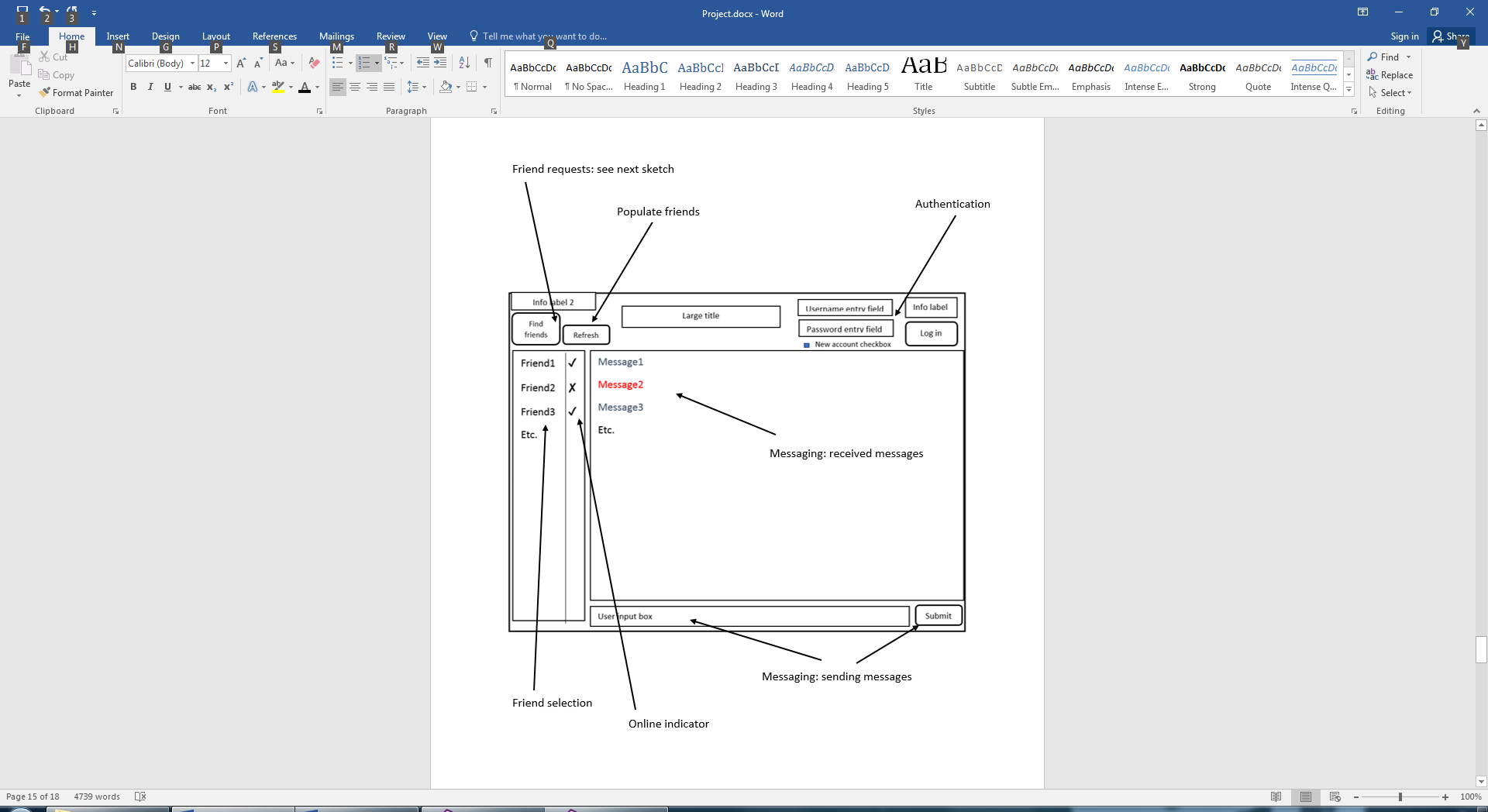
1. Key exchange and port distribution
2. The server will have one port which is nearly always open and waiting for a connection.
3. Every user will first connect to this port, where they will establish the keys to be used in the session’s encryption. From here on, every communication between the server and client will be encrypted. The client will then undergo authentication and, if authentication is successful, be ‘told’ of the port they should connect to for the rest of their session. The client then disconnects from this port.
4. Key exchange will follow the steps of PGP, which was covered in the analysis section.
5. Authentication
6. If the user does not have an account, they create an account by sending a username and password to the server. The server would then check if the username is taken. If it is, a suitable response is sent to the client, which informs the user. If the username is not taken, the user’s password is salted, hashed and all of this information is inserted as a record into the database. The user will automatically be friended with the admin account in the database (covered later) and logged in to their new account.
7. If the user has an account, their username is checked in the database. If it exists, the relevant salt is retrieved, added to the supplied password and hashed. This is then compared to the stored hash to verify the user. If the username is not found, or the hashes do not match, then the client is sent an appropriate message. Otherwise, the user is logged in.
8. Connection management
9. Once the client is informed of their designated port, the server will start a new listener on this port. A connection will only be accepted on this port from the same IP address as the device which authenticated earlier. The management of this connection will be achieved via OOP, as one new instance of a ‘master’ class, which owns other classes and will completely service the client. This new master class instance will be saved as a value in a dictionary, with the key as the username of the user. The initialisation of the master class (to start servicing the client) will be called in a separate thread, so as to allow the server to manage multiple connections at once.
10. The server now awaits commands from the client and has also begun waiting again for connections on the main port, so as to allow other clients to log in.
11. Populate friends
12. This service supplies the client with a list of their friends along with whether each is online or not. The client sends this command automatically when the user first logs in.
13. The server queries the database for all friends of the user, checks whether each is in the dictionary (to determine whether each is online), and sends the result to the user.
14. The client formats these results and displays the information to the user via a list.
15. Friend Selection
16. The client must be able to select the friend they wish to message or view previous conversations with. Once the friends list has been populated, clicking on any friend should update the screen with previous messages, and allow messages to be sent to that user.
17. When the user clicks on a friend, a command is sent to the server with the friend’s username. The server then queries the database for all messages between the user and their requested friend and sends these to the client.
18. The client then colour codes and displays these messages
19. Friend requests
20. An auxiliary form allows the user to search for other users, and send a friend request. A simple entry allows a username to be entered, and a button sends this to the server.
21. The server will then query the database for usernames similar to the one supplied and will return a list of these to the client.
22. The client will then filter out those results who are already friended with the user (done to reduce the load on the server), and display the rest in a list. The user can then click on their desired friend to send a friend request to the server.
23. The server will acknowledge the sent friend request by recording the two usernames in a friends table in the database but set an accepted field to false, to signify the friend request not being as yet accepted by the admin account (see part 8).
24. Messaging
25. Once the user has selected a friend, as detailed in part 5, they can communicate by entering text into an entry, and either clicking a button or pressing the enter key. The client will then send the message to the server.
26. The server will check the dictionary to see if the recipient is online by consulting the dictionary of connection. If so then the message will be sent to that user by accessing the necessary master class instance through the dictionary with the key of the recipient’s username. Regardless of whether the user is online, the database will be updated with the message.
27. Admin facilities
28. When the admin account logs in, the server should recognise the user and return a message signalling the client application to allow access to an additional form. This is more secure than hardcoding the client to check if the user is the admin, since all users have access to the client application, but not the server application.
29. This admin form will replace the friend requests form, as the admin account will not require this functionality, since his account is automatically friended with all users.
30. The form will contain a list of all users, with options to change passwords, view messages and delete users. There will also be a list of pending friend requests, which can be accepted or rejected. If accepted, the accepted attribute in the database should be set to true. If rejected, the record containing the friend request should be deleted.

Each of the client requests that can be made to the server will be identified with a unique string which prepends any information supplied with the request.

## User Interface

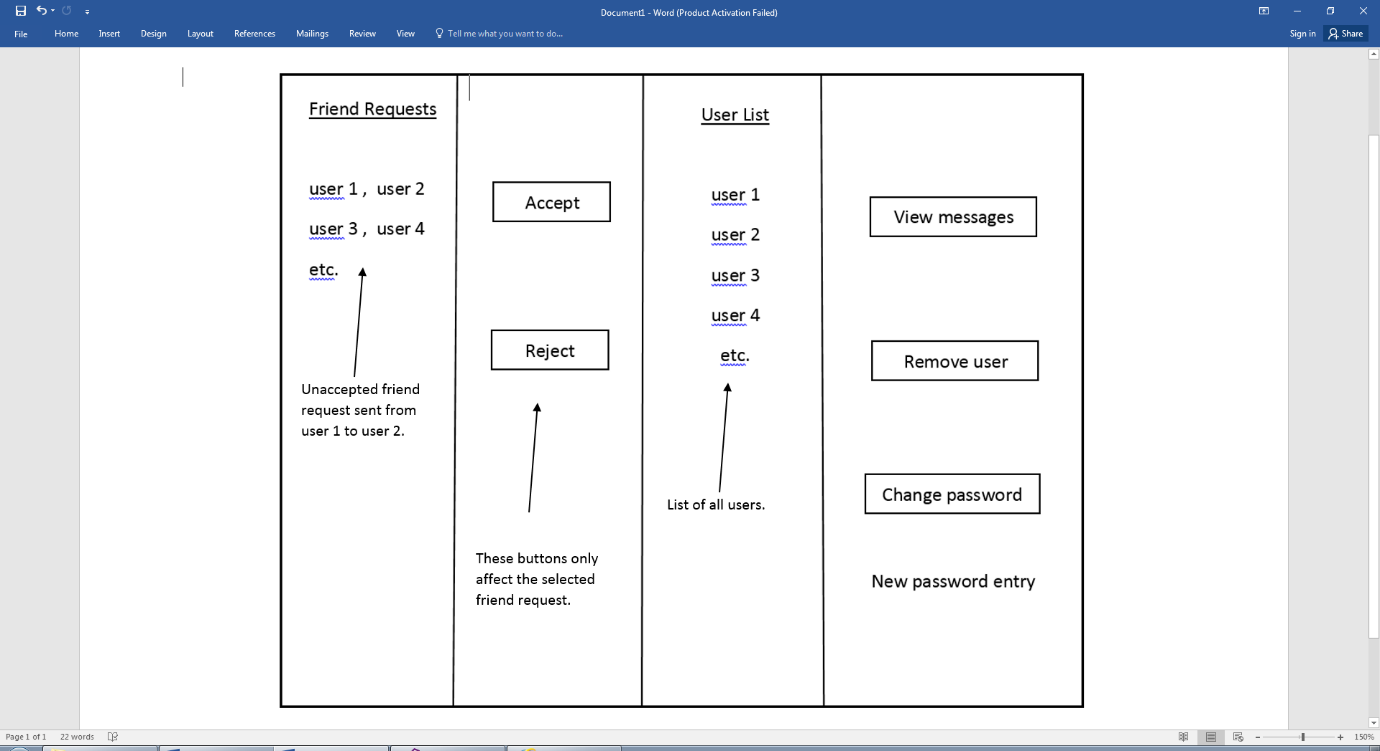
On the following pages are initial sketches of what the client’s user interface may look like. The user interface was not a high priority for this project, but I have aimed to make most of the required functionality easily available on the main form, and the less commonly used functions on a separate form. The corresponding functions to each of the buttons are briefly covered below also, and follow the same naming convention set out in the overview. Therefore, for a high-level view of a particular element on a form’s function, the relevant section of the overview can be easily referenced from the sketches below.

### Main Form



### Friend Requests Form

### Admin Form



## Database Design

The server requires a database to store user details, messages, and relationships between users such as friend requests. The database will be in Third Normal Form.

### Entity - tblUsers

tblUsers(UserID, Username, PasswordHash, Salt)

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Data type** | **Size** | **Information** |
| UserID | Integer = INT | 4 bytes | Primary key. Auto increments each time a user is added. |
| Username | String = VARCHAR | 25 characters = 25 bytes | Identifies a user. |
| PasswordHash | Binary = BINARY | 64 bytes | The user’s password hashed with SHA512. |
| Salt | Binary = BINARY | 64 bytes | The randomly generated salt appended to the user’s password before hashing. |

### Entity - tblMessages

tblMessages(MessageID, Text, UserID, FriendID)

