Table of Contents

[Problem Background 4](#_Toc506759360)

[Current System: 4](#_Toc506759361)

[User and Analytical Techniques 5](#_Toc506759362)

[Objectives: 5](#_Toc506759363)

[Detailed Objectives: 6](#_Toc506759364)

[Critical Path: 8](#_Toc506759365)

[Solution Modelling 8](#_Toc506759366)

[Research 12](#_Toc506759367)

[Elliptic Curves 12](#_Toc506759368)

[ECDH 15](#_Toc506759369)

[ECIES 15](#_Toc506759370)

[AES and Block Cipher Modes of Operations 16](#_Toc506759371)

[Sub Bytes 17](#_Toc506759372)

[Shift Rows 18](#_Toc506759373)

[Mix Columns 19](#_Toc506759374)

[Add Round Key 20](#_Toc506759375)

[Key Stretching 20](#_Toc506759376)

[Message Signing 22](#_Toc506759377)

[ECDSA 22](#_Toc506759378)

[HMAC 23](#_Toc506759379)

[CBC-MAC 23](#_Toc506759380)

[Documented Design 24](#_Toc506759381)

[Encryption (ECC, ECDH, ECDSA, HMAC) 24](#_Toc506759382)

[Prime Field Operations 24](#_Toc506759383)

[Representing Points on a Curve 28](#_Toc506759384)

[Point Arithmetic 29](#_Toc506759385)

[Point Addition 30](#_Toc506759386)

[Point Doubling 32](#_Toc506759387)

[Point Multiplication 34](#_Toc506759388)

[Representing a Curve 36](#_Toc506759389)

[A Wrapper for a Predefined Curve 38](#_Toc506759390)

[Elliptic Curve Diffie-Hellman 40](#_Toc506759391)

[Writing the AES algorithm 44](#_Toc506759392)

[AES Encryption 52](#_Toc506759393)

[AES Decryption 56](#_Toc506759394)

[ECEIS 60](#_Toc506759395)

[TCP Design Principles 67](#_Toc506759396)

[Server UI Design 67](#_Toc506759397)

[Design of the TCP server 76](#_Toc506759398)

[Extending the functions of TcpListener 76](#_Toc506759399)

[Managing connected clients 77](#_Toc506759400)

[Interacting with the UI on the Main Thread 80](#_Toc506759401)

[SocketMT Class 82](#_Toc506759402)

[Forwarding from client to client 82](#_Toc506759403)

[Processing a new client connecting 90](#_Toc506759404)

[Starting and stopping the TCP server 93](#_Toc506759405)

[Client functions encapsulation 104](#_Toc506759406)

[Handling client crashes 113](#_Toc506759407)

[Databases 114](#_Toc506759408)

[Tools for databases 114](#_Toc506759409)

[Connecting a user 117](#_Toc506759410)

[Administrator functions 122](#_Toc506759411)

[Saving messages for sending later 127](#_Toc506759412)

[Removing Exit Fors 129](#_Toc506759413)

[Designs of the Client 130](#_Toc506759414)

[Backgrounds 130](#_Toc506759415)

[The UIs 130](#_Toc506759416)

[Login form 130](#_Toc506759417)

[User Settings Form 131](#_Toc506759418)

[The Files Form 134](#_Toc506759419)

[Main UI 136](#_Toc506759420)

[TCP 145](#_Toc506759421)

[The Main UI Updater for Threads 145](#_Toc506759422)

[Heartbeat for the Server 147](#_Toc506759423)

[User Lookup Table 148](#_Toc506759424)

[SendRecieve Class 149](#_Toc506759425)

[Error Handler 172](#_Toc506759426)

[Unit Testing 174](#_Toc506759427)

[Encryption 174](#_Toc506759428)

[EC Point Tests 174](#_Toc506759429)

[EC Point Operations Tests 174](#_Toc506759430)

[Testing Domain Parameters class 175](#_Toc506759431)

[Testing Importing of a Curve 175](#_Toc506759432)

[Testing the Fp Class 176](#_Toc506759433)

[Testing the AES Routines 176](#_Toc506759434)

[Testing ECDH 178](#_Toc506759435)

[Testing ECIES 179](#_Toc506759436)

[Database Tools 181](#_Toc506759437)

[Server Unit Testing 181](#_Toc506759438)

[Testing the client lookup table class 181](#_Toc506759439)

[Testing the client heartbeat 182](#_Toc506759440)

[Testing New TCP Listener 182](#_Toc506759441)

[Testing the UI Updater 182](#_Toc506759442)

[Testing the Client Send and Receive Function 183](#_Toc506759443)

[Testing SocketMT 184](#_Toc506759444)

[Unit Testing the Client 186](#_Toc506759445)

[Encryption Registry Tools 186](#_Toc506759446)

[Testing the UI Updater 186](#_Toc506759447)

[Testing Hearbeat Timer 186](#_Toc506759448)

[Testing the user lookup table 187](#_Toc506759449)

[Testing the Send and Receiving function 187](#_Toc506759450)

[Testing the UIs 189](#_Toc506759451)

[Testing the Server UI 189](#_Toc506759452)

[Testing the Client UI 189](#_Toc506759453)

[Evaluation 191](#_Toc506759454)

[The User’s Thoughts and Concerns 208](#_Toc506759455)

[Problems, Drawbacks and Design Changes 208](#_Toc506759456)

[Bibliography 211](#_Toc506759457)

Project Research

# Problem Background

## Current System:

Of present, the company is limited to the use of Skype and email to communicate internally with other employees working on the shift, managers and other areas of the company. There were also some problems that arose with the system, firstly the use of emails was a clunky way of transferring files, also being insecure. The use of Skype too meant that the company found it hard to manage users and accounts, and it featured no way of transmitting files like email, often causing employees to have to change from skype to email continuously. A solution to this was to create a secure software package that communicated over the internal network in the office that combines the messaging features and security of skype, with an even easier file transfer system like email and easy manageability of accounts and users.

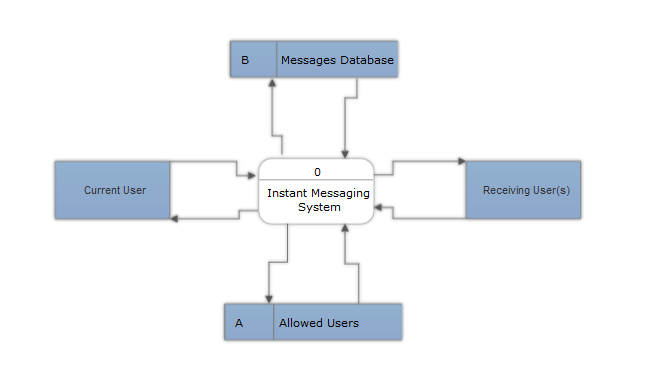


Figure 1: Level 0 DFD of the system to be implemented

As an overview to the new system shown by Figure 1, the current user authenticates with the software enabling him or her different privileges to access the data and write data to their message database (Datastore B), and edit the list of current logons allowed (Datastore A). The messaging system allows then users to send and receive different types of data from one or multiple users in the office (Receiving User(s)) such as files or messages.

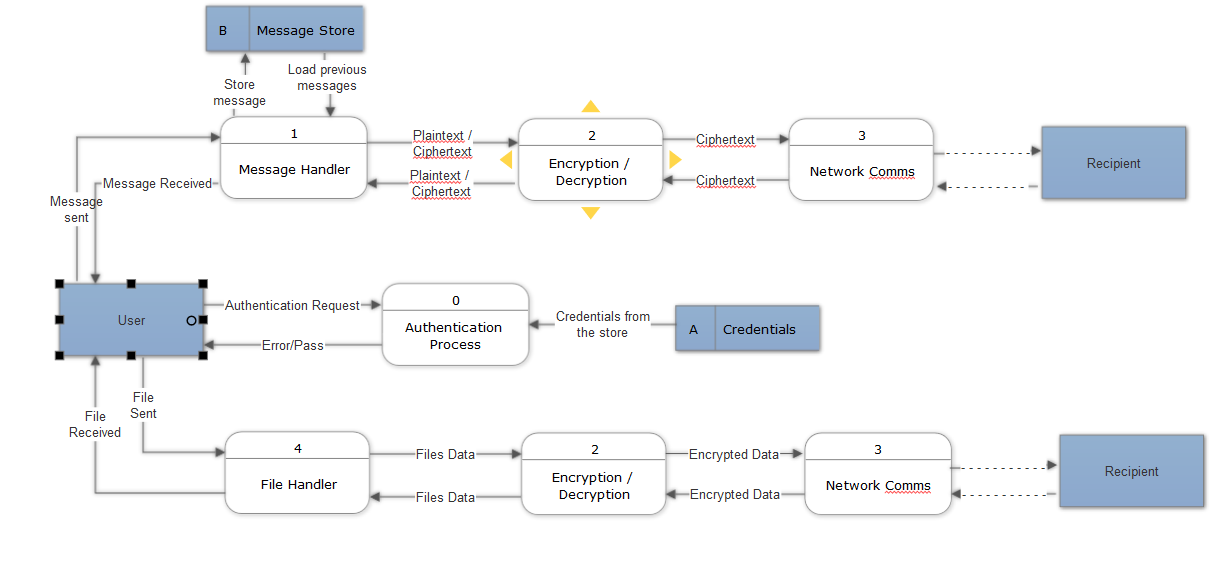


Figure 2: Level 1 DFD

This gives an overview as to roughly how the system will work. It doesn’t include the flow of data from a user changing settings, but that would be included later in a level 3 DFD.

## User and Analytical Techniques

The company have allowed me to use an employee from their company, which provides me with a mechanism to attain information about the current system. It also gives another opportunity for the company to give me some ideas about the contents of the software package and in the finale, the employee will have a chance to test the product and offer his judgment on the project.

Whilst researching the current project I will provide all of my information sources in an appendix section at the back of the paperwork and I will reference these sources throughout the project. The source will be in many different forms, mainly websites, but I will add reference when needed to books, PDFs or other materials that I have used for my research.

# Objectives:

Thinking about the overall design of my program and what should be implemented for the user, I have created a list of measurable objectives to follow over the course of the project. These objectives will be listed below and they will have to be achieved to prove and ensure that I have not missed anything from my project. The objectives for the Instant Messenger application should be:

1. The program should communicate via either a centralised database over TCP/IP or over TCP/IP with a local database for storage of messages
2. The program should be built around a secure messaging service that allows its users to communicate quickly over the local network in the office.
3. It should host some form of encryption either symmetric via XOR or asymmetric using (RSA or ECC)
4. The program should be able to show online and offline users in the company including a way for the current user to authenticate his or herself with the program
5. To differentiate the program from normal IM applications this project should find a way of transferring files to and from different users so that employees may transfer any file they need.
6. Must have a simple user interface that allows simple point and click access to other windows, but not a priority
7. Could have message searching feature allowing users to search previous messages for keywords or phrases

These objectives have to be included in the final application for the project to have met the user’s requirements.

# Detailed Objectives:

The program should communicate via either a centralised database over TCP/IP or over TCP/IP with a local database for storage of messages.

* TCP/IP
  + Get the receiving IP and MAC
  + Setup the stack
  + Open a port to communicate
  + Split the message into packets of X size
* Database (MySQL, Postgres)
  + Install database software on computer if needed
  + Create a new database
  + Password protect/encrypt the database
  + Generic queries to lookup items in the database
  + Possibly authenticate users with a passwords database
* Secure Messaging
  + Authenticate with a server, with a list of known users, local window user accounts or a Microsoft Active Directory server.
  + Setup secure communication channels via signed RSA encrypted messages or via signed ECDH encrypted messages.
    - If using ECDH the domain parameters must be agreed on beforehand and computed public keys should be made available by both parties [1]
    - With RSA the public keys should also be transferred between the two clients
  + Send the messages to a server or local clients inside the network.
    - Open up a port and each client that connects uses something similar to a NAT table or ARP table so each user has a known MAC address or IP for their session, also enabling user identification from client to client.
* Encryption
  + Use of an applicable encryption scheme for how we are transmitting data
  + Possibly use XOR with a key but this means sending the key over the same network so not really a secure encryption type
  + Use of RSA is a better idea yet the security of RSA means a greater bitlength for better security which might pose problems if in the future the program needs to be ported to mobile.
  + The best idea, although very complicated, is to use Elliptic Curve encryption. This form of encryption can use the same curve to not just generate keys to encrypt a message but also digitally sign any data.
    - To use this form, both parties must agree on domain parameters. [1]
    - An algorithm needs to be implemented for point addition, point doubling, and scalar multiplication of an EC point. [1]
    - Data structures and OOP should be used to create objects for these data structures.
* Online and offline users
  + If we are using a dedicated message server, the server can send authenticated users, otherwise each logged on client sends a message similar to an ARP request but to save a DDOS affect it only requests when a message is typing or being sent.
  + If a dedicated message server offline messages can be sent to and stored on the server until a user next logs on
* Sending and receiving files
  + Some form of FTP sending
  + Sockets can also be used to send data from client to server
    - Send a request to open a socket session
    - Setup the client as a server
    - Get the sender to make a connection request to the port on the server
    - The server grants it to one specific user, so some authentication is needed here
    - Send the file
    - Close the connection, save the data, kick the client off

# Critical Path:

The critical path was produced in the form of a Gantt Chart, which displays the start and end times of each of the selected key tasks.

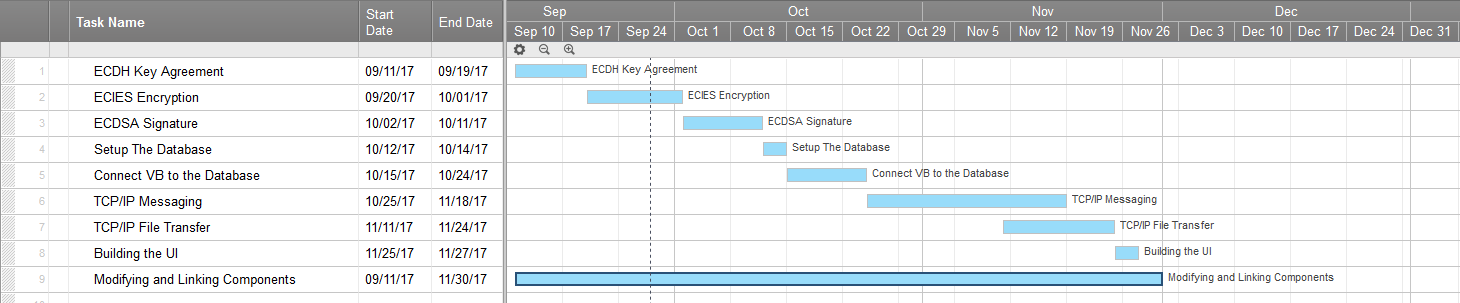
This is the programming critical path:

Figure 3: Programming Critical Path

Most of the work will be done sequentially as there is only one person working on the project, except for TCP/IP Messaging and TCP/IP File Transfer, where TCP/IP will be implemented first of all and then adapted for use with messaging and file transfer. Lastly running the length of the project will be Modifying and Linking components where the systems implemented above will need to be interweaved together to produce the final product.