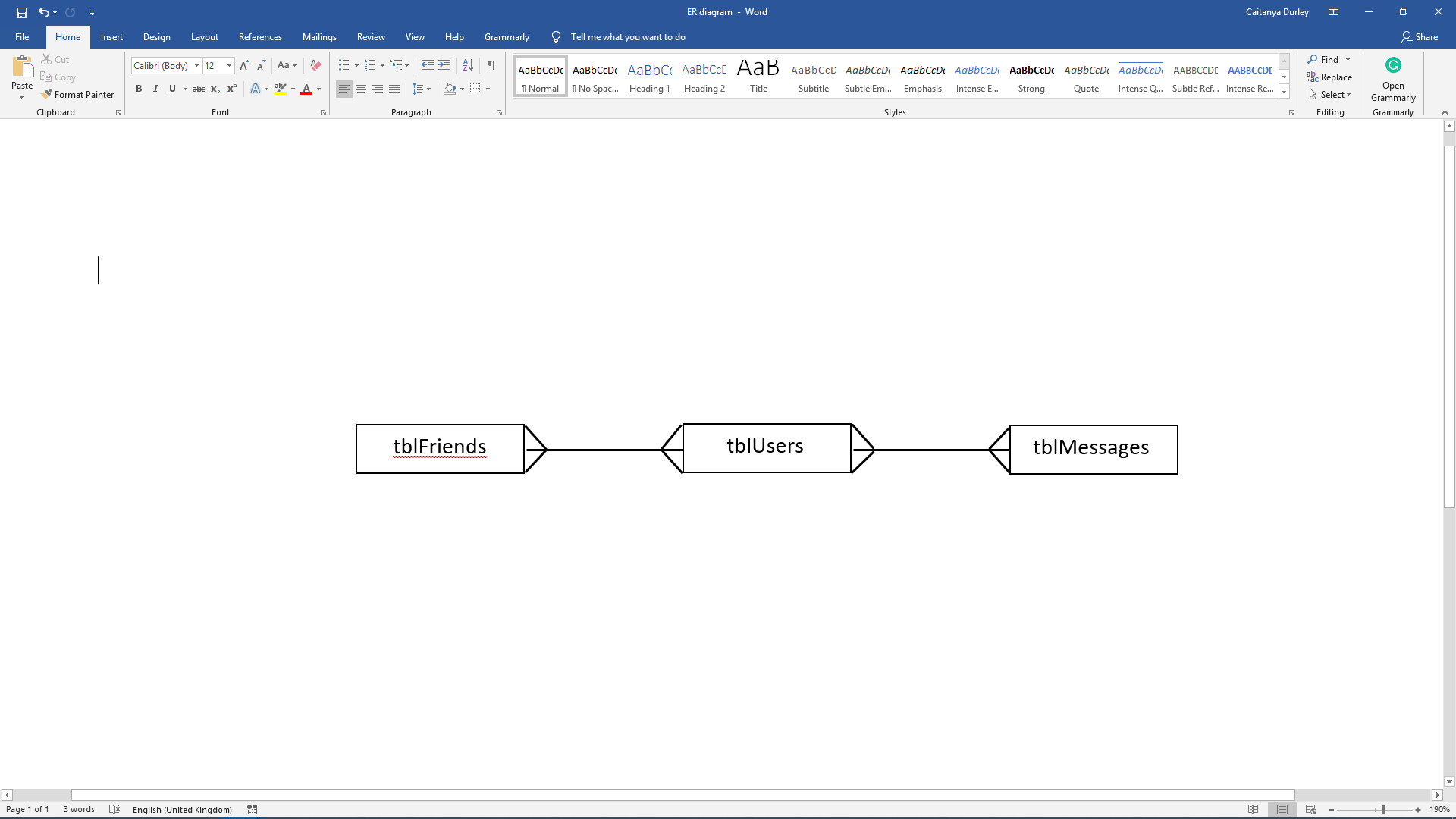
|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Data type** | **Size** | **Information** |
| MessageID | Integer = INT | 4 bytes | Primary key. Auto increments each time a message is added. |
| Text | String = VARCHAR | 980 characters = 980 bytes | The string of text sent from one user to another. |
| UserID | Integer = INT | 4 bytes | Foreign key from tblUsers, references UserID. Holds the user ID of the user which sent the message. |
| FriendID | Integer = INT | 4 bytes | Foreign key from tblUsers, references UserID. Holds the user ID of the user who should receive the message. |

### Entity - tblFriends

tblFriends(UserID, FriendID, Accepted)

|  |  |  |  |
| --- | --- | --- | --- |
| **Attribute** | **Data type** | **Size** | **Information** |
| UserID | Integer = INT | 4 bytes | Primary key. Foreign key from tblUsers, references UserID. Holds the user ID of the user who sent the friend request. |
| FriendID | Integer = INT | 4 bytes | Primary key. Foreign key from tblUsers, references UserID. Holds the user ID of the user who is the recipient of the friend request. |
| Accepted | Boolean = BIT | 1 bit | Either true or false to signify whether the friend request has been approved by the admin account. Default value of false (0). |

### Entity-relationship diagram



One user can have many messages, and one message belongs to exactly two users. Hence the relationship between tblUsers and tblMessages is many-to-many. Similarly, one user can have many friends, and one record in tblFriends belongs to exactly two users. Therefore, the relationship between tblUsers and tblFriends is also many-to-many.

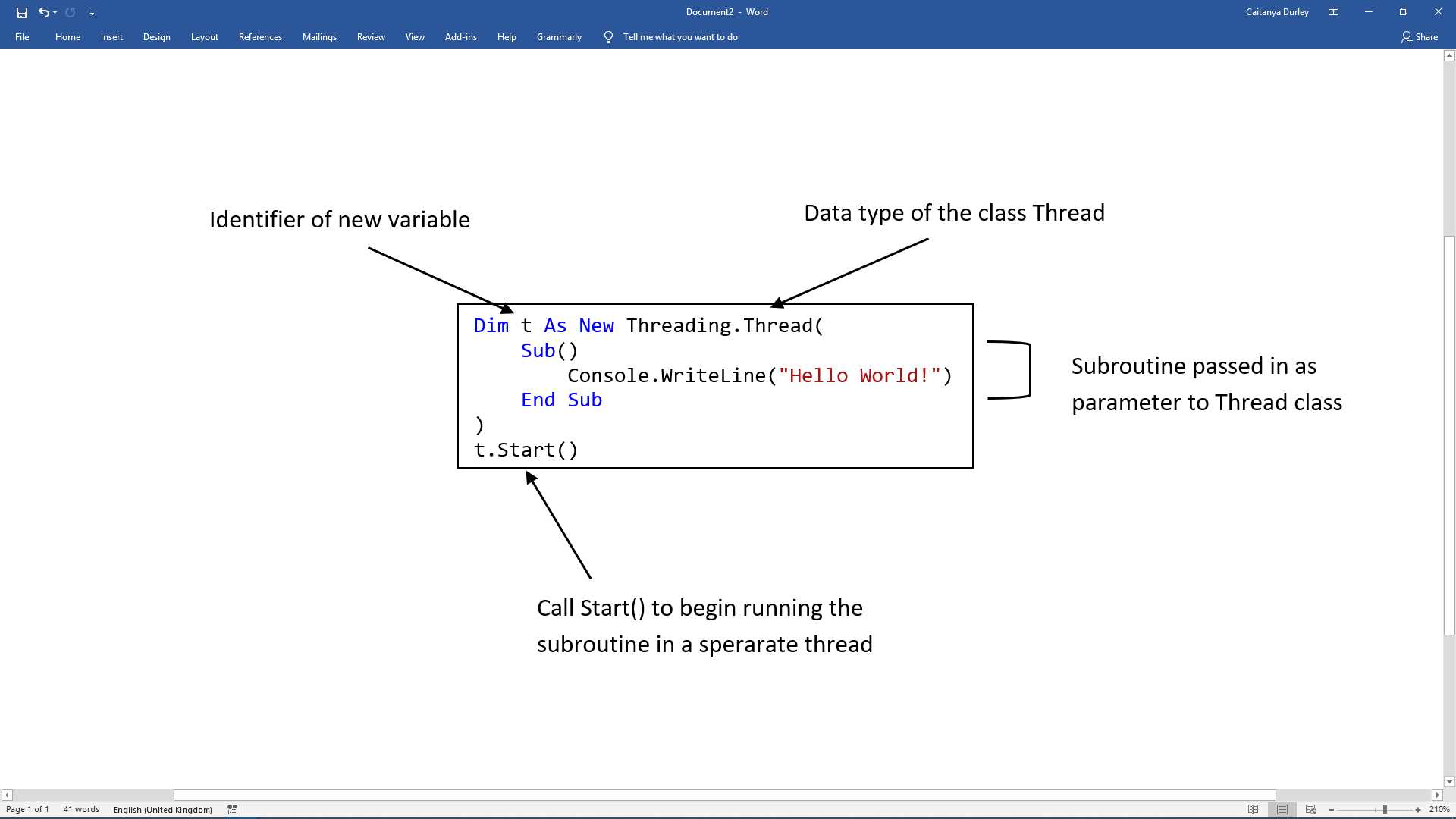
## Connection

As noted in the analysis section, VB.NET has two classes which can be easily used to establish a connection over the internet: TcpListener and TcpClient.

A TcpClient can be started by creating a new instance of the class with the server’s IP address and open port as parameters to the New method. Once the client connects, a network stream can be obtained from the connection for reading and writing by calling the GetStream method. To read from the stream simply call the Read method and to write to the stream simply call the Write method.

A TcpListener can be started by creating a new instance of the class and calling the Start method. Once the listener receives a connection, this can be interacted with through a TcpClient class and a network stream as before, or alternatively, through a socket. Since I am using a stream for the client-side application I will use sockets for the server-side application for variety. To obtain the socket from the TcpListener simply call the AcceptSocket method.

## Threading

In VB.NET, a new thread can be created by defining a new instance of the Thread class with the desired method to be run (i.e a method) as the parameter to the New method. To start the thread, call the Start method in the Thread class. This is rather difficult to explain, and harder still to write as pseudocode, but should become much clearer with the following example, written in VB.NET:

## Encryption

As detailed in the analysis section, I will be using AES-256 and RSA-2048 for the symmetric and asymmetric portions of the PGP algorithm respectively. To use either of these, the “Security.Cryptography” library must first be imported. To recap from the analysis, the server will generate an AES symmetric key and the client will generate a RSA public/private key pair. The client sends the public key in plaintext to the server. The server encrypts the symmetric key with the client’s public key and sends this to the client. The client decrypts the symmetric key with the private key and now they can communicate with the fast symmetric key.

### RSA

The RSACryptoServiceProvider class included in the aforementioned library makes the use of RSA quite simple. Firstly, create a new instance of the RSACryptoServiceProvider class. In order to use a key size of 2048 bits, this should be passed in as a parameter to the New method of RSACryptoServiceProvider.

To encrypt a string, convert the string to a byte array using ASCII. Then call the Encrypt function on this byte array from the RSACryptoServiceProvider class. The function should return an encrypted byte array.

To decrypt a byte array, call the Decrypt function on this byte array from the RSACryptoServiceProvider class. The function should return a plaintext byte array, which can be converted back to a string using ASCII.

### AES

Encryption using AES is somewhat more involved. Similarly to RSA, we begin by creating a new instance of the AesCryptoServiceProvider class. AES uses a key size of 2048 bits (which equals 256 bytes) and mode CBC by default. However, we will also require a memory stream, CryptoStream and ICryptoTransform to use AES in VB.NET. The ICryptoTransform class “does” the encryption or decryption, the CryptoStream is a .NET standard in cryptography and the memory stream is a prerequisite for the crypto stream.

To encrypt a string, convert the string to a byte array using ASCII. Then create the encryptor object by calling the CreateEncryptor function in the AesCryptoServiceProvider class and assigning its return to the ICryptoTransform. Create the memory stream and the CryptoStream objects. Finally, to encrypt our byte array, call the Write method in the CryptoStream. The encrypted output will be written to the memory stream, which can then be read out.

To decrypt a string, create the decryptor object by calling the CreateDecryptor function in the AesCryptoServiceProvider class and assigning its return to the ICryptoTransform. Create the memory stream and the CryptoStream objects. Finally, to decrypt our byte array, call the Write method in the CryptoStream. The decrypted output will be written to the memory stream, which can then be read out and converted back to a string using ASCII.

## Hashing

User passwords will be salted and hashed before storing in the database. I will use SHA-512 with a 64-byte salt for this.

To generate a random salt, we can use the RNGCryptoServiceProvider class included in the cryptography library. Create a new instance of the class and call the GetBytes method to fill a pre-defined byte array with random bytes.

To hash a string, convert the string to a byte array and append the salt. Create a new instance of the SHA512Managed class and call the ComputeHash method on the byte array to return the message digest.

## Noteworthy Data Structures

### Port Queue

The server accepts a connection on one port per client, and must keep track of which ports are currently in use. This could be achieved through a global integer queue of port numbers. The queue could be populated with a simple for loop at the beginning of runtime, which would dictate the maximum number of connections the server could maintain.

When the server authenticates a client, a port can be dequeued and sent to the client. When the user disconnects, the thread which initialised the master class can enqueue the now available port before closing.

### Master Class Dictionary

As mentioned in the overview, a connection with a client is completely managed through one instance of a master class and this instance will be stored as the value in a dictionary. Firstly, this allows easy access to and organisation of connections to client. Secondly, and perhaps more importantly, it provides a way to forward messages from one client to another.

Say one user sends a message to the server intended for their friend. How should the server forward the message? If the server had to wait for the friend to request messages from the user again, then the client would have to repeatedly check for updates. Instead, the user’s master class can access the dictionary and, using the friend’s username, get the friend’s instance of the master class and use that class to send a message to the client.

As with the port queue, the thread which initialised the master class can remove the user’s instance of the master class before closing.

## OOP Overview

Below are lists of classes to be used in both applications, specifically the client application and specifically the server application respectively.

### Common classes

1. AES – Used for symmetric encryption.
2. RSA – Used for asymmetric encryption.
3. distAndAuth – Used for key exchange, port distribution and authentication.
4. transmission – Used for the main communication between the server and the client.

### Client-specific

1. Form1 – Used for interacting with the main user interface.
2. findFriendsForm – Used for interacting with the auxiliary find friends form.
3. adminForm – Used for interacting with the admin form which contains the admin controls set out in the objectives.

### Server-specific

1. instances – Used as a master class as detailed in part 3a of the overview.
2. SHA – Used for hashing.
3. databaseIO – used for interacting with the database.

### Server UML

G:\Visual Studio 2015\Projects\IMServer\Server UML.png

Please note this UML omits some details, such as the numerous string constants present in transmission, as these do not add to overall understanding and merely obfuscate the purpose of the class.

### Client UML

G:\Visual Studio 2015\Projects\IMClient\Client UML.png

Please note this UML omits some detail, such as numerous methods in Form1, whose only purpose is to aid the user to interact with the UI.

## Class Design

The aim of this section is to explain the decisions represented by the class diagrams above. Each class will be taken in turn, and its fields and methods will be justified.

### AES and RSA

These classes have only two methods: encrypt and decrypt. Encrypt should take a plaintext string and return an encrypted byte array. Decrypt should take an encrypted byte array and return a plain text string. Both the client and server need these classes because anything one encrypts has to be decrypted by the other.

### distAndAuth

#### Server

Multiple fields are required by this class: socket and tcpListener to handle the connection, port to store the port the client will connect to, user to store the client’s username and databaseIO to interact with the database.