And this is the overall project’s critical path:

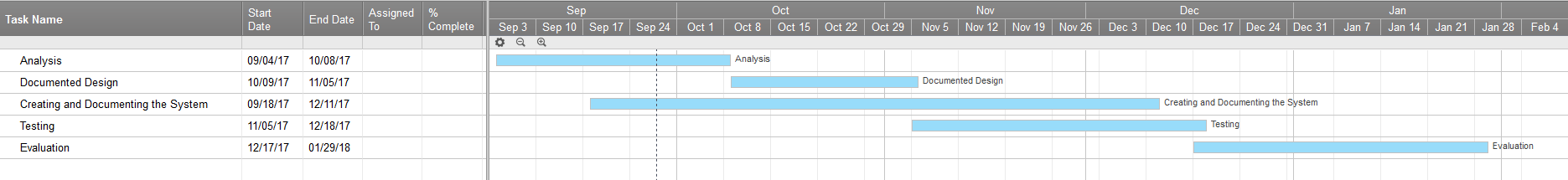


Figure 4: Overall NEA Gantt Chart

This chart shows the structure of work, making certain that the project stays on deadline. Most of the key tasks will be running sequentially except for creation and documentation of the system, which will be running in the background of most of the preparatory work and then running alongside testing which aims to examine the system.

# Solution Modelling

In order to clarify some of the designs of the project that were amassing, I created various diagrams to help me later on in the prototyping and design stage of my project. One of my first systems that I looked at was the database, creating an ER diagram:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | tbl\_users | |  |  | tbl\_messages | |
|  | 1 | PK | userID | 1 | M | FK | userID |
|  |  |  | Username |  |  |  | messageData |
|  |  | FK | permissionsID | 1 |  |  | messageCode |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  | tbl\_passwords | |  |  | tbl\_permissions | |
|  | 1 | FK | userID |  | 1 | PK | permissionsID |
|  |  |  | passwordHash |  |  |  | Message\_Read |
|  |  |  |  |  |  |  | Message\_Send |
|  |  |  |  |  |  |  | File\_Send |
|  |  |  |  |  |  |  | Encrypted |
|  |  |  |  |  |  |  | Admin |
|  |  |  |  |  |  |  | Banned |

Figure 5: EER Diagram

This shows how the tables link together in the database backend of the system, something which I was not confident about. An ideal situation would be for this database system to be running on a server so that the users can be managed on one machine or by an admin logged in with the client. Having the database on a server also allows for messages to be stored encrypted online and if the employees have to change workstations, they would have access to all of the messages previously sent. This database is based around a table of users with the primary key userID which links to the messages it has sent in the messages table and the password hash in the password table. The table of users also has a foreign key to permissionsID which details the user’s current permissions.

I started off with the design of the main UI like so:

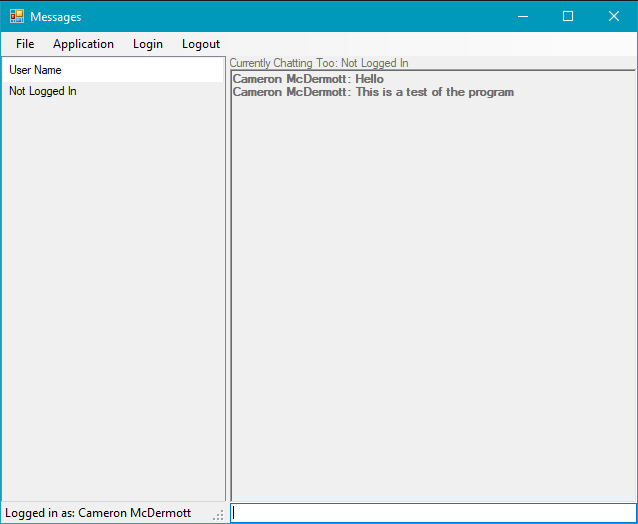


Figure 6: Main window GUI

Which does have some working code to show an example of how the UI will look when it is running correctly. Down the left-hand side, we have the current users that are online and offline which will enable you to select these users and then the messages from them will be displayed on the right-hand side in that large box. At the bottom on the left-hand side we have a status strip which shows the current logged in user and their respective username. And next to that on the right is a text box that allows the user to enter messages to send to the recipient and that message will be shown in the rich text box above it. Above the rich text box is a label which shows what user you are currently chatting to.

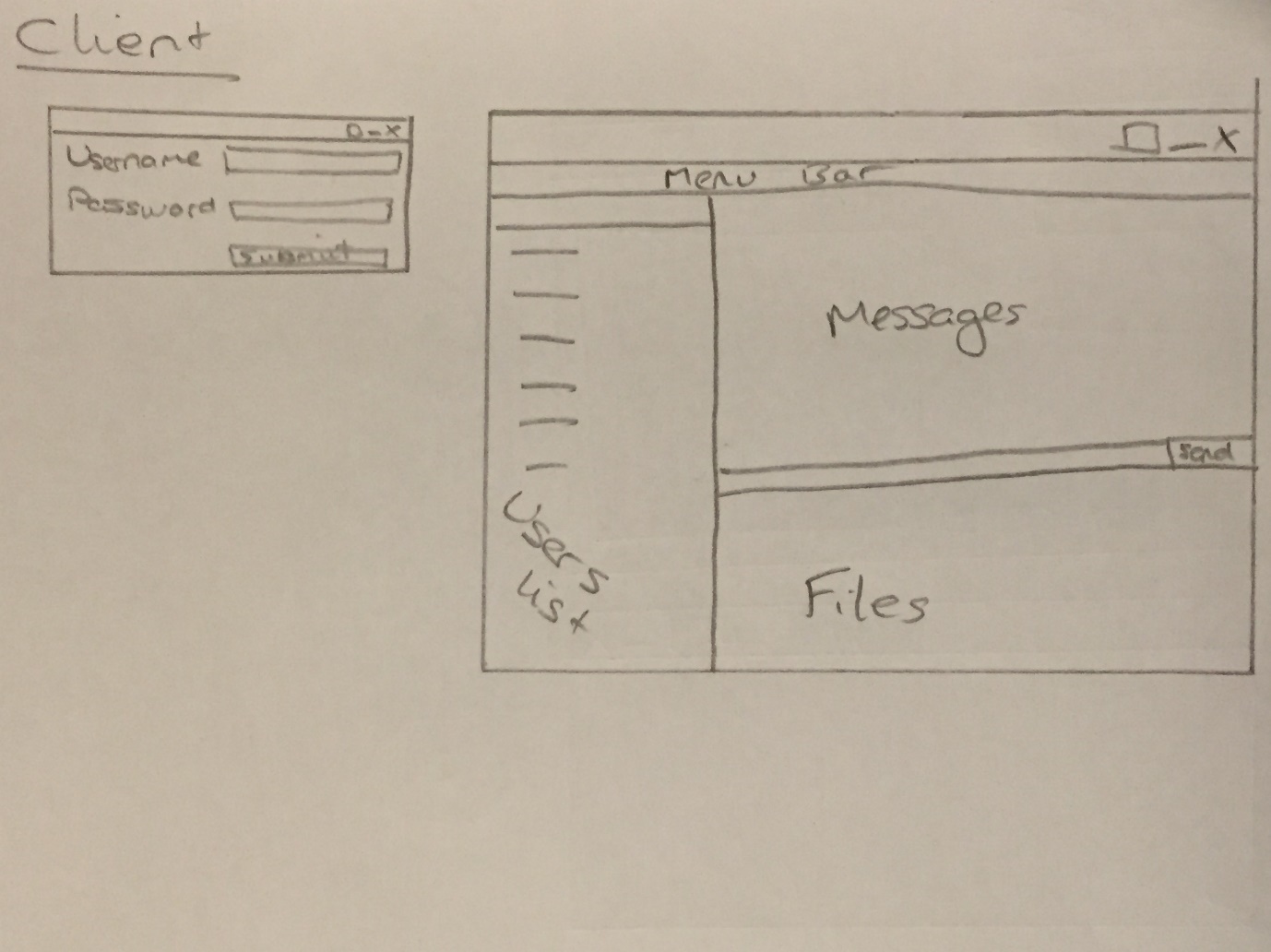


Figure 7: Client paper design

Here shows the initial design for the client, on the left is the login form, very simple design. On the right shows the main UI for the client. Left hand side of the UI shows the user’s list and split with a moveable splitter on the right hand side shows two sections, the bottom would show a windows explorer design for the list of files and above that the Rich text box acting as the message history view box with a small text box below where the user inputs their message and a send button to the right.

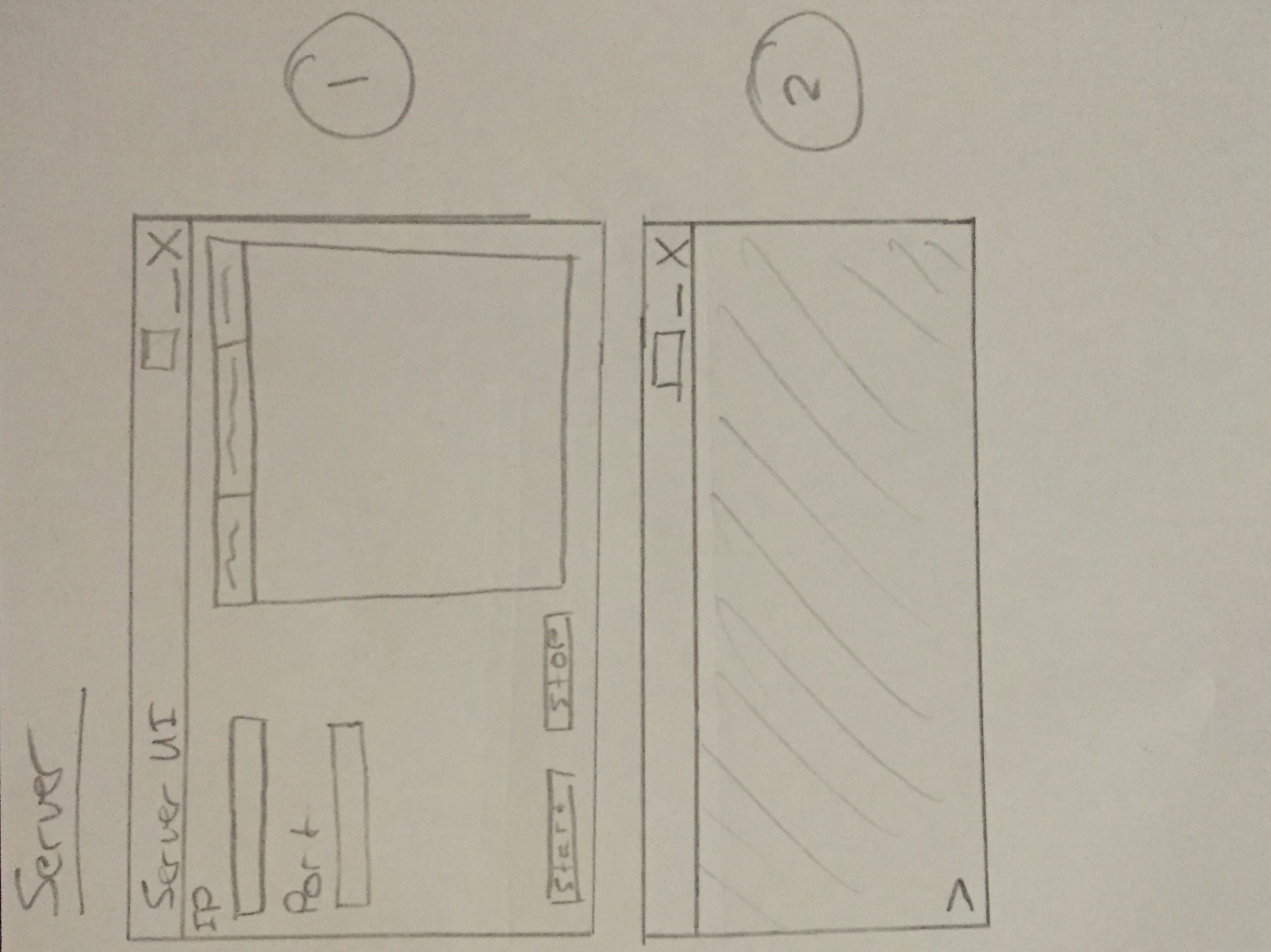


Figure 8: Server UI designs

Above shows off two designs I considered for the server, the top one has the IP and port tex boxes on the left and the list of users and their details on the right with a start and stop button, lower UI is a console window interface with a lower section for entering in commands. I even considered opting to combine the two, the console window showing the log of the server and the other as the main server controls.

# Research

## Elliptic Curves

I started off my research looking by looking for different forms of encryption and came across a form known as ECC or Elliptic Curve Cryptography. Which is predominantly a way of generating public and private keypairs for low powered mobile devices. The reason why this form of encryption is preferable for mobile devices was that it featured components that were very easy to process and also, what attracted me to it, was the fact that for 2048-bit RSA encryption only 256-bit ECC was actually needed which made implementing it much easier. The reason why this system is useful for mobile processors, is that most curves were already pre-computed which left the easy computations to the mobile processors, unless you were calculating your own curves, which was advised against on many forum posts.

There are 3 uses for ECC at the moment which are known as:

1. **ECIES:** Elliptic Curve Integrated Encryption Scheme
2. **ECDH:** Elliptic Curve Diffie-Hellman
3. **ECDSA:** Elliptic Curve Digital Signature Algorithm

Both ECIES and ECDSA use ECDH to generate a set of keys and thus a shared secret from those keys to sign or encrypt the data that is being sent.

ECC forms its basis around curves known as elliptic curves, which feature useful mathematical properties that help calculate keys for encryption. Most curves used in ECC are known as Weierstrass curves and are defined by the equation:

Where:

These curves can be altered forming different shapes on a plane by changing the values of **a** and **b**, yet the curves remain symmetrical about the x-axis. For this form of cryptography, the curves also need to have a point of infinity denoted by the symbol **O**. The reasoning for why elliptic curves are so good for key generation, is the fact that they feature mathematical group properties, so therefore we can define a group over most elliptic curves like so:

* The elements of the group are the points of the curve
* The identity element is the point at infinity
* The inverse of point ***P*** is the one symmetrical about the x axis
* Addition is points that are in a line like so:

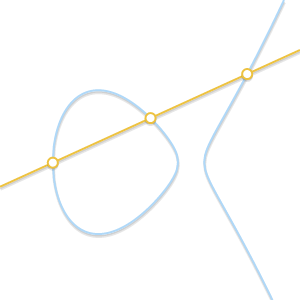


Figure 9: Point Addition on ECC [2]

Due to the fact that groups are defined over elliptic curves means that group arithmetic can be applied to points on a curve. Addition of two of these points results in the negation of the point where the line, defined by the two points, intercepts the curve E, therefore:

Assuming that the curve is in Weierstrass normal form we can calculate addition as:

And finally, if this addition is being performed over prime field then the values of ­­and ­­need to be modulo p.

Calculating point doubling on a curve is similar to point addition but the calculation for lambda requires us to calculate the tangent to the curve at the points.