There is one public function getPort, which should be called by Main, passing in RSA and AES. Since RSA is only needed for initial setup, this can be passed by value, however, AES will be needed for the duration of the session and so will need to be passed by reference. The function should return the port the client will connect to and the IP address of the client, so that a listener can be set up. Of course, as part of this encryption will need to be established and authentication undertaken, so two other private methods will be needed: auth and getPublicKey.

Starting with getPort, a listener will be created on the distribute port, which all clients must connect to. getPublicKey will be called, which will receive the client’s public key and setup the RSA class parameter with this. Then the server generated AES credentials (key and IV) will be sent to the client, but encrypted with RSA. Therefore, as discussed before, both the client and server now have symmetric encryption in place. All communications from this point onwards are encrypted with AES, even though I may not explicitly state so.

Now auth can be called, which will receive the user’s credentials, and whether a new account should be created or not. auth will call one of two relevant functions in databaseIO, corresponding to whether a new account should be created, or credentials checked. auth will return the username of the user if authentication/creation was successful, and “NULL” otherwise. getPort will dequeue the first element out of a global queue of ports, and send this to the user, if authentication was successful and the server has not exceeded its maximum number of allowed connections. Otherwise, if auth returned “NULL”, “0000” will be sent, and if authentication was successful but the server cannot accept any more connections “0001” will be sent. The client knows how to interpret these special cases.

The connection with the user can now be closed, and getPort can return the username, port and IP address to Main().

#### Client

The process is similar for the client. There are three methods, following the same naming convention as the server. getPort is called by Form1 when the user clicks the log in button. getPort uses tcpClient to connect to the server’s distribute port and sets up a network stream.

Next, sendPublicKey will be called, which sends the RSA public key in plaintext to the server. The client will read in the server’s AES symmetric key and IV, and decrypt them using RSA. Then AES will be assigned these values, and all communications here forward are encrypted. auth is now called, which sends the user entered credentials to the server. The credentials could be sent in the format “username:password:newAccount”, where newAccount is a Boolean value indicating whether the server should create a new account or not.

Processing returns to getPort, which will now read in the port supplied by the server. Once again these can be “0000”, “0001” or a valid port number. getPort will inspect the port and return 0, 1 or the port number respectively to Form1. Form1 can interpret these results.

### transmission

#### Server

transmission has fields similar to that of distAndAuth. Socket and tcpListener for communication, aes for encryption, databaseIO for interacting with the database, and statusCode, which is used when the user needs to be disconnected. There are also multiple string constants relating to the available commands the client can send.

Four methods will be needed: communicate for the majority of interaction with the client; New to setup AES, tcpListener and socket; logOffSub to disconnect users; and userOnlineRead to send messages between two users to the client.

New is of course the method called upon creating a new instance of the transmission class. This will be done from the instance of the master class running in a thread. Since New sets up the connection, it will need the port number, the AES instance from distAndAuth and the client’s IP address passed in as parameters. The parameter AES will be copied into the field, while port will be used to define the tcpListener and socket fields. If the IP address of the connecting device does not match that stored in ipAddress, then socket will not be defined, causing an exception to be raised later in communicate. Note that, since socket is not defined, it doesn’t need closing, so logOffSub will not need to be executed. This will be handles by the use of statusCode (see later section).

communicate is, at its core, a large loop containing multiple if statements. I should also point out that the very vast majority of runtime will be spent here. My service follows the client-server model, that is, clients request a service from the server and the server obliges. Hence, the server will spend most of its time waiting for a request for the client. An endless loop is created which first reads in what the client sends and, since the user interacts with the server through a client application, the format of the text sent is known to the server. If the read fails, then the user has likely disconnected and the loop is exited. Next, a large elseif statement should check whether the string received is intended to request one of many functions. I will list out the various requests a normal user could make, and how these are uniquely identified to the server:

1. Log off – "!$%IM\_LOGOFF^&\*"
2. Change partner (and get messages between the user and this partner) - "!$%IM\_PARTNER^&\*" + the username of the requested partner
3. Get the list of friends of the user - "!$%IM\_FRIENDSPOPULATE^&\*"
4. Get a list of usernames like the one supplied - "!$%IM\_SEARCHFRIENDS^&\*" + the username
5. Send a friend request - "!$%IM\_FRIENDREQUEST^&\*" + the username
6. Send a message - "!$%IM\_MESSAGE^&\*" + the message
7. Recommend friends (auto sent by client upon opening findFriendsForm) - "!$%IM\_RCMDFRIENDS^&\*"

The server’s response to these requests follow the same convention, so if a user requested to change partner, the server’s response would be "!$%IM\_PARTNER^&\*" + the messages between them.

The server can send two additional commands: "!$%IM\_LOGOFFFORCE^&\*", which tells the client to log the user out; and "!$%IM\_ADMIN^&\*", which tells the client that the user is admin, and should allow access to the admin form.

If the user is an admin, the following commands can be sent to the server:

1. Get messages between two users - "!$%IM\_GETMESSAGE^&\*" + the usernames delimited by commas
2. Get a list of all users registered - "!$%IM\_GETUSERS^&\*"
3. Get unapproved friend requests - "!$%IM\_GETFRNDRQST^&\*"
4. Accept a friend request - "!$%IM\_ACPTFRNDRQST^&\*" + the usernames delimited by commas
5. Reject a friend request - "!$%IM\_RJCTFRNDRQST^&\*" + the usernames delimited by commas
6. Change a user’s password - "!$%IM\_CHNGPSSWD^&\*" + the username and password delimited by commas
7. Delete user - "!$%IM\_DELUSER^&\*" + the username

These strings will be stored as constants.

Whichever command the user has requested is then executed, usually by calling the relevant routine from databaseIO. The only two commands which require any significant processing (outside of databaseIO) are when the user changes partner and needs past messages or when the admin requests messages between two users; in both cases this is handled by the local private method userOnlineRead.

userOnlineRead is a method that sends messages to the client. The volume of data sent at one time is limited to 1 kilobyte, or 1024 bytes, for optimisation. Therefore, it is unfeasible to simply send a string of concatenated messages, as this would grossly exceed the limit. userOnlineRead fits as many messages as will fit into a string that will be encrypted into a 1024 byte array, and sends it. This process is then repeated with the remaining messages, until all messages are sent.

logOffSub closes the connection and possibly sends the log off string to the user. It is called from Main, if a user tries to connect from two devices, and just before execution returns from the master class. Transmission’s status code field lets logOffSub know whether: it has been called before/it doesn’t need running and if two devices are trying to connect on the same account. The default value of statusCode is 0. In the case of two devices connecting to the same account, Main will set the value of statusCode to 2, before calling logOffSub. If logOffSub is called at any point, statusCode is set to 1. In New, if the connecting device does not have the correct IP address, statusCode is set to 1. logOffSub will only send the log off string to the client if:

1. logOffSub has not been called before
2. It needs to

logOffSub will only close the connection if:

1. logOffSub has not been called before/it doesn’t need running

#### Client

The client-side version of transmission is quite similar to the server-side. Its fields include tcpClient and a network stream for communication with the server, aes for encryption, and various string and Boolean variables to contain data, which will be read out from different classes.

Four methods will be needed: streamRead for getting data from the server; New to setup AES, tcpClient and stream; shutdown to disconnect from the server; and streamWrite to send messages to the server.

New is called upon creating a new instance of the transmission class. This will be done from Form1 once distAndAuth has returned a port, and will be started in a new thread. Since New sets up the connection, it will need the port number and the AES instance from distAndAuth passed in as parameters. The parameter AES will be copied into the field, while port will be used to define the tcpListener and stream fields. If an error occurs here, an error message will show and the client application will exit.

streamRead could be considered what is to the client as communicate is to the server: a large loop containing multiple if statements. An endless loop is created which first reads in what the server sends. If the read fails, then the loop is exited. Next, a large elseif statement should check whether the string received is intended to be the response to one of many commands. As mentioned before, the format of the strings received is identical to that laid out in the server section.

The fields friendsPopString, which is a string, searchFriendsString, which is a string, and admin, which is a Boolean, are used to hold the result from this read. If the response is a list of friends, this is passed into friendsPopString. If the response is a list of usernames, from a search friends request, this is passed into searchFriendsString. If the response is to inform the client that the user is admin, then the admin variable is set to true. If the response is a message received, or a list of messages from userOnlineRead from the server, then a method in Form1 will be called to handle inserting the messages into the UI.

streamWrite is a simple method which takes a message as a parameter, encrypts it and sends to the server.

shutdown sends the log off string to the server and closes the connection. It is called from Form1, when the user clicks the log off button.

### instances

instances is the server’s master class. When a client connects to the server and is authorised, the rest of their session is handled by a single instance of this master class. The instances are kept track of through a dictionary with key being the username and value being the instance of instances.

instances has only one field: transmission. This is an instance of the transmission class detailed earlier.

instances has no method called upon defining it (usually New) because it must be set as the value in the dictionary before being started in a new thread. Instead, there is one method, initialise, which takes the port number, username, IP address and instance of AES (from distAndAuth) as parameters. This initialise method should be called from the new thread. initialise will assign the value of transmission and call the communicate method in transmission. instances will also pass in the required parameters for transmission. While transmission’s communicate method runs, execution will hang on the line which calls communicate in instances’ initialise. Therefore, the following code in initialise will only be run once communicate returns, which is when the user disconnects. The only following code required is transmission’s logOffSub which should be called to close the connection.

### SHA

As mentioned in the analysis, SHA-512 will be used for hashing user’s passwords before storing them in the database. An instance of SHA512Managed, which is available through the System.Security.Cryptography library, is the main field in this class, and “does” the actual hashing. RNGCryptoServiceProvider is the only other field, and allows easy generation of a salt, which will be 512 bits or 64 bytes.

Obviously, a method, hash, will be needed which takes a password and possibly a salt and returns a hashed byte array. The password will need to be converted to a byte array, in the same way as would be done before encryption. Then, if a salt is provided, it should be appended to the password byte array. If a salt is not provided, it should be assumed the user is new, and a new salt should be randomly generated. Finally, compute the hash and return it.

Another method, compare, could take a supplied password, salt and previously computed hash and determine whether the password is correct. It can use the method hash, with the password and salt, to compute the corresponding hash and compare it to the supplied hash. If they are equal, the password is correct and compare can return true, else return false.

### databaseIO

Interacting with a relational SQL database is a crucial part of this project. It is needed for authentication, friend relations and storing messages.

VB.NET has a SqlClient namespace which contains many useful classes for connecting to a SQL server. SqlConnection, SqlCommand and SqlDataReader are among these, and will be stored as fields. SqlConnection represents a connection to the database and is equivalent to a network connection were the database stored on another server. SqlCommand represents a SQL command to be executed through SqlConnection. SqlDataReader provides a way to read the results from SqlCommand. It can be accessed like an array, with each element representing a record returned from the database. Finally, databaseIO will need an instance of SHA, as the authentication undertaken in distAndAuth is farmed out to databaseIO.

databaseIO will have a large number of methods: around one per client request. The algorithms behind each SQL command will be explained in the algorithms section, so here I will detail the general way in which a SQL command would be made and the result interpreted.

When a new instance of databaseIO is created, New will be called which will setup the connection. This is achieved by simply assigning new instances of the classes mentioned above to the fields. The path to the database will need to be passed to SqlConnection and a few parameters will need to be created. Most commands will be a simple string, however, if, for example, a user’s password needs changing, the new hash cannot be included in the command string, because the hash is binary. Therefore, parameters are created for the attributes in the database whose data type is not integer or string, namely PasswordHash and Salt, both of which are BINARY or a byte array.

One other method worth mentioning here is getUserID. This will take a username as a parameter and return the corresponding user ID. This is useful because all tables other than tblUsers do not have a username field.

Every method will begin by opening the connection through SqlConnection. I have chosen this over leaving the connection open for the following reasons:

1. There could be very large time intervals between requests to the database
2. If one connection was maintained per user, there could potentially be a large number of active connections at one time which could lead to concurrent access and performance issues.

After opening the connection, the SQL query to be executed will be assigned to SqlCommand’s CommandText field, as a string. Calling the method ExecuteReader will execute the query and the result will be assigned to SqlDataReader. At this point it is a simple matter of iterating through each record returned, completing any processing required, and closing the connection.

### Form1

Form1 is the class which owns the UI and interacts with the user. Important fields include an instance of transmission, the user’s username, the username of the user they are currently talking to and a Boolean variable indicating whether the user is an admin or not.

By and large, due to the event driven nature of a UI, there will be approximately one method per button. For example, when the user clicks the log in button, the method logIn\_Click should be called, which can then call distAndAuth’s getPort and create an instance of transmission in a new thread.

|  |  |  |
| --- | --- | --- |
| **Method** | **Event** | **Purpose** |
| addItemsListView | Called from transmission | Adds messages to the UI. |
| findFriendsBtn\_Click | Button click | Launches findFriendsForm or adminForm if user is admin. |
| friendList\_SelectedIndexChanged | Select a different friend | Tells the server the username of the user’s new conversation partner. |
| friendsPopulate | Button click, upon logging in | Gets the user’s list of friends from the server and displays them. |
| logIn\_Click | Button click | Logs the user in, or if already logged in, logs them out. |
| profanityDetector | When sending a message | Checks a string for profanities against an array. |
| Submit\_Click | Button click | Sends a message to the server. |

### findFriendsForm

This form allows users to search for friends by username and send friend requests. It also gets a list of recommended friends from the server automatically upon opening.

|  |  |  |
| --- | --- | --- |
| **Method** | **Event** | **Purpose** |
| fillFriends | Called from other methods | Gets a list of usernames from Form1’s transmission’s search friends string and displays them. |
| recommendFriends | Upon start | Asks server to send a list of recommended friends for the user. Calls fillFriends to display them. |
| friendList\_SelectedIndexChanged | Select a different friend | Asks the server to send a friend request to the selected user. |
| friendSearchBtn\_Click | Button click | Asks server to send a list of usernames similar to the one provided. Calls fillFriends to display them. |

### adminForm

This class contains the admin command identifier strings as fields so that admin commands can be sent to the server.

|  |  |  |
| --- | --- | --- |
| **Method** | **Event** | **Purpose** |
| getUsersAndFriendRequests | Upon start | Asks server for a list of users and unapproved friend requests and displays them. |
| requestAccept and requestReject | Button clicks | Tells the server to either accept or reject a friend request depending on which button clicked. |
| viewMsgBtn\_Click | Button click | Asks the server for messages between the selected users. Closes admin form so messages can be displayed in Form1. |
| passChangeBtn\_Click | Button click | Sends server the username of a user and their new password. |
| delUserBtn\_Click | Button click | Asks server to delete the selected account. |

The client-side forms have not been explained in as much detail due to their mostly simplistic nature of sending a request and waiting for a response. The more complex methods shown here, such as addItemsListView, as well as an explanation of the general principle on which most of these methods operate will be discussed in the algorithms section.