The last part of group arithmetic is point multiplication, normally achieved by repeated addition of the number until the multiplier is reached. But this method is fully exponential and for ECC it would waste valuable resources and energy on low powered devices. An alternative method to this is known as double and add which uses the functions above to perform multiplication on an elliptic curve point.

This form of multiplication works like so:

If we want calculate the value of we need to convert the multiplier into its binary representation:

So, 180 =

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

And from the binary representation we get 180 as:

180 =

So =

Therefore, the double is based upon the 2 so the exponential is the number of time P is doubled and then each double is added together. [3]

When these curves are used for cryptography it requires some certain values to be present which are agreed before the process is started by both parties. These values are known as domain parameters ad come in the form:

Where refers to the finite field that the curve is defined over where **P** is the prime integer that defines the set of elements of the finite field. Every group and field operation are always modulo the prime (**P**) to keep the values within the set of integers:

The values in the domain parameters refer to the elliptic curve that is defined by the general Weierstrass equation mentioned above. But when working with elliptic curves over prime fields the curve equation must be Mod **P**.

**G** refers to the base point of the curve, , also known as the generator point of the curve because it’s a point that when multiplied should be a multiple of every point on the curve, as elliptic curves are based on a cyclic subgroup. Also refers to the order of the base point where and the value of is usually prime and is also represents the size of the subgroup. The cofactor of this size is represented by which should be less than 5 but most preferably should be equal to 1.

These domain parameters are not generated by the average user due to the fact that this form of encryption is designed for use on mobile devices, and generating the value of requires the number of points on the curve to be calculated which wastes valuable time and resources. Instead several standard bodies publish a list of pre-calculated domain parameters known as standard curves. The only problem with this approach is that these standard bodies can engineer the curves to have predictable calculations by using secret values only known by them, which NIST were recently accused of doing because of their link with the NSA. So now most pre-determined parameters feature a 7th value known as or a seed. The significance of this is that it takes a “nothing up my sleeve” approach to proving the generation of these parameters were done on a legitimate basis. [4] [5]

## ECDH

ECDH is a method used to generate a public and private keypair and is a way of exchanging these keys between different parties. However, this is not the actually method of encryption which is known as ECIES. ECDH must be performed by both parties and must follow this sequence:

1. Firstly, the keypair has to be generated by each party.
2. Next the public keys must be exchanged between parties
3. Finally, the parties must both generate a shared secret for encryption

To generate the private key each party must generate a random integer between 1 and n – 1 where n is the order of the subgroup.

To generate the public key the base point or generator point must be scalar multiplied by the value of the private key.

Next the public keys are transferred from party to party over an unencrypted channel, if the public keys are intercepted it is intractable to try and calculate the private key from those keys. These public keys once received by the other party are used to generate a shared secret by multiplying the received key with their generated private key. If all is correct the shared secrets should be equal and thus do not need to be exchanged, the shared secret can then be used to encrypt data that is being sent. [6]

## ECIES

Elliptic Curve Integrated Encryption Standard otherwise known as ECIES uses the keys that were generated in the steps above to encrypt data and specify the layout that data should assume when being sent over the internet. The data that is to be encrypted is passed into this system along with the public key of the recipient, or if using ECDH, the x-value of the shared secret.

The system relies on the fact that the recipients have already calculated keys and that the sender is in possession of their public key in order for them to generate a shared secret using ECDH. The shared secret that was produced via ECDH is then passed to a key derivation function (KDF) which can come in different forms. The KDF function was initially developed to produce a key that has desirable properties, therefore avoiding weak keys. The common use of these functions is to provide a secure password hashing method using a known salt to produce a hash of a password for storage or verification inside the passwd file on Linux systems or the SAM file that exists under the Windows directory C:\Windows\System32\config\. The system that ECIES uses to extend the secret key into two keys without worrying about security is to append the hex value 0x01 onto the end of the original secret key and hash that, then append a 0x02 onto the end of the original secret key and hash that again producing two different keys from the secret key provided.

Next, one of the keys is selected for use with an encryption function, depending on the encryption function, it might require a mode of operation. The mode of operation is usually specific to block ciphers like AES as the encryption algorithm only works on a set number of bits, also known as a block, hence the name block cipher. Using this encryption algorithm produces a ciphertext when provided with a key and the plaintext, depending on the mode of operation an IV can be used alongside the cryptographic function to make it more secure.

Finally, the ciphertext, IV, senders public key and the second value produced by the KD function (KDF) are used together to create a message authentication code (MAC) using a function like HMAC. The MAC function provides a signature for a message, so the when the recipient receives the message they can calculate their version of the signature and check to see if the two values equal. If this is not the case the message is deemed as invalid and can be discarded. [7]

The values for the ciphertext, the IV, the senders public key and the MAC can all be sent to the recipient ready for decryption. [8]

## AES and Block Cipher Modes of Operations

As mentioned above, ECIES requires A mode to operate so that it can process more than 16 bytes of data. I looked through the Wikipedia page [9]and decided that I would use the cipher block chaining mode to encrypt data longer than 16 bytes. There were other methods present to use but it was suggested to use this method by a colleague of mine who has a good knowledge of encryption. This method also provides the better data protection mechanism than ECB for example. Due to the fact that you XOR the starting 16 bytes with a cryptographically secure random selection of 16 bytes eliminates any possible chances to spot patterns inside the encryption which could possibly give away the data that was encrypted. For example

Figure 10: Left: Original Image, Middle: Encrypted with AES and ECB, Right: Encrypted with AES and CBC [9]

As illustrated by the diagrams above you can see that whilst using the ECB as the mode of operation it encrypts the original image correctly but with a big dataset it is possible to spot the original pattern that forms the image, thus it is potentially possible to calculate the decryption from that value.

Encryption with CBC is fairly simple, it requires that an IV and a key has been generated and a non-zero message to be encrypted. To encrypt the data that the mode of operation passes to it does require a compatible block cipher algorithm that has functioning encryption and decryption routines. The algorithm for CBC encryption is as follows:

* The first n bytes of the message are selected. Where n is equal to the number of bytes that a block is, so for example AES requires 16 bytes.
* Next an IV is generated and must be cryptographically secure so as it is impossible to predict what the IV could have been and the IV must also be the length of n bytes
* The IV and the n bytes of message must be XORed together and passed into the block cipher along with the key used for encryption.
* The block cipher returns a ciphertext and that text must be XORed with the next n bytes of the message before encryption. [9]

Whereas CBC follows this pattern for decryption:

* The first 16 bytes in the ciphertext are selected and passed to the block cipher along with the key used for encryption in order to decrypt the ciphertext
* If the first set of 16 bytes was decrypted, XOR the IV with the plaintext, else XOR the previously decrypted ciphertext with the newly produced plaintext and store it.

AES is a secure method to modify the data in a way that is secure, yet possible to reverse the change with the right tools. AES requires two things before starting, a set of 16 bytes of data to encrypt and a key of length 128, 192 or 256 bits. The algorithm must first stretch the key into enough bytes to be used in the encryption process, also known as the round keys, some of the features of the encryption is dependent upon the length of key provided. As the block size is normally restricted to 128bits (but can also be 192 or 256 bits) the size of each round key must be equal to the block size being used. The core of the encryption is repeated over a certain number of rounds less one as the last round is slightly different from the previous, where the number of rounds is determined by the key length so the number of rounds 10, 12 or 14 is respectively represented by the key sizes 128, 192 and 256.