## Non-class code

### Server – Main

This is the first code to be executed server-side, and is quite vital. Main will populate a global queue of port numbers and create a new instance of distAndAuth.

Then a loop will be created which creates a new instance of RSA and AES and passes these to distAndAuth’s getPort. Once getPort returns the username, port and IP address of a user, Main will check if the user is already connected, and if so disconnect them. This is done so that if a client suddenly disconnects without the server noticing, then the user is not permanently locked out. A global dictionary containing the connections will have the current connection added, with username as the key and the new instance of the master class as the value, as discussed in the overview. The new connection will be started in a new thread by calling a private method newThread, which will call the initialise method in the master class, to begin processing. This thread will also add the port number back to the queue when the client disconnects (i.e. when the master class returns execution).

The loop will now repeat, recreating RSA and AES (so every client doesn’t have the same key!) and calling getPort again. This process is very quick and ensures it is unlikely that a client connects while getPort isn’t listening on the distribute port.

### Client – ConstantsModule

The client-side application involves many constants that, unlike on the server, are needed by multiple classes and forms. For example, the IP address of the server is needed by distAndAuth for initial authentication but also by transmission for the rest of the session and some of the command identifier strings are needed in findFriendsForm and adminForm as well as Form1. Therefore, these constants have been defined in a module.

A module does not allow inheritance or instantiation (creating an instance of it), like a class would. Its fields are global and its namespace is imported by default so it is perfect for storing constants.

## Group messaging

Some adjustments will need to be made to the standard model of this messenger to implement group messaging.

To access the group, the group would need to be a ‘friend’ of every user in the group. If the group was stored in tblFriends then messages from the group can be retrieved in a similar way to messages between two users. Instead of getting messages from tblMessages between User1 and User2, for example, the server would need to get any message between the group and anyone.

When a user messages the group, the partner stored in tblMessages should be the group name. Therefore, sending messages to a group chat is almost identical to sending a message to another user.

## Server Algorithms

### Main

This flowchart summarises the key functions of Main, as laid out in the non-code section.

G:\Visual Studio 2015\Projects\IMServer\Draw.io\Main.png

### distAndAuth.getPort

G:\Visual Studio 2015\Projects\IMServer\Draw.io\distAndAuth.getPort.png

Part, if not all, of these steps would be delegated to these additional local methods:

1. auth
2. getPublicKey

### newThread

Please note the method newThread defines and starts the following thread, while this flowchart shows what the new thread will do.

The final steps had to be included here as opposed to in instances.initialise as the instance of instances could not remove itself from the dictionary.



### instances.initialise

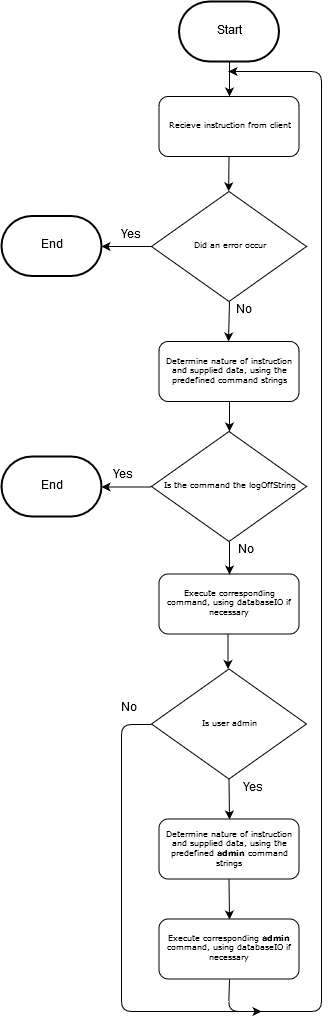
As mentioned in the class design section, instances is the master class which entirely handles a connection with one client.

### transmission.New

G:\Visual Studio 2015\Projects\IMServer\Draw.io\transmission.New.png

As can be seen from the previous flowchart, if socket isn’t defined, then when transmission.communicate is called an error will occur. Therefore, execution will return to instances.initialise immediately, where logOffSub will be called. This could present an issue since socket is not defined, which is why statusCode is set to 1.

### transmission.communicate



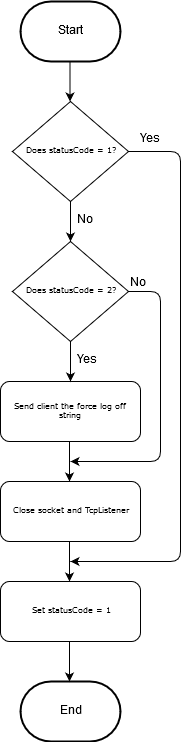
This loop will service all client requests until:

1. The client logs off cleanly, and sends the logOffString to the server

OR

1. The client logs off abruptly (e.g. system crashes) and an error occurs server-side

### transmission.logOffSub



The justification for this method can be found in the class design section under the server’s transmission class.

### Key Exchange

G:\Visual Studio 2015\Projects\IMServer\Draw.io\PGP.png

The client would decrypt the AES symmetric key with their RSA private key. From this point onwards all communication can be encrypted with the AES symmetric key.

### Authentication

Both databaseIO functions return either the username of the user if the operation completes successfully, or “NULL” if a problem arises.

### G:\Visual Studio 2015\Projects\IMServer\Draw.io\Authentication.pnguserOnlineRead – Sending past messages to user

When the user selects a friend to converse with, all previous messages must be retrieved and sent to the user. Since any one transmission is limited to 1024 bytes, the number of messages sent at once must be limited such that when encrypted this is not exceeded.

Pseudocode has been used for this algorithm because I encountered difficulty showing the required detail on a flowchart.

databaseIO’s userOnlineRead returns a list of all messages between two users in chronological order. partnerString is the string variable defined in the class design section(under transmission-server) and is used here as a delimiter between messages.

An example of the start of one of the strings sent to the client could be:

“!$%IM\_PARTNER^&\***User1: Hello**!$%IM\_PARTNER^&\***User2: How are you?**”

The maximum number of characters in a plaintext string that will be encrypted by AES-256 into 1024 bytes is 1023. Therefore, the next message (and its delimiter) should only be appended if the new length of the string is less than or equal to 1023.

.

A final transmission to the client is made outside the for loop in case the last iteration didn't go over 1023 - partnerString.Length, in which case there is still data waiting to be sent.

Method userOnlineRead(String user, String partner)

String data = “”

For Each message in databaseIO.userOnlineRead

If Length(data) + Length(message) <= 1023 – Length(partnerString)

data = data + partnerString + message

Else

SendToClient(data)

data = partnerString + message

End

End

SendToClient(data)

End

### Hashing – Compute Hash

### G:\Visual Studio 2015\Projects\IMServer\Draw.io\Hashing - Compute Hash.pngHashing – compare

A salt is provided if the call is being made from compare (next page). Otherwise this method facilitates the creation of a hash for a new password if no salt is provided.

### G:\Visual Studio 2015\Projects\IMServer\Draw.io\Hashing - compare.pngdatabaseIO

#### Existing user logging in

When a user attempts to log in to an existing account, this is the subroutine called (from distAndAuth). The password hash and salt should be obtained from tblUsers for the supplied username through a query.

Now either the query returned a hash and salt or it didn’t. In the latter case, the supplied username does not exist and “NULL” is returned as such. If a hash and salt is returned, the user’s supplied credentials need to be compared to those retrieved. Hence we call sha.compare with the user supplied password, the salt from the database and the hash from the database. If sha.compare returns false then “NULL” is returned, similarly to if an incorrect username were supplied. If compare returns true, then the user has supplied correct credentials (or a hash collision has occurred!) and the username is returned.

The choice to return a username instead of a Boolean value, for example, was fairly arbitrary, though it does simplify processing in distAndAuth somewhat.

#### Creating a new account

The SQL query:

SELECT PasswordHash, Salt FROM tblUsers WHERE Username = suppliedUsername

The pseudocode:

If queryReturned = True AND sha.compare(suppliedPassword, salt, hash) = True

Return suppliedUsername

Else

Return “NULL”

End

When a user wishes to create a new account, on the client side they merely check a box. This information is sent to the server and distAndAuth checks if the user is logging in to an existing account or creating a new one. If the user does wish to create a new account then this is the method called. It functions similarly to existingUser, and also returns either “NULL” or the supplied username.

There are a few nuances to creating a new account. Firstly, the database must be consulted to ensure the username supplied has not already been taken. Secondly, if the first condition is passed then the supplied password must be salted, hashed and stored in the database. Finally, if the first condition is passed then the user must automatically be friended with the admin account. Computing the hash and obtaining a salt is handles by the hash method in the sha class.

Checking if the username is taken and storing the salt and hash can be accomplished in one query in SQL. We check if the username exists by querying the database for any records with the supplied username. If the query does not return anything then the username does not exist and the salt and password hash can be inserted as a new record into tblUsers. If the original query does return something, then the entire query can return an indicative value (e.g. 1) to let the method know. Note that the entire query only returns a value if the username existed, since inserting values does not cause a return of any values. If the query returns something (i.e. 1) then the method can return “NULL” to signify this. Otherwise a record can be inserted into tblFriends which friends the user with the admin account.

#### Populate friends list of a user

The SQL queries:

1. IF NOT EXISTS (SELECT \* FROM tblUsers WHERE Username = suppliedUsername) BEGIN INSERT INTO tblUsers VALUES (suppliedUsername, passwordHash, salt) END ELSE BEGIN SELECT 1 END
2. INSERT INTO tblFriends VALUES (1, UserID, True)

The pseudocode:

If query (1) returned something

Return “NULL”

Else

Execute query (2)

Return suppliedUsername

End

friendsPopulate is a method whose purpose is to get a list of a user’s friends which can then be sent to the user. The server must also determine whether each friend is online.

From the database design section, it can be seen that there is only one record to indicate a relationship between two users. This was done to minimise storage requirements as storing user1 as the user and user2 as the friend and user2 as the user and user1 as the friend doubles the volume of information stored. However, this choice means we cannot simply get a list of FriendIDs from tblFriends whose UserID value is our user’s ID since we would be missing all the users who have our UserID in their friends column. For example, the following table shows how searching for all friends of “Alice” misses out “Charlie” and “Elon”. In this case only “Bob” and “Denise” would be found.

|  |  |
| --- | --- |
| **User** | **Friend** |
| Alice | Bob |
| Charlie | Alice |
| Alice | Denise |
| Elon | Alice |

To return all friends of a user we must check both the user and friend value of each record, and if either is our user’s username then the other value should be returned.

Another issue is that tblFriends stores IDs, not usernames, however this is easily solved by bridging to tblUsers’s username field using the correct ID. For example, if in a given record in tblFriends our user’s UserID is in the user column, then we bridge to tblUsers using the friend column’s ID. Similarly, if our user’s UserID is in the friend column, then we bridge to tblUsers using the user column’s ID.

Remembering that some friend requests may not have been accepted yet also imposes the condition that the Accepted field must be true. Finally, determining whether each user is online can be achieved by testing whether the connection dictionary contains the key of the username to check.

This method is called from transmission, when the client requests it, and from recommendFriends, when the client opens the search friends form (see later). In the first case, the username is known, and determining whether each user is online is desired. In the second case, usernames are not known, however user IDs are, and determining whether each user is online is not desired. To accommodate these differences, this method should be able to be called with either the username or user ID and supplying information regarding whether a user is online or not should be optional. Both of these cases can be handled with some simple selection, as will be shown in the pseudocode.

Commas are used delimiters between usernames and colons are used as delimiters between a username and whether it is online or not (Boolean). For example, User1:True, User2:False etc.

#### Get UserID from username

The SQL query:

SELECT tblUsers.Username FROM tblUsers, tblFriends WHERE tblFriends.Accepted = True AND ((tblFriends.UserID = suppliedUserID AND tblFriends.FriendID = tblUsers.UserID) OR (tblFriends.FriendID = suppliedUserID AND tblFriends.UserID = tblUsers.UserID)

The pseudocode:

If no UserID is supplied

UserID = GetUserID(suppliedUsername)

End

Create a new empty string variable data

If OnlineStatusWanted = True

For Each username in SQLQueryResult

data = data + “,” + username + “:” + connectionDictionary.DoesContainKey(username)

End

Else

For Each username in SQLQueryResult

data = data + “,” + username

End

End

Return data

In the previous block of pseudocode, I referred to a method getUserID. This method takes a username and returns the corresponding unique UserID. This is used by many of databaseIO’s methods, even if I have not explicitly included it in the relevant section. For example, in creating a new account, when creating the record in tblFriends the UserID of the user is needed and would be obtained by calling this method. Many methods’ first line of code would be calling this method.

Recalling the newUser method, when a user creates a new account, they must use a username not already in use. The effect of this is that each username is unique and maps to a unique UserID. Hence to get the UserID which corresponds to a username one must simply consult tblUsers.

#### Get messages between two users

The SQL query:

SELECT tblUsers.UserID FROM tblUsers WHERE tblUsers.Username = suppliedUsername

The pseudocode:

Return SQLQueryResult

This method is called userOnlineRead, since it reads messages when a user comes online and selects a friend to talk to. However, it can also be called when the admin wishes to see messages between two users. Therefore, it takes two parameters: the usernames of the two users. The admin controls both, while a normal user could only control one: the username of their partner.

This method returns a list of messages in chronological order. Since messages are directed, i.e. it is important who sent the message and who received it, each message is prepended with the sender’s username. For example, if the message is “hello” and the sender’s username is “Bob”, then one element in the list would be “Bob: hello”. This convention follows all the way through to the client which can then simply insert the message into the UI.

The same problem encountered in populating a user’s friend list presents itself here, that is, regardless of whether our user sent or received the message it needs to be sent to the user. It can be solved in the same way: by checking both the UserID and FriendID field to see whether either is our user’s ID.

If the UserID (i.e. the sender ID) is returned with the message body from the database then the sender of a message can be ascertained.

#### Storing a new message

The SQL query:

SELECT tblMessages.UserID, tblMessages.Text FROM tblMessages WHERE (UserID = suppliedUserID AND FriendID = suppliedFriendID) OR (FriendID = suppliedUserID AND UserID = suppliedFriendID)

The pseudocode:

suppliedUserID = getUserID(suppliedUsername)

suppliedPartnerID = getUserID(suppiledPartnerUsername)

Create a new empty string list data

Execute SQLQuery

For Each SQLQueryResult In SQLQueryReturn

If SQLQueryResult.UserID = suppliedUserID

data.Add(suppliedUsername + “: ” + SQLQueryResult.Message)

Else

data.Add(suppliedPartnerUsername + “: ” + SQLQueryResult.Message

End

End

Return data

This method caters for the very frequent scenario in which a message is sent from one user to another. Regardless of whether the recipient is online or not, the message still needs to be stored in the database.

While essential, this method is quite simple needing only to insert a new record into tblMessages. getUserID will be used to get the corresponding IDs for the supplied usernames of the sender and recipient. Then these IDs, along with the message itself, can be stored.

#### Searching for a friend

The SQL query:

INSERT INTO tblMessages VALUES (Message, suppliedUserID, suppliedPartnerID)

The pseudocode:

suppliedUserID = getUserID(suppliedUsername)

suppliedPartnerID = getUserID(suppiledPartnerUsername)

Execute SQLQuery

The client application has the capability of searching, by username, for a potential friend. From a security standpoint, this is an obvious candidate for a SQLi attack, which has been handled by both blocking escape characters in the client application and double escaping out single quotes server-side.

SQL has a “like” operator which can be used to specify simple patterns similarly to regex. The % wildcard specifies zero or more characters, so a very simple pattern to search for would be %INPUTSTRING% where INPUTSTRING is the string the user enters. Therefore, as long as the user knows part of the username, the remaining characters can be filled in for them. Of course, this does allow for trivial exploitation such as entering a common character such as ‘a’ and the server returning all usernames containing ‘a’. However, as a proof of concept it does work.

Usernames are stored in tblUsers, so we merely need to SELECT all usernames FROM tblUsers LIKE our pattern. These can then be delimited with commas and returned to transmission.

Clearly, users already friended with the user should be filtered out, however this is done client-side to relieve some of the load on the server.

#### Send a friend request

The SQL query:

SELECT Username FROM tblUsers WHERE Username LIKE ‘%suppliedUsername%’

The pseudocode:

Execute SQLQuery

Create a new empty string variable data

For Each username in SQLQueryResult

data = data + “,” + username

End

Return data

After searching for a friend, the user can send a friend request to a given username, to be later approved by the admin account. Accommodating this request requires very little in terms of processing.

First the IDs of the user and their “partnerToBe” are obtained through calls to getUserID, at which point the IDs can be inserted straight into tblFriends, setting the Accepted field to False to signify that the admin account has not yet approved it.

#### Recommend friends

The SQL query:

INSERT INTO tblFriends VALUES (userID, partnerToBeID, False)

The pseudocode:

userID = getUserID(suppliedUsername)

partnerToBeID = getUserID(suppliedPartnerToBeUsername)

Execute SQLQuery

When the user opens the searchFriends form on the client application, a list of recommended friends should be automatically shown. What constitutes a user worthy of recommendation is debatable, but I have taken it to mean that the number of our user’s friends who have a friend not known to our user is greater than or equal to some threshold value rcmdFriendsDepth.

If, say, rcmdFriendsDepth equals 2 then at least 2 of our user’s friends must be friended with another user (who isn’t friends with our user) for said user to be recommended. The following example may make matters clearer.

|  |  |
| --- | --- |
| **User** | **Friends** |
| User1 | User2, User3 |
| User2 | User1, User4 |
| User3 | User1, User4, User5 |
| User4 | User2, User3, User6 |

In this example, if rcmdFriendsDepth equals 2, then User1 would be recommended User4 as a friend since at least two (User2 and User3) of User1’s friends are friends with User4. Neither User5 nor User6 would be recommended to User1 since only one of User1’s friends are friends with User5 or User6 (User3 and User4 respectively). Also, and obviously, User2 and User3 should never be recommended to User1 since User1 is already friends with them and User1 should never be recommended to themselves. In my project I will take rcmdFriendsDepth to be 3, though this is a rather arbitrary choice.

There are some obvious similarities between this method and friendsPopulate, and the following steps make use of this.

We begin by getting the UserIDs of all the friends of our user, omitting the admin and unaccepted friend requests. Actually, we would get both the UserID and FriendID and then filter in exactly the same way as in friendsPopulate, but this is beside the point. We can omit unaccepted friend requests by imposing the condition that the Accepted attribute is equal to True and omit the admin’s UserID by imposing the condition that the UserID attribute is not 1. We don’t have to consider the possibility of the admin’s UserID being in the FriendID column because this friend request is inserted by the server when a new account is created (see previous section). The reason for omitting the admin is because every user is friended with the admin, including all of our user’s friends. Each UserID will be inserted into a list, so we now have a list of all the UserIDs of our user’s friends.

We then define a new empty string, which will eventually hold the usernames of all our friends’ friends. For each UserID in the list, we call friendsPopulate with the ID, decline online information and append the result to the string. In our previous example, our string would now be “User1,User4,User1,User4,User5” with the first two usernames being the friends of User2 and the last three being the friends of User3.

As specified before, our user’s friends should not be recommended and neither should the user himself. However, since we don’t have the friends’ usernames, we will just remove all occurrences of our user’s username and let the client filter out the rest. So now our previous example string becomes “User4,User4,User5”.

To iterate through each friend, we can create an array by splitting the string up using the comma delimiters. Doing this allows for reusing the friendsPopulate subroutine which returns a string. Next, we can create an empty list which will eventually hold the recommended friends. If we iterate now through the array holding each friend’s friends we can use the fact that the number of times a username appears is equal to the number of our user’s friends who are friends with that user. So, for example, User4 appears twice and is a friend of two of User1’s friends. We can count the number of occurrences of the current username and if it is greater than or equal to the rcmdFriendsDepth the username can be added to the list.

The only problem here is that since a username can appear more than once in the list (by necessity), the same friend can be in the recommended friends list more than once. To remedy this, we just ensure that every element in the list is unique, which VB makes very simple with the Distinct method callable on lists. Finally, compile into a string and return.

The SQL query:

SELECT UserID, FriendID FROM tblFriends WHERE Accepted = 'True' AND (UserID = userID OR FriendID = userID) AND NOT UserID = 1

The pseudocode:

userID = getUserID(suppliedUsername)

Execute SQLQuery

Create new empty string list IDList

For Each userIDPair in SQLQueryResult

If SQLQueryResult(0) = userID

IDList.Add(SQLQueryResult(1))

Else

IDList.Add(SQLQueryResult(0))

End

End

Create new empty string nameString

For Each ID in IDList

nameString = nameString + friendsPopulate(ID)

End

nameString = nameString.Remove(suppliedUsername)

Create new string array arraySplit = nameString.Split(,)

Create new empty string list recommendList

For Each username In arraySplit

If arraySplit.CountNumberOf(username) >= rcmdFriendsDepth

recommendList.Add(username)

End

End

recommendList = recommendList.Distinct

Create new string returnString

For Each username in recommendList

returnString = returnString + username + “,”

End

Return returnString

#### Admin Get Friend Requests

As specified before, the admin account approves friend requests, but to do this the admin must first be able to view the proposed friend requests. Only unapproved friend requests should be displayed so the condition of the Accepted attribute being False can be imposed.

Since we need both usernames for each request, we can simply bridge from tblFriends to tblUsers with either the UserID of tblFriends matching the UserID of tblUsers, or the FriendID of tblFriends matching the UserID of tblUsers.

If no unapproved friend requests are present, then the SQL request will return nothing, in which case the server can send the client a signifying string. If the SQL query does return something, we know there must be an even number of usernames. Therefore, the usernames can be consecutively appended into a string, delimited by commas, and sent to the client who knows two consecutive usernames constitute one friend request.

#### Admin Accept Friend Request

The SQL query:

SELECT tblUsers.Username FROM tblUsers, tblFriends WHERE tblFriends.Accepted = 'False' AND (tblFriends.UserID = tblUsers.UserID OR tblFriends.FriendID = tblUsers.UserID)

The pseudocode:

Execute SQL Query

Create new empty string friendRequestString

If SQLQueryResult is not null

For Each username in SQLQueryResult

friendRequestString = friendRequestString + username + “,”

End

Else

friendRequestString = “NULL”

End

Return friendRequestString

Once the admin has retrieved all unapproved friend requests from the server, he can either approve or reject each one. In the case a particular request is approved, this is the method to be called.

All that needs to be done to accept a friend request is change the Accepted attribute in tblFriends to True. The only problem is that the client only knows the user and friend’s usernames, not UserIDs, though this is quickly handled with getUserID.

#### Admin Reject Friend Request

The SQL query:

UPDATE tblFriends SET Accepted = 'True' WHERE (UserID = userID AND FriendID = friendID) OR (FriendID = userID AND UserID = friendID)

The pseudocode:

userID = getUserID(suppliedUserUsername)

friendD = getUserID(suppliedFriendUsername)

Execute SQLQuery

Once the admin chooses to reject a friend request, the request no longer needs to be stored in the database and can be deleted. If we merely left the Accepted attribute as False, then this friend request would be displayed to the admin every time he gets friend requests. Furthermore, it would be a needless use of storage.

Once again, the record can be identified using the getUserID method and the usernames supplied by the client.

#### Admin Delete User

The SQL query:

DELETE FROM tblFriends WHERE (UserID = userID AND FriendID = friendID) OR (FriendID = userID AND UserID = friendID)

The pseudocode:

userID = getUserID(suppliedUserUsername)

friendD = getUserID(suppliedFriendUsername)

Execute SQLQuery

Obviously, no student stays at school forever and eventually accounts would need removing. There are numerous aspects to this. Firstly, their account details need deleting from tblUsers, secondly all friend requests, approved or otherwise, need removing from tblFriends and finally, any messages from or to this user need deleting from tblMessages.

However, attempting to delete information from the database in this order would cause problems due to foreign key dependencies. Both tblMessages and tblFriends reference tblUsers’ UserID field as a foreign key and so tblUsers’ information must be deleted last.

#### Admin Change Password

The SQL query:

DELETE FROM tblMessages WHERE UserID = userID OR FriendID = userID; DELETE FROM tblFriends WHERE UserID = userID OR FriendID = userID; DELETE FROM tblUsers WHERE UserID = userID

The pseudocode:

userID = getUserID(suppliedUserUsername)

Execute SQLQuery

The final piece of admin functionality is the ability to change a user’s password. This could be useful if, for example, a student forgot the password to their account. Recalling the method called when a new account is created, the new password will have to be hashed and salted before replacing the original password hash and salt in the record.

To hash the password, we can call sha.hash with the plaintext password and passing in nothing as the salt so the method knows to generate a random salt. The array returned can then be used to update tblUsers, using the username as the unique identifier.

The SQL query:

UPDATE tblUsers SET PasswordHash = passwordHash, Salt = salt WHERE Username = suppliedUsername

The pseudocode:

Create new byte array hashAndSalt = sha.hash(suppliedPassword, Nothing)

passwordHash = hashAndSalt(0)

salt = hashAndSalt(1)

Execute SQLQuery

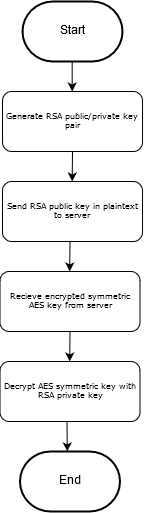
## Client Algorithms

### Logging in/out

This method is called when the user clicks the log in button (which is renamed log out upon logging in) and is somewhat similar to the server’s Main. The various other methods referenced here will be covered in due course.

G:\Visual Studio 2015\Projects\IMClient\Draw.io\Login_logout.png

### Key exchange



The server would generate an AES symmetric key and encrypt said key with the client’s RSA public key.

### distAndAuth.getPort

G:\Visual Studio 2015\Projects\IMClient\Draw.io\distAndAuth.getPort.png

Part, if not all, of these steps would be delegated to these additional local methods:

1. auth
2. sendPublicKey

### Setup transmission



At this point, transmission fully handles all communication with the server, just as one instance of instances is handling all communication with this client server-side.

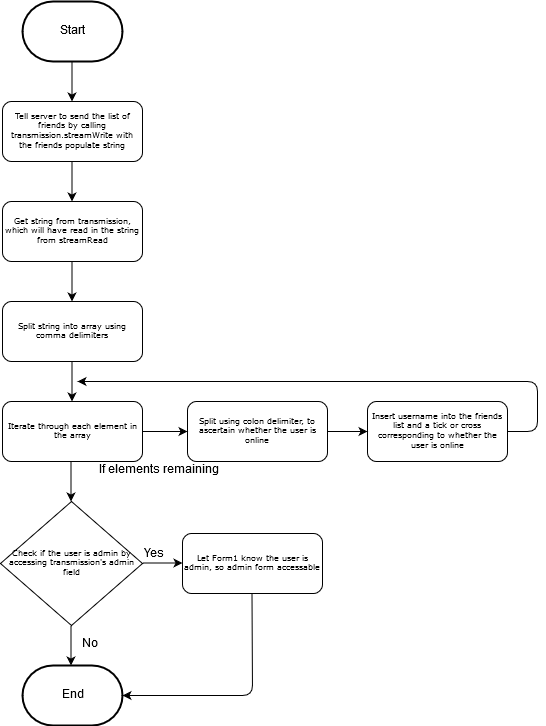
### transmission.streamRead

### transmission.streamWrite

This method is similar to the server’s transmission.communicate and these two methods are the foundation of communication between the client and server. The various commands alluded to here will be covered later.