Each round is a composition of 4 steps:

1. Sub Bytes- A byte lookup and substitution scheme using Rijndael’s S-box
2. Shift Rows- cyclically shifts the last three rows in the state
3. Mix Columns- another substitution but this time over the Galois Field
4. Add Round Key- XOR the state with the round’s key

When AES is provided with the data to be encrypted, 128 bits of this is selected by the block cipher mode and that 128-bit selection of the data must be transformed into a state which is a 2-dimensional representation of the data like so:

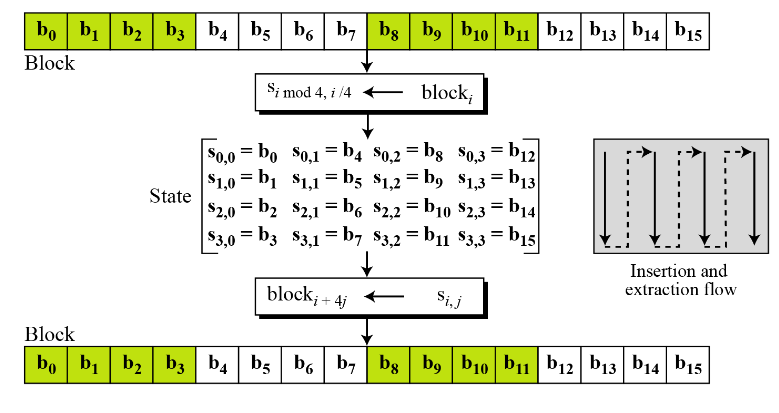


Figure 11: Block, State transformation [10]

### Sub Bytes

This method of substitution uses two tables for encryption and decryption, for decryption one table is the reverse of the other so that the substitution can be undone. The tables look like so:

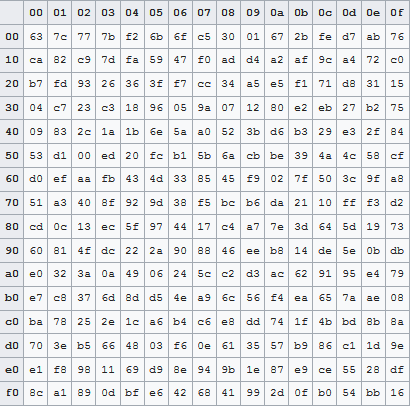


Figure 12 Rijndaels S-Box [11]

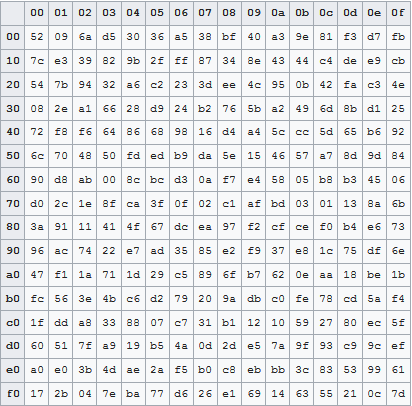


Figure 13: Rijndaels Inverse S-Box [11]

The substitution is done on a byte by byte basis and each byte is converted to hex and the left hex value selects what row is used and the right hex value selects what column. The value from the table then replaces the value used to look it up. [11]

### Shift Rows

In AES, another transformation applied to the state comes in the form of a cyclic bit shift on each row of the state. The first row must be left unchanged but the second, third and fourth row are shifted left 1, 2 and 3 respectively.

To reverse this operation, the same rules are applied to the state apart from each value must be shifted to the right by a number of positions.

### Mix Columns

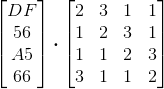
In this step each byte in a column is combined using a linear transformation by multiplying each value with a transformation matrix and then XORing the rest of the value like so:



Figure 14: Transformation Matrix [12]

The multiplications done in Mix Columns must be kept inside the Galois Field, with the same representation used in SBox operations, . Each output element is calculated like so:

Each column is worked on individually and each byte in that columns row is used to calculate every output. In this notation **S** stands for the given state column, **M** is the transformation from above, **B** is the output column to replace the one in the state. The value of **r** determines the current row we are working on and when the last row is reached in the state we must cycle back round to the first row and so on. For example, calculating the result of the first line would be:



Each bracket is added together but due to the fact that the calculation is done within the XOR operation is used which acts like addition when used in conjunction with multiplication.

There is an alternative method to calculate each multiplication which reduces the computation needed, reducing the time to calculate the values of each bracket.

Any value multiplied by the 01 will stay the same as any value multiplied by 1 will always be the same as it was before.

Values that are multiplied by 02 can be calculated by shifting the value left by one position and if the high bit is set to one we exclusive-or the value with the hexadecimal 1B.

Calculating the multiplication of a value with 03 is easier when the calculation of 02 using left shifts is implemented as:

The brackets containing the can be exchanged for the method mentioned earlier to calculate 2 times a value. This means that we can now calculate the sum above like so:

So finally, we can reach the answer by XORing the values of A5, FA, A5, 66 which gives us the final value of 96 which is the same value as seen above using the longer method. [13]

This transformation must be applied to each column in the state in order to change every byte. The method to reversing this uses the same arithmetic as before but must be applied to a different transformation matrix, shown below.

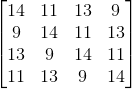


Figure 15: Inverse Transformation Matrix [3]

Unlike before there is no easy way to simplify any of these calculations so the multiplication must be calculated over a Galois field of 256. [12]

### Add Round Key

The last transformation that is needed for AES is known as “Add Round Key”. This is a simple XOR function that takes the current state and applies the current round key via XOR. This is reversed by applying the same round key again to the encrypted state which should return the unencrypted state.

### Key Stretching

This is the initial process performed once the algorithm is given the encryption key for that session. The purpose of this is to lengthen the original key into multiple keys of 128 bits and depending on the key size the number of keys that need to be formed differs, for 128bit keys we need 10 keys + 1 as the 10 are used for the rounds which will be explained later and the +1 is used before the rounds start, for 192bits 12 keys +1 are needed and 256bits 14 keys + 1 are needed. The plus one is normally the given key and the rest are generated for use in the rounds. The method to generate the keys uses the Rijndael key schedule algorithm which expands a shorter key into a number of round keys, there are some common operations that are used in the procedure:

* **Rotate**- This is a cyclic bit shift to the left where the hexadecimal is rotated 8 bits to the left so each hex pair moves one place to the left like so:

Before: **AE DF 38 FF**

After: **DF 38 FF AE**

* **Rcon-** This is the exponentiation of 2 to a user specified value within the Galois field, there is a lookup table for values between 0 and 255, enough needed for working with 8-bit hex values.
* **SBox-** This is the same used in AES Sub bytes.
* **Key Schedule Core-** This operation follows the process:
  1. The input is 4 8-bit values that forms a 32-bit word which produces the same layout output
  2. Next the rotate operation is applied to the input
  3. Next SBox is applied to each byte in the input
  4. Lastly each byte is XORed with a value from the Rcon lookup table determined by a counter.

The procedure used to extend the key follows a simple algorithm but firstly requires some constants to be set, the values of **n** is 16 for 128bit keys, 24 for 192bits and 32 for 254bit keys and the value of **b** is 176 for 128bits, 208 for 192 bits, 240 for 256 bits. The process of stretching looks like so:

* The **n** bytes, starting from byte 0 of the given key, is the first **n** values of the stretched key
* The Rcon iteration value must be set to 1 as the first byte is not used
* While we have less than **b** bytes of we generated **n** more bytes of the new keys like so:
  + The next 4 bytes of the key is generated:
    - The last 4 bytes of the stretched key is set in a variable called **t**
    - The **t** values are passed to schedule core with the value of Rcon iteration
    - Rcon iteration is increased by 1
    - **t** is XORed with the four bytes taken to produce **t** in the first point
  + These steps are repeated 3 times:
    - To create the 12 new bytes, we take the previous 4 bytes in the stretched key
    - XOR those bytes with the bytes **n** bytes before the new stretched key
  + Next is just for a 256bit key
    - Take the previous 4 bytes of the stretched key and run them through Rijndaels SBox
    - These 4 bytes are then XORed with the previous 4 bytes of the stretched key
  + If working with a 128bit key these steps are not done, a 192bit key this is run twice and three times for a 256bit key
    - The previous 4 bytes of the key are XORed with the four-byte block **n** bytes before the new stretched key.