### Populate friends list

Every time I say ‘send data to the server’ I mean call this method. Therefore, everything sent over the internet is encrypted.



This method is called immediately after the user logs in. The check for admin is put here simply because it is executed early.

An example string received by transmission.streamRead is:

User1:True,User2:False,User3:False

This would result in the following:

User1 Online

User2 Offline

User3 Offline

### Sending a message

### Profanity detector

This method would be called when the user clicks the submit button.

An example of the final string sent to the server (before encryption) could be:

!$%IM\_MESSAGE^&\*User1: Message

Where the first part is the message string identifier as outlined in the class design section.

G:\Visual Studio 2015\Projects\IMClient\Draw.io\Profanity detector.png

This method would be called when the user attempts to send a message to another user.

A similar algorithm is used to check for invalid characters in the user’s login credentials when they attempt to log in and when they search for friends.

### Adding messages to the list

This algorithm utilises recursion, which I found difficult to show on a flowchart. Therefore, pseudocode has been used here.

There are multiple situations where a message (or messages) may need to be added to the list of messages the user is currently looking at. These are: when the user submits a message, when the user receives a message from another user and when the user selects a new partner and all messages between them must be displayed. In the first two cases, only one message needs adding to the list, but in the last case multiple messages could need adding. Finally, only the first case would be called from Form1, which owns the ListView object used to display messages. In the remaining two cases, this method would need to be called from transmission’s streamRead which is not only in another class but also running in another thread.

For transmission to access the UI thread (Form1), transmission must call an invoke, which essentially asks the UI thread to perform the operation for transmission. VB has a very useful function InvokeRequired which, when called on an object such as Form1’s ListView, will return a Boolean value indicating whether an invoke is necessary. So, to account for the different locations where this method can be called from, the method should start by checking if an invoke is required on ListView and, if so, invoking itself.

Now, each message will be in the form “User: message”, which allows us to extract the username of the user sending of the message. This can either be our user (they sent the message to someone else), the user they’re currently in a conversation with aka their partner, or the username of one of their friends, but not the one they’re currently in a conversation with. So, either the user’s username, their partner’s username, or neither. This information is useful because it allows messages to be colour coded, and also lets us determine whether the message belongs in the current list. For example, if our user, User1, was talking to User2, messages sent by User1 could be one colour and messages sent by User2 another colour. If User1 then receives a message from User3, this should not be inserted into the list of messages between User1 and User2, but instead notify the user they received a message from another user. This final action needs to be suspended if the admin is getting messages between two users, as otherwise none of them would be displayed. To accommodate this, a second argument could be passed to this method, which acts as an admin override.

Iterating through each message in the list of messages provided, we begin by getting the username of the user who sent the message. We need the substring of the message from the start up till and including the colon and space. The reason for not just getting the username will become apparent later. Since the string is zero indexed, the amount of characters needed is the index of the colon plus two (one for the indexing and another to include the space). So if the message were “User1: My message”, the username obtained would now be “User1: “. Comparing this to the username of the user’s partner (not forgetting to add the colon and space) allows us to determine whether the message is from the partner or not.

The message needs adding if any of the following conditions are met: the message is from the user, the message is from the partner, or the command was issued by the admin in which case the adminOverride argument should have been passed in. If none of these conditions are met, the user is notified and this iteration ends. Otherwise, we begin the work which really necessitates this method.

The ListView is obviously limited in size, much more so than the allowed length of messages. Therefore, a long message can’t actually fit on one line, and needs splitting over multiple lines. However, then the issue arises of where to split the message. The message should still be easily readable, but use as much space as possible on the first line. The last space that fits on one line is a good candidate, as a single word will never need to be split over two lines. If the message has no spaces, then just fit what will on one line. Of course, this is all dependant on the message being longer than the size of the ListView. If this is not the case, then the message can just be inserted, setting the colour as necessary.

To get the zero-starting index of the final space, we must first obtain the substring of where the message actually starts to the end of what will fit on one line. Since the extracted username included the colon and space, we know the starting index of the actual message is equal to the length of the stored username. The length of the substring is the size of the ListView subtract the length of the username. We can then find the index of the last space in this substring and then add on the length of the username to get the index of the last space in the whole string. This should be stored. We have to substring first so if the string contains no spaces the index isn’t set to the space in the username. If there are no spaces, the LastIndexOf function referenced in the pseudocode below should return -1.

If there are no spaces then the index stored would be the length of the username minus 1. In this case, the message can be split at the index of the last character that fits on one line, which would be the size of the ListView minus 1 to account for zero indexing. If there were spaces we now have the index of the last space which will fit on one line. Regardless, the message colour can now be set depending on whether the user which sent the message was our user’s partner or not. The message can now be inserted and what remains of the original string can be prepended with the username and passed into a recursive call to this method. Note this call is still inside the for loop.

The method will keep calling itself until the whole string is displayed, at which point it will unwind to the original call which can then proceed to the next message in the list.

In the pseudocode below, Substring is a function callable on a string which takes two parameters: the index of the initial character and the number of characters to take. If only the first parameter is supplied, all following characters in the string are obtained. Choose partner

Method AddItems(List of messages: Items, Boolean variable: adminOverride)

If ListView.InvokeRequired = True

ListView.Invoke(AddItems(Items, adminOverride))

Else

Create integer variable Index

Create string variable Name

Create Boolean variable isPartner = False

For each item in Items

Name = item.Substring(0, item.WhereIs(“:”) + 2)

If isPartner = True OR Name = username + “: “ OR adminOverride

If Length(item) > ListViewSize

Index = item.Substring(Length(Name), ListViewSize-Length(Name)).IndexOfTheLast(“ “) + Length(Name)

If Index = Length(name) – 1

Index = ListViewSize - 1

End

StringToInsert = Item.Substring(0, Index)

Set font based on isPartner value

ListView.Insert(StringToInsert)

AddItems(String Array(Name+Item.Substring(Index+1)), adminOverride)

Else

StringToInsert = Item

Set font based on isPartner value

ListView.Insert(StringToInsert)

End

Else

Notify user, message not from partner

End

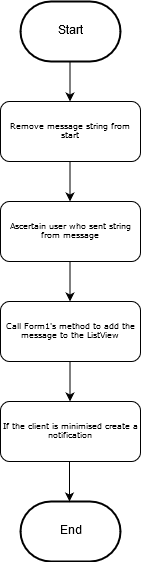
isPartner = False

End For

End

When the user selects a partner to talk to, the client must inform the server so that all messages between the two users can be retrieved. The partner string is the identifier outlined in the class design section.

### Receive message(s)

The client can receive one message if another user sends them a message, or multiple messages if the user selects a partner and messages between them must be displayed. In the first case the string sent will start with the message identifier string. In the second case, each message will be delimited with the partner string. The flowchart on the left shows how to deal with the first case, and the right the second.



### Receive Populate Friends Response



When the user requests their list of friends be retrieved, the server sends it and transmission’s streamRead stores it in friendsPopString. Then Form1’s method which handles populating the friends list can read it out and process it.

### Receive Search Friends Response

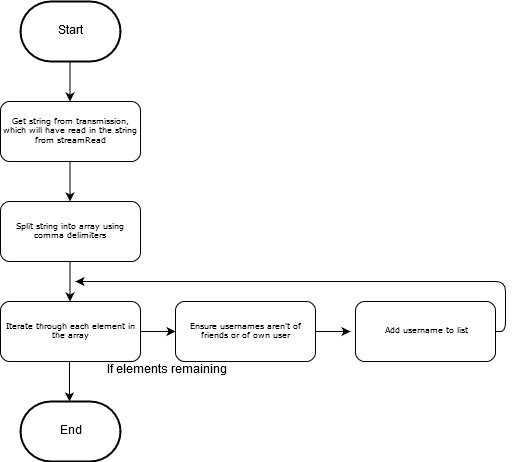
This is almost identical to the previous section. When the user searches for friends the server sends the client a list of usernames. Transmission’s streamRead stores this in a field which is then accessed by the relevant method in another class.

### Receive Admin Account

When the server tells the client that the user is the admin, transmission’s streamRead needs to let Form1 know. Transmission’s admin field is set to true, and Form1 checks the status of this field at the end of friendsPopulate.

### findFriendsForm Algorithms

#### Fill Friends List



This method is similar to the method used to populate the friends list. The exception is that this method is called to insert usernames of potential friends which have been searched for or which have been recommended by the server.

#### Search for a friend

This method would be called when the user clicks a button in the search friends form.



#### Sending a friend request



Upon receiving a list of usernames, the user can click on any one to send a friend request to that user.

### Admin form algorithms

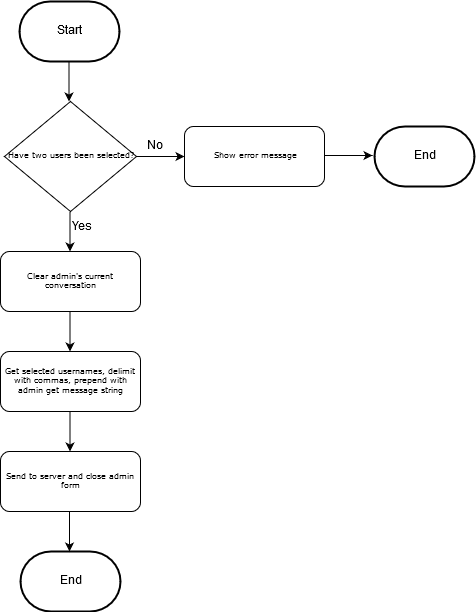
#### Get list of users and friend requests

This algorithm is almost identical to the method which populates the user’s friends list, and similarly makes use of fields in transmission. The only difference is the data the server sends to the client. The unapproved friend requests sent by the server are in the form “User1,User2,User3,User4” where User1 has sent a friend request to User2 and User3 has sent a friend request to User4. Therefore, splitting the string based off the comma delimiters and a simple for loop are all that is required to display the friend requests.

#### Accept/Reject Friend Requests



#### View messages between two users



Once the string is sent to the server, the admin form closes because the messages will be displayed in Form1. The server returns the messages as if the admin were retrieving messages between themselves and another user. Then the messages are passed into addItemsListView with the admin flag.

#### Change password

To change a user’s password the admin merely has to select a user, enter the new password and click a button. The client will delimit the username and password with a comma, and then prepend the whole string with the admin change password string. This can then be sent to the server.

#### Delete user

Similarly, to delete a user the admin must select a user and click a button. The client will confirm the action, prepend the username with the admin delete user string and send the result to the server. The user will then be removed from the list.

# Testing

## Outline

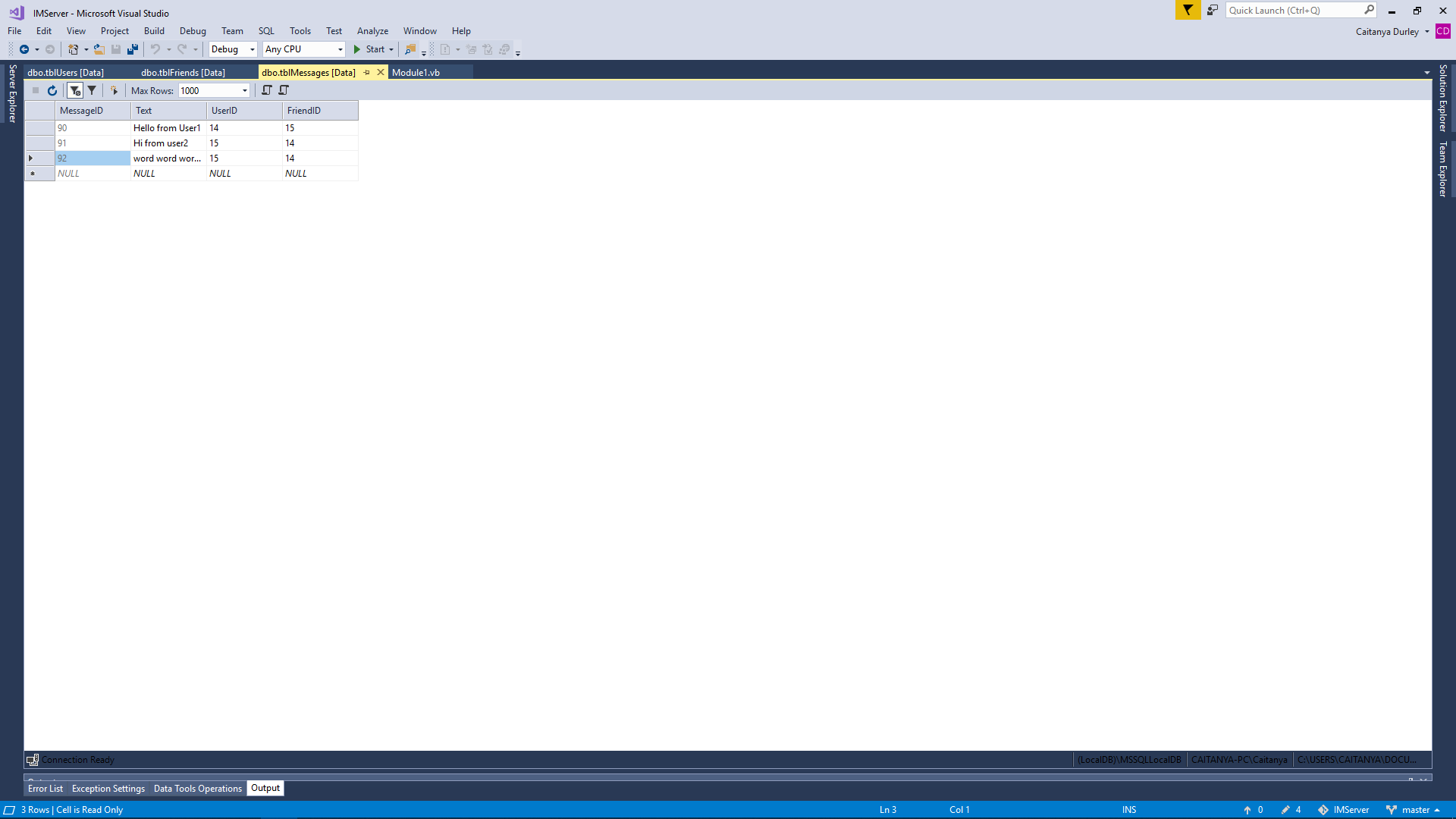
Please note that this system will be tested on a LAN, with the server and two client instances running on one device (for recording simplicity). I would have preferred to run the server on another device on the network however required SQL libraries have made this difficult on my home network. The client feedback section in the evaluation does demonstrate the service working over three separate computers, however.

The testing video can be found here: <https://youtu.be/3rWbtLmaBtc>

## Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test ID** | **Description** | **Data/Action** | **Expected output** | **Actual result** | **Evidence** |
| 1.1 | Logging in – incorrect username/password | Enter an incorrect username/password and attempt to log in | Not logged in, message box informing user | Not logged in, message box informing user displayed | 0:53 |
| 1.2 | Logging in – correct username and password | Enter correct credentials and attempt to log in | Log in user, clear details from login entries | User is logged in and username and password entry boxes cleared | 1:09 |
| 1.3 | Logging in – correct username and password but already logged in on another device | Attempt to log in from two devices | First device should be disconnected and second device logged in | First device is disconnected and second device logged in | 1:32 |
| 1.4 | Logging in – correct username and password but server reached maximum number of connections | Enter correct credentials and attempt to log in | User should not be logged in and an error message should detail why | User is not logged in and an error message details why | 2:34 |
| 1.5 | Create account – taken username | Attempt to create an account with a username that is already taken | Error message informing the user the username is taken should be displayed | Error message informing the user the username is taken is displayed | 3:10 |
| 1.6 | Create account – new username | Attempt to create an account with a username that is not taken | New account should be created and automatically logged in. The user should be automatically friended with the admin account | New account is created and user automatically logged in. The only friend is the admin account | 3:28 |
| 2.1 | Populate friends list | Upon logging in, the friends list should be automatically populated | All friends’ usernames should be displayed along with whether they are online | A list of friends’ usernames with a tick indicating online and a cross indicating offline are displayed | 3:52 |
| 2.2 | Populate friends list manually | The same function can be triggered by pressing the refresh button | See 2.1 | See 2.1 | 4:27 |
| 3.1 | Select a friend to talk to | Click on a friend’s username | Friend’s username should be highlighted | Friend’s username is highlighted | 4:36 |
| 3.2 | Upon selecting a friend, messages between the user and this friend should be displayed | See 3.1 | Messages should be displayed between the two users in chronological order | Messages are displayed between the two users in chronological order | 4:50 |
| 4 | Messages should be formatted correctly | See 3.2 | Long messages should be split over multiple lines, different colour per user | Messages split correctly and all correctly colour coded | 5:45 |
| 5.1 | Sending messages to an online user | User types in message and clicks send | Message should be displayed sender-side | Message is displayed sender-side | 6:16 |
| 5.2 | Receiving messages when in conversation with sender | The recipient has selected the sender as their conversation partner. See 5.1 for sender-side action | Message should be appended to conversation | Message is appended to conversation | 6:49 |
| 5.3 | Receiving messages when not in conversation with sender | The recipient has not selected the sender as their conversation partner. See 5.1 for sender-side action | Recipient should be alerted to new message not from current partner | Recipient alerted to new message not from current partner | 7:13 |
| 5.4 | Sending messages to an offline user | User types in message and clicks send | Message should be displayed as if recipient were online | Message displayed as if recipient were online | 8:03 |
| 5.5 | Receiving messages when offline | Next time user logs in and selects the sender, the message is shown | Message should be shown | Message shown | 8:27 |
| 5.6 | Sending message with profanity in | User types in an unclean message and clicks send | Message should not be sent and error displayed | Message not sent and error displayed | 19:29 |
| 5.7 | Receiving messages when client application minimised | The client application is closed when the user receives a message | A notification should let the user know they received a message | A notification shows the sender and the message | 8:53 |
| 6.1 | Open search friends form | Click on find friends button | The search friends form should open | The search friends form opens | 9:21 |
| 6.2 | See recommended friends automatically | See 6.1 | Upon the form opening, a list of recommended friends should be retrieved | Recommended friends shown. Recommended friends meet criteria (recommended friends depth = 2) | 9:33 |
| 6.3 | Searching by username | Type in part of a username and press send | A list of usernames similar to this should be listed. Friends’ usernames should be omitted | A list of similar usernames are shown.  Usernames of friends are omitted | 10:35 |
| 6.4 | Sending a friend request | Clicking on any username that has been retrieved by 6.3 | A dialog box should confirm the action before sending a friend request | A dialog box confirms the action and a friend request is sent | 11:05 |
| 7.1 | Login as admin | Enter admin credentials | Find friends button should be replaced with admin button | Find friends button is replaced with admin button | 11:18 |
| 7.2 | Open admin form | Click on admin button | The admin form should open | The admin form opens | 11:40 |
| 7.3 | See unapproved friend requests automatically | See 7.2 | Upon the form opening, a list of unapproved friend requests should be retrieved | Unapproved friend requests shown | 11:48 |
| 7.4 | Accept a friend request | Select a friend request and click accept | Friend request approved and removed | Friend request approved and removed | 12:06 |
| 7.5 | Reject a friend request | Select a friend request and click reject | Friend request rejected and removed | Friend request rejected and removed | 12:06 |
| 7.6 | View messages between two users – selected two users | Select two users and click view messages | Admin form should close and messages should be shown in the usual place. Any current conversation the admin is in is should be closed | Admin form closes and messages shown in list view. Current conversation closed | 13:19 |
| 7.7 | View messages between two users – didn’t select two users | Select one user instead of two and click view messages | Error message should be displayed explaining the error. | Error message states two users should be selected. | 14:15 |
| 7.8 | Remove user | Select a user and click remove user | Message box should confirm action and user should be removed | Message box confirms action and user is removed upon confirmation | 14:31 |
| 7.9 | Change password | Select a user, enter the new password and click change password | User’s password should be changed | User’s password changed | 15:09 |
| 8.1 | Large string input to message box | Enter a large string as a message to a friend and send it | Message length should be limited, receiver should display message correctly | Message length limited sender-side, and the message is displayed correctly receiver-side | 16:03 |
| 8.2 | Message input sanitisation | Enter an apostrophe, which is an SQL escape character | Message should display correctly, with no issues server-side | Message displays correctly, with no issues server-side | 17:16 |
| 8.3 | Encryption and hashing | Any request to the server | Messages should be encrypted before sending over the network and only password hashes and salts should be stored in the database. Plaintext passwords should not be retrievable. | I found it very difficult to show that encryption was in place, but this can be verified by inspecting the source code. The database shows only password hashes and salts are stored. | 17:53 |

## Screenshots



# Evaluation

## Objectives

The objectives outlined in the analysis section are copied here for ease of reference.

**Server-side**

* Accept connections from clients
* Verify authentication details by consulting the database
* Facility to create a new user
* ‘Distribute’ client connection to a free port
* Keep track of free/used ports
* Receive ASCII messages from clients
* Forward to the relevant partner
* Store messages in a database
* Allow users to have ‘friends’ and search for potential friends
* Handle multiple connections at once
* Recommending friends to the user
* Admin facilities: monitor messages, reset passwords, approve/reject friend requests, remove users
* Encryption and hashing

**Client-side**

* Connect to server
* Supply username and password to the server
* Send ASCII messages to the server
* Choose desired ‘partner’ from list
* Ability to see if users are online
* Retrieve all messages in the current conversation
* Search for friends and send friend requests (to be approved by the admin account)
* Simple user interface
* Profanity detection
* Encryption
* Group messaging

### Server-side

#### Accept connections from client

Clearly this objective has been met. The server is nearly always waiting for clients to connect, and can establish two-way communication with the client. Theoretically, if a client attempted to connect while the server was busy authenticating another client then their connection would be refused but, in practice, I have not been able to create this situation. Even clicking log in on two client applications at the same time has not given rise to this, presumably due to minute differences in latency. The only other case in which a client would be refused access is if the server had reached its maximum number of connections, however this is easily changed and is present to maintain stability.

#### Authentication using database

This objective has been met. When the client connects to the server, they provide a username and password. The server successfully checks if the username is present in the database. If so, it retrieves the stored hash and salt and compares this to the hash computed from the user’s password and retrieved salt. The server can then either allow the client access if they supplied correct credentials or inform the client otherwise. I have not discovered a case where authentication has raised an error or mistakenly allowed access.

Obviously SQLi attacks are an issue here, and input should really be sanitised server-side, however input is sanitised client-side and by this point everything is encrypted with a symmetric key which was never sent unencrypted to or from the client. Therefore, intercepting the transmission from the client, for example with a proxy, and editing the user input would necessitate obtaining the symmetric key from a memory dump (as far as I am aware). Alternatively, one could feign being a legitimate client, and manually interact with the server by guessing the steps taken during connection and authentication and what constitutes a valid request. Then the attacker could send their own symmetric key and hence craft their own requests and send these to the server. However, both of these potential vulnerabilities are beyond the scope of this project, which is closer to learning how to develop an instant messenger.

#### Facility to create a new user

This objective has been met. There is little to say here apart from that this facility works. By selecting a checkbox, the user can inform the client and hence the server that they wish to create a new account. The username and password are sent to the server and inserted into the database. If the username is taken the client is informed, otherwise the client is logged in to their new account and automatically friended with the admin account. I have not had any issues with this functionality not working as intended. Once again, the previous vulnerabilities apply here, however.

#### Distribute client connection to a free port

This objective has been met. A queue of free ports is maintained server-side so that the server always knows which port are available for use. Once a client authenticates, the server will dequeue a free port, setup a listener on it and send the port number to the client.

This works as intended in all cases. The only potential caveat would be if the queue was empty, however this is already avoided by the maximum number of connections which defines the size of the queue and would dictate that the new client is refused access.

#### Keep track of free/used ports

This objective has been met. Keeping track of free ports was discussed in the previous section. Knowledge of used ports is obtainable by omission from the queue, however knowing which ports are used hasn’t really been necessary. The only case it could be useful in is when messages are forwarded to another connected client, however this is handled through OOP anyway.

#### Receive ASCII messages from clients

This objective has been met. The user enters a text message using the ASCII character set and the client encrypts it and sends it to the server. The server can then decrypt and store the message. The client limits the length of the entered string, so there are no problems with a message being too large for the Text field in tblMessages.

While everything relating to this objective works as intended, I do question my choice of ASCII as the character set. Unicode would have greatly expanded the characters users could send, which could be useful in the context of students needing to send, for example, mathematical symbols. Furthermore, I doubt it would require any significant changes to the source code of both the client and server applications.

#### Forward messages to the relevant partner

This objective has been met. Once the server receives and stores a message as outlined in the previous section, it will need forwarding to the user it was intended for, providing the user is online. This can be checked using the connection dictionary, and for the rest of this section we will assume that the user is online.

The partner’s instance of transmission can then be accessed through the dictionary, and their transmission’s socket and AES fields accessed through this. The message can then be easily encrypted and sent to the partner’s client. This objective underpins the instant part of an instant messenger; the user’s message is instantly forwarded to the intended recipient. I was very pleased with the speed at which messages were transmitted from device to device during testing with my client (see later section), albeit with the server hosted on the same network as the devices.

#### Store messages in the database

This objective has been met. As detailed in the design section, all messages must be stored in the database so they can be retrieved at a later date if necessary. Doing so is a simple case of inserting a record into tblMessages. I have not encountered any cases where this fails. The client restricts the length of messages, and the server sanitises input, so this should be secure.

In hindsight, storing a timestamp with each message would have been useful for both the users and from a data processing point of view. That being said, since messages are stored and hence retrieved in chronological order, timestamps are not strictly necessary.

#### Have friends and search for friends

This objective has been met. A user can have friends, which are displayed and available for selection as a conversation partner. It is only possible to converse with a user if they are your friend.

A client’s request to the server to search for friends is catered for by a SQL query for usernames similar to that provided. A very lenient pattern that a username is matched against makes it easy to obtain a list of nearly every user connected to the server, though this is easily changed and less of an issue in a school environment. However, the feature does work and provides the foundation for sending friend requests.

#### Handling multiple connections at once

This objective has been met, though the solution may not be scalable. Handling multiple connections at once is a rather broad objective, though I didn’t realise this at the time of writing the objectives. There is the obvious necessity of splitting processing time between the various client requests, but also handling how connections interact with each other, how these connections are organised and identified, and how new connections can be simultaneously added.

As detailed extensively in the design section, each connection can be encompassed by an instance of a master class, named instances. This allows for interaction between connections, as one instance of instances can access another and hence modify the connection between the server and another user. A username and instance of instances make up a key-value pair which is stored in the connection dictionary. This allows unique identification of connections, and organises them neatly. New connections are created through a new instance of instances, and added to the dictionary. Each instance is started in a different thread, so all can run simultaneously.

This solution seems to handle multiple connections well, as evidenced in my testing. Also, in Windows a single application can run 2000 threads, which means the server could theoretically handle 2000 simultaneous connection at once. That being said, I do have serious concerns about the stability of this solution long before the number of simultaneous users gets anywhere near that number.