Each time the 4 bytes are XORed with the previous 4 bytes the output forms the next 4 bytes in the stretched key and that forms the new set of round keys. [14]

Putting all of the AES steps above together we get a system that should encrypt and decrypt data like so:

For encryption the overall algorithm is comparable to:

**Inputs:** The encryption key of length 128, 192 or 256 bits and the state of size 128 bits.

**Output:** Produces a cipher text that can be in the form of a 128-bit state.

**Method:**

1. The inputted encryption key is expanded into enough round keys determined by the key length, via Rijndael key schedule.
2. The first key in the expanded key is taken and is XORed with the 128-bit state (AddRoundKey).
3. The next part is repeated the (Number of rounds) -1
   1. SubBytes
   2. ShiftRows
   3. MixColumns
   4. AddRoundKey
4. The final round looks like so:
   1. SubBytes
   2. ShiftRows
   3. AddRoundKey

For decryption the process follows this method:

**Inputs:** The encryption key of length 128, 192 or 256 bits and the encrypted state of size 128 bits.

**Output:** Produces a plain text that can be in the form of a 128-bit state.

**Method:**

1. The inputted encryption key is expanded into enough round keys determined by the key length, via Rijndael key schedule.
2. AddRoundKey
3. Inverse ShiftRows
4. Inverse SubBytes
5. Next is repeated (Number of rounds) -1
   1. AddRoundKey
   2. Inverse MixColumns
   3. Inverse ShiftRows
   4. Inverse SubBytes
6. Finally, AddRoundKey is performed

With the decryption the rounds keys are used up backwards so if you were to place each key in its own array index, encryption uses the keys from 0 to (Number of rounds) + 1 and decryption uses the keys from (Number of rounds) +1 down to 0.

The Block Cipher Mode of Operation can then take the 128bit state produced by AES and convert it to a byte array and depending on the length of the message passed to it, concatenate each of these arrays together producing the final output. [15]

## Message Signing

### ECDSA

This form of message signing uses the same elliptic curve used for the encryption but instead a random value must be generated each time so the value of the private key from the encryption session cannot be found. Like before the parties must decide upon the curve and the domain parameters, especially the value of ***n***, which is used in the ECDSA algorithm. The method to generated the signature of the message is done like so:

1. Firstly, calculate the hash of the encrypted message using a suitable hash function, preferably SHA256
2. Then take the ***N*** number of leftmost bits of the hash, where ***N*** is the bit length of ***n***
3. Generate a random number and must be within ***1*** and ***n-1***, this value is then multiplied with the base point ***G*** to give the point with an x and y coordinate, the ***x*** value is then used to calculate value of ***r*** like so:
4. Finally, ***s*** is calculated ***z*** to be the leftmost bits of the hash, ***dA***  as the private key used for encryption, ***k*** as the value of the random integer, so ***s*** is found:

Verifying the signature is also very easy and requires the encrypted message and the value of ***s*** and ***r*** generated by the signing algorithm.

1. Firstly, calculate the hash of the encrypted message ***e*** using a suitable hash function, preferably SHA256
2. Then take the ***N*** number of leftmost bits of the hash, where ***N*** is the bit length of ***n***
3. Calculate the value of ***w*** as ***s***-1 ***MOD n***
4. Next find two integers like so:
5. Find the value of the point ***P*** by multiplying ***u1*** by the base point ***G*** and add onto it the result of ***u2*** multiplied by the public key of the sender.
6. The ***x*** value of ***P*** should equal ***r MOD n***

[16]

### HMAC

This form of message signature or MAC uses a cryptographic hash function along with a secret cryptographic key. The cryptographic algorithm used is normally appended to HMAC, for example if using SHA256 then the name will be HMAC-SHA256. HMAC follows a two-step method to try and prevent length extension attacks by deriving the secret key into an inner and outer key then, using the inner key, a hash is produced using the inner key and the message, this hash is then used along with the outer key to derive a HMAC code. [17]

### CBC-MAC

This method to generate a message authentication code uses the block cipher mode of operation mentioned above and uses a block cipher like AES to create a MAC for the data passed too it. When using CBC-MAC only the last block of encrypted message is needed rather than the same number of blocks generated when encrypting. To validate the message after being received the same process is used to generate the MAC and that can then be compared to the one sent.

There are some known attacks against this method of generating a MAC compared to using HMAC or something equivalent. The first known vulnerability is when using a variable length message when encrypting, it allows an attacker to append their own message to the end of an already encrypted message and generate a valid MAC for the new message. The way to mitigate this risk is to prepend the cipher text length to the start of the cipher text before generating the MAC of the text.

Another known bad example is to use the same key for message signing as used for encryption, this presents a possibility for the attacker to regenerate the private key used for the encryption and signing. This risk is bypassed when generating a valid shared secret, because the shared secret provides two keys, one for use whit the encryption algorithm and one for use with the signing application, therefore in effect using two different keys that both parties are reliably able to generate. Thus, therefore mitigating the risk of regenerating the key and being able to decrypt the cipher text too and because the two shared secret keys are generated using hashing, which is a one-way function, it is impossible to determine what the pre-hashed keys were, therefore making it a safe alternative. [18]

# Documented Design

## Encryption (ECC, ECDH, ECDSA, HMAC)

### Prime Field Operations

I started off by creating a class to contain all of the useful field operations that could be required to generate a curve. Later on, in my research, I found a paper advising the programmer to use a list of pre-defined curves, such as those from Brainpool or NIST, alas the prime number generation was never needed for the program as later a class was created to read in the values of a curve from an xml file. The method for generating a prime follows the Miller-Rabin prime tests, confirming the primality of a prime with a specific accuracy of the test. Although generating a prime number was never later used, the prime tests are used when setting the prime in **sub New**, as a method of error checking the provided details.

Imports System.Numerics

Imports System.Security.Cryptography

Imports System.Threading

Namespace Fp\_Operations

Public Class Fp

Public p As BigInteger

Private \_\_a\_gen As BigInteger

Private \_\_max As BigInteger

Private \_\_prime As BigInteger

Public Sub New(ByVal p As BigInteger)

If Test\_prime(p, 20) = Errors.MR\_P\_prime Then

Me.p = p

Else

Throw New Exception("Specified parameter should be prime")

End If

End Sub

Public Sub New(Num\_Bytes As Integer)

Generate\_Params(Me.p, Num\_Bytes)

End Sub

''' <summary>

''' A list of custom errors used in the private methods

''' </summary>

Private Enum Errors

MR\_P\_LESS\_2 = -100

MR\_P\_MOD\_2\_ZERO = -101

MR\_P\_NOT\_prime = -103

MR\_P\_prime = 100

End Enum

''' <summary>

''' So far only generates the domain p which is specifically a prime number for which a prime filed will later operate over

''' </summary>

Public Sub Generate\_Params(ByRef p As BigInteger, ByVal Num\_Bytes As Integer)

Dim P\_ret As Boolean

Do

P\_ret = Generate\_P(p, 40, Num\_Bytes)

Loop While P\_ret = False

End Sub

Private Function Generate\_P(ByRef p As BigInteger, ByVal k As Integer, ByVal Num\_Bytes As Integer) As Boolean