#### Recommending friends

This objective has been met. I won’t go into how this was accomplished as this was covered in the design section. My approach of mutual friends seems justified and certainly seems to work in practice, however I can see some situations where it may not be ideal. For example, say many students study maths and so are ‘friends’ with their maths teacher. If a student not studying maths is friends with these students (because they all study computing, for example) then the maths teacher would be a recommended friend. There is little I can do about this, without storing further information, as would be done in a true social media akin instant messenger.

#### Admin facilities

This objective has been met. The admin account can view messages between two users, change passwords, approve or reject friend requests and remove users. All these facilities can only be accessed if the server recognises the user as the admin and informs the client. Therefore, even if an admin request were correctly forged by a non-admin user, it would not be executed server-side.

I chose to incorporate the admin facilities into the standard client application to enable reuse of code, however, a separate application just for the admin controls would improve security and make the user experience more streamlined for the admin. Mr Deane could then have a separate account which is the same as all the students’ in addition to the admin application.

Nevertheless, all the aforementioned facilities work as requested by Mr Deane in the interview.

#### Encryption and Hashing

This objective has been met. Encryption of messages sent over the internet was a high priority for my client, in addition to encryption of stored passwords.

I made the decision to, instead of storing encrypted passwords and presumably the key for decryption, store hashes of the passwords which could then be compared to hashes of the password supplied by the user. Therefore, the users’ passwords are safe even in the case of a server breach. Furthermore, passwords were salted before being hashed, reducing the effectiveness of dictionary attacks using hash or rainbow tables.

All data sent over the internet is encrypted with AES-256, which is more than secure enough especially for an application such as this. The initial symmetric key exchange is secured with RSA-2048, which again is more than sufficient.

### Client-side

#### Connect to server

This objective has been met. The majority of what to say here was covered in the first server-side section of the evaluation. The client will only be unable to connect if the server has reached its maximum number of connections, in which case a suitable error message will be shown to the user. In all other cases, the client successfully establishes communication with the server.

#### Supply username and password to server

This objective has been met. The user can enter their username and password into two text boxes, the contents of which are sent to the server upon clicking login. Certain characters are unable to be entered here for security purposes. This always works and I fail to see a more efficient or user-friendly way of authenticating with the server.

One improvement to this objective that I can see would be a save password feature, where the client can automatically authenticate with the server by storing the username and password on the device. However, this raises issue of how to safely store the password, and would require further research.

#### Send ASCII messages to the server

This objective has been met. The user can enter a message which the client then sends to the server. This functions exactly as expected, though as stated before Unicode should probably have been used.

#### Choose desired partner from list

This objective has been met. By clicking on any friend, the client informs the server of the user’s desired partner. The user can then send messages to their partner. This seems to work very well in practice and ensures the user can only talk to their friends.

There is no selection server-side to check if the client’s request to change partner is valid, that is, the requested partner is friends with the user. It is assumed this is the case because the user can only select a partner from a list of their friends. However, this does mean that if the user could somehow edit a request to the server that they could converse with a user who isn’t their friend. The difficulties in doing this would be identical to those in the server-side’s authentication section.

#### Ability to see if friends are online

This objective has been met. Next to the list of friends is an indicator showing whether each user is online. The indicator is a tick/cross I found in Unicode but I think a green or red coloured indicator would be more user friendly.

Another improvement would be an auto update as to the online status of friends as currently the user has to manually refresh the list by clicking a button. Alternatively, the server could inform all online friends of a user when they either come online or go offline. The client could then notify the user that a friend is now available to talk to.

#### Retrieve messages

This objective has been met, but many improvements could be made. When the user selects a partner, all messages between the two users are automatically retrieved and displayed. The process is almost instantaneous for a small number of messages, though the delay becomes slightly noticeable for extended conversations. This is because multiple transmissions are needed.

To mitigate this effect and reduce the load on the server, messages could be stored offline by the client application. Henceforth, upon selecting a partner, messages need only be read out of a file and the server only needs to send any new messages not stored client-side.

Finally, if the user receives a message not from their partner, currently the client only tells the user that a message has been received, not who from. This becomes more inconvenient as the number of friends a user has increases. Instead a notification could inform the user who sent the message.

#### Search for friends and send friend requests

This objective has been met. The user can enter a username or part of a username and have returned a list of usernames similar. Friends of the user are automatically filtered out, as is the user’s username itself (in case two users had similar usernames, for example). Selecting any of the usernames returned will prompt confirmation from the user after which a friend request will be sent.

The entirety of this process is carried out in a separate form which I deemed necessary to prevent the main form from becoming too cluttered. Overall, this system seems to work well although a different approach may be necessary for a large number of users. During my testing, I noted that there is no way for a regular user to review the progress of their friend requests. I think a brief indicator, for example “pending” on a friend requests not yet reviewed by the admin, would be greatly beneficial.

#### Simple user interface

This objective has been met. I encountered no issues implementing the UI sketches in the design section and as such the UI is simple. Each portion of the main form is labelled, as is the secondary form for searching for friends.

#### Profanity detection

This objective has been met. The string entered by the user is checked against an array of banned words and if any are contained in the string, the message is not sent to the server and the user is informed. This will catch any blatant profanities; however, obfuscation could defeat this method. Furthermore, user input is checked for profanities client-side only.

I can see a profanity filter being important in the context of a school based instant messenger. If students can easily circumvent the filter, then the system could be used for unintended purposes, e.g. cyber bullying.

#### Encryption

This objective has been met. Anything noteworthy has already been covered in server-side: Encryption and Hashing.

#### Group messaging

This objective has not been met. Unfortunately, and primarily due to time constraints, I have not been able to implement the plan for group messaging I laid out in the design section.

I deliberately left this objective for last due to the low priority assigned to it by my client, and with the deadline for this project fast approaching I have decided to leave group chat functionality out of the final product. I may add this for the client at a later date, but for now and for the purposes of this write-up, this instant messenger does not support group messaging.

## Client feedback

This project was tested with the client, Mr Deane, in a school environment with the server and two clients running on three computers on the same subnet. After testing all the features, the client said the following:

Me: What’s good about the project?

Client: All of the core objectives have been successfully fulfilled with the exception of group messaging. I like that the user environment is simple to use which will be useful for younger students. I am pleased that I have to approve all friend requests, which will help ensure the service is used for educational purposes only. New accounts are easy to create and clear in terms of what is required. The admin features are separate from the rest of the functionality and clearly labelled. It’s good to see online and offline indicators for users. The profanity filter seems successful and is quite important in a school environment. The encryption of messages sent over the internet and the hashing of passwords stored on the server should keep student data secure as requested.

Me: What could be improved with respect to the objectives laid out?

Client: I would have liked to see group chats working, so teachers can send homework to a whole class at once, for example. Apart from that everything seems to work well.

Me: Can you think of any useful extensions to the project? These need not be relevant to the objectives.

Client: If students were able to attach files then this project could be used like Google Classroom to upload completed homework. When a user registers, they could have to input their year group which would then decide which facilities they’re allowed. For example, only Year 10s and above can use the upload file feature.

Overall the client feedback seems mostly positive. As Mr Deane said this project achieves its primary aims as covered previously. He also noted the simple UI and admin facilities which would both be beneficial in the academic setting this project was designed for. Mr Deane aptly pointed out the need for encryption early on in this project, and I think it would be unlikely I would have thought to research it without his guidance. It certainly does help secure students’ data and makes this project much more viable for real world use.

Mr Deane accurately pointed out the missing functionality of group messaging and recalling my research this is a staple for nearly all instant messengers. The extensions mentioned seem very well informed, with file uploads particularly taking my liking. I could see file uploads being incredibly useful not only for students but also for teachers distributing resources to a class, for example.

## Potential extensions & improvements

From my evaluation and the client feedback I can see a number of improvements and extensions to this project.

1. **Unicode** – The ASCII character set allows only 256 characters to be used. UTF-16 could allow for up to 65536 characters. Unicode therefore allows many more symbols to be entered, which could be useful in the context of, for example, language students who wish to send symbols from another language. Implementing this shouldn’t require much more than simply changing all occurrences of ASCII in the source code to Unicode.
2. **Timestamp messages** – Storing the date and time a message was sent in the database opens up a range of possibilities. Users could search for messages sent on a certain date to, for example, find a homework assignment set by a teacher. The admin could monitor the times people use the service in order to ensure its appropriate use. The client could sample the time a message is sent and send this to the server, though the famous phrase “Don’t trust user input” probably applies here. Instead, the server could sample the time when they receive the message, which would likely be less than a second difference to the client.
3. **Separate admin application** – Creating a separate admin application, with a separate account, would improve the security of the client application as all admin relating source code could be removed. Logging in as admin could then only be done from the separate application. Mr Deane could have a normal account for interacting with students, and the admin account and application could be distributed to all teachers who wish to use the service. Many more features could be added without cluttering the main client application for Mr Deane. Modifying the client application would be fairly straightforward. The admin application could send a special string, for example, in addition to the credentials entered so the server knows the connection came from the admin application.
4. **Save password** – Many applications which require authentication provide a ‘Remember me’ function where, providing only one user uses the device, upon launching the application the client automatically logs in. This could be useful for students using the client application at home on their own computers. Presumably, the user’s credentials would have to be stored on the device, but then the problem arises of how to encrypt or protect the credentials while still being retrievable by the client. Clearly, more research would be required to implement this.
5. **Save messages** – Currently, each time the user selects a partner all messages between the user and the partner are retrieved from the database and sent to the client. This is hugely inefficient, especially as the number of messages increases. I think messages should be stored client-side to enable quick retrieval of the majority of messages without troubling the server. Upon selecting a partner, all the server would need to do is check for new messages, unless the client is being launched on a new device. Once again, these messages would have to be encrypted, perhaps using the user’s password as a symmetric key.
6. **Auto refresh friends list** – At present, the refresh button exists because the server doesn’t inform the user when there is a change in the status of their friends list. Calling the populate friends method every minute, say, would remove the need for the refresh button but defeat the point of having two way communication with the server. Instead, when a user logs in, their list of friends could be retrieved and any of those friends who are online could be told by the server to update their friends list regarding that one user. Therefore, the populate friends method would only need to be called once, at the start. Thereafter, the status of their friends list would be updated in real time.
7. **Who sent message when not in conversation** – As shown in my testing, if the client receives a message from a friend that the user is not currently in a conversation with, they are not informed who the unshown message is from. This means that a user could have to click through multiple friends to find the one who sent the message. This is easily rectified by either marking the friend who has an unread message or displaying a notification which has the friend’s username.
8. **Status of friend requests** – I think it would be useful, as a user, to see the status of friend requests. Currently, there is no way to see that a friend request has been sent until it is approved by the admin. If, perhaps, unapproved friend requests were shown in the friends list, but greyed out and not available for selection as a conversation partner, then users would know their request is under consideration. This would be quite simple to implement, requiring a little change to the SQL query which retrieves a user’s friends and some changes to the client UI.
9. **Group messaging** – This was a failed objective, and already discussed previously. I will add that nearly all instant messengers support group chats and, as a student myself, they are invaluable for quickly communicating with everyone relevant to a subject. For example, I myself am in a group chat dedicated to Physics questions, whose members are those of my peers studying Physics. Rather than sending a question I have to each of those users separately, everyone receives the message at once, and can contribute together. To implement this, I would use the plan from the design section.
10. **Attach files –** This is a very interesting idea and one I fully intend to implement soon. Having the option to attach files opens up a host of possibilities with regards to, for instance, submitting homework to a teacher, or sending a picture of a question to another student. Some care would be required to ensure only certain file types are allowed to be uploaded, and the question arises of how to store these files server-side, if indeed they should be. As for the actual transmission, I don’t see why a file could not be read in (as in read in the binary which makes up the file), encrypted, and sent to the server. The server can then decrypt the file and write the binary to local storage.
11. **Store more information** – Mr Deane suggested asking for further information upon registration, for example year group. Then different permission could be allocated based on year group. This would require some extra input fields client side and another attribute in the tblFriends table. Every year, the Year attribute of every user could be incremented, and once any users’ Year attribute equals 14, that user can be removed automatically.
12. **Network error handling** – I have implemented very little error handling with respect to received data in this project which I can see being an issue with poor internet connections. I would be very interested to understand how to detect errors and how to handle them in an appropriate manner in the context of a client-server model. Much research would be required to implement this.

## Conclusion

All in all, I think this project is quite effective in providing an academia-oriented messenger. The core functionality is present, and the objectives specific to the school environment this solution is intended for have been met. The source code is, by my standards, well commented and I hope the design section provides sufficient understanding such that my client can make any necessary changes.

# Technical Solution

The source code for this project will be appended to this write-up, firstly the server-side code and secondly the client-side. Some examples of the more complex techniques used are referenced here for ease of the marker, though this list is not by any means exhaustive.

## Contents

|  |  |
| --- | --- |
| **Technique(s)** | **Section(s)** |
| Cross-table parameterised SQL | Server: databaseIO |
| Lists/List operations | Server: databaseIO: recommendFriends, Server: databaseIO: userOnlineRead |
| Queue, Queue operations | Server: ports, Server: distAndAuth: getPort, Server: Main, Server: newThread |
| Dictionary/Dictionary operations | Server: connectionDict, Server: Main, Server: newThread, Server: transmission: communicate |
| Recursive algorithm | Client: Form1: addItemsListView |
| Complex OOP | Server: instances, transmission, distAndAuth |
| Complex client-server model | N/A |
| Simple data model in database | N/A |
| Multi-dimensional array | Server: SHA: hash |
| Exception handling | Server: transmission: communicate, Client: transmission: New, Client: transmission: streamRead, Client: distAndAuth: getPort |
| Complex user defined algorithm | Client: Form1: addItemsListView, Server: databaseIO: recommendFriends, Server: transmission: userOnlineRead, Client: Form1: friendsPopulate |
| Complex use of threading | Server: newThread |
| Encryption/Hashing | Server: RSA, Server: AES, Client: RSA, Client: AES, Server: SHA |