Dim Temp\_\_prime As BigInteger

Do

Temp\_\_prime = New BigInteger(SecureRandom(Num\_Bytes))

If Temp\_\_prime.IsEven = False Then

If Test\_prime(Temp\_\_prime, k) = Errors.MR\_P\_prime Then

p = Temp\_\_prime

Return True

End If

Return False

End If

Loop While Temp\_\_prime Mod 2 = 0

Return False

End Function

''' <summary>

''' Generates a random binary value of lenght Num\_Bytes in a byte array

''' </summary>

Private Function SecureRandom(ByVal Num\_Bytes As Integer) As Byte()

Dim RND = New RNGCryptoServiceProvider()

Dim RawRandomBytes(Num\_Bytes) As Byte

RND.GetBytes(RawRandomBytes)

Return RawRandomBytes

End Function

'''<summary>

''' This function uses the Miller-Rabin prime tests as stated on wikipedia where p is the odd number to be tested and k is the accuracy of the test

''' </summary>

Private Function Test\_prime(ByVal p As BigInteger, ByVal k As Integer) As Errors

Dim \_\_Thread As Thread

Dim s As BigInteger

Dim d As Integer

Dim a As BigInteger

If p < 2 Then

Return Errors.MR\_P\_LESS\_2

End If

If p <> 2 And p Mod 2 = 0 Then

Return Errors.MR\_P\_MOD\_2\_ZERO

End If

\_\_max = p - 1

\_\_prime = p

\_\_Thread = New Thread(AddressOf Gen\_Random\_BigInt)

\_\_Thread.Start()

s = p - 1

d = 0

While s Mod 2 = 0

s = s / 2

d += 1

End While

For i = 0 To k - 1

Dim x As BigInteger

If \_\_Thread.IsAlive Then

\_\_Thread.Join()

End If

a = \_\_a\_gen

\_\_Thread = New Thread(AddressOf Gen\_Random\_BigInt)

\_\_Thread.Start()

'a = Gen\_Random\_BigInt(p.ToByteArray.Length - 1, p)

x = BigInteger.ModPow(a, s, p)

If x = 1 Or x = p - 1 Then

Continue For

End If

For q = 1 To d - 1

x = BigInteger.ModPow(x, 2, p) 'ModPow (a \* b) mod c

If x = 1 Then

\_\_Thread.Abort()

Return Errors.MR\_P\_NOT\_prime

End If

If x = p - 1 Then

Exit For

End If

Next

If x <> p - 1 Then

\_\_Thread.Abort()

Return Errors.MR\_P\_NOT\_prime

End If

Next

\_\_Thread.Abort()

Return Errors.MR\_P\_prime

End Function

''' <summary>

''' Genetares a random BigInteger value that should be positive and less than or equal to the value of prime being passed

''' </summary>

Private Sub Gen\_Random\_BigInt()

Dim int As BigInteger

Dim rng As New RNGCryptoServiceProvider

Dim bytes(\_\_max.ToByteArray.LongLength) As Byte

Do

rng.GetBytes(bytes)

int = New BigInteger(bytes)

Loop While int < 2 Or int >= \_\_prime

\_\_a\_gen = int

End Sub

''' <summary>

''' Calculates field addition

''' </summary>

Public Function Field\_Addition(ByVal A As BigInteger, ByVal B As BigInteger, ByVal P As BigInteger) As BigInteger

If P = 0 Then

Return New BigInteger(-999)

End If

Dim Q As BigInteger

Q = New BigInteger(((A + B) Mod P).ToByteArray)

Return Q

End Function

Public Function Field\_Subtraction(ByVal A As BigInteger, ByVal B As BigInteger, ByVal P As BigInteger) As BigInteger

If P = 0 Then

Return New BigInteger(-999)

End If

Dim Q As BigInteger

Q = New BigInteger(((A - B) Mod P).ToByteArray)

Return Q

End Function

Public Function Field\_Mult(ByVal A As BigInteger, ByVal B As BigInteger, ByVal P As BigInteger) As BigInteger

If P = 0 Then

Return New BigInteger(-999)

End If

Dim Q As BigInteger

Q = New BigInteger(((A \* B) Mod P).ToByteArray)

Return Q

End Function

Public Function Field\_Div(ByVal A As BigInteger, ByVal B As BigInteger, ByVal P As BigInteger) As BigInteger

If P = 0 Then

Return New BigInteger(-999)

End If

Dim Q As BigInteger

Q = New BigInteger(((A \* Point\_Operations.Inverse(B, P)) Mod P).ToByteArray)

Return Q

End Function

End Class

End Namespace

The bottom 4 functions, named **Field\_...** were never needed for generating elliptic curves or primes or the encryption side of the program.

### Representing Points on a Curve

In order to deal with the size of the values that 256-bit elliptic curve encryption uses we needed something more than int64 to store these values. A good contender was **System.Numerics.BigInteger**, it features some of mathematical functions already implemented for the **BigInteger** class, therefore it seemed a better idea than writing my own **BigInteger** class.

Setting the x value y value is a simple process and, its possible to set the prime number the point was generated over, but not necessary:

Private \_\_x, \_\_y, \_\_p As BigInteger

    Public Sub New(Point As ECPoint)

        Me.x = Point.x

        Me.y = Point.y

        Me.p = Point.p

    End Sub

    Public Sub New(X As BigInteger, Y As BigInteger, P As BigInteger)

        Me.x = X

        Me.y = Y

        Me.p = P 'Could add in prime testing from Fp Operations

    End Sub

    Public Sub New(X As BigInteger, Y As BigInteger)

        Me.x = X

        Me.y = Y

    End Sub

    Property x() As BigInteger

        Get

            Return \_\_x

        End Get

        Set(value As BigInteger)

            \_\_x = value

        End Set

    End Property

    Property y() As BigInteger

        Get

            Return \_\_y

        End Get

        Set(value As BigInteger)

            \_\_y = value

        End Set

    End Property

    Property p() As BigInteger

        Get

            Return \_\_p

        End Get

        Set(value As BigInteger)

            \_\_p = value

        End Set

    End Property

There are three overloaded versions of sub **New** where each one specifies different values that the point can use to create it, the first one allows us to copy the value in another instance of **ECPoint** thusly creating a duplicate instance of the point. The other two allow us to set the X and Y values one allowing the prime to be specified and the other without the prime being specified.

There are some other checking functions that allow us to check if the value of the point is correct or if it equals some value that we intend:

''' <summary>

''' Gets if the point is an infinity point

''' </summary>

Public Function IsPointInfinity() As Boolean

    If x = 0 And y = 1 Then

        Return True

    End If

    Return False

End Function

''' <summary>

''' Zero out an ECPoint for security Reasons

''' </summary>

Public Sub Zero()

    x = BigInteger.Zero

    y = BigInteger.Zero

    p = BigInteger.Zero

End Sub

''' <summary>

''' Checks that the x or y values are set to zero

''' </summary>

''' <returns></returns>

Public Function IsNull() As Boolean

    If Me.x = 0 Or Me.y = 0 Then

        Return True

    End If

    Return False

End Function

''' <summary>

''' Check if the point lies on a specific curve, checked by the general Weistrass Equation (x^3 + ax + b - y^2) % p == 0

''' </summary>

Public Function IsPointOnCurve(ByVal Param As Curve.Domain\_Parameters) As Boolean

    Return ((x \* x \* x) + (Param.a \* x) + Param.b - (y \* y)) Mod Param.Fp.p = 0

End Function

''' <summary>

''' Returns a boolean value if the x y and p values are set to the value -99

''' </summary>

''' <returns></returns>

Public Function isErroneous() As Boolean

    If x = -99 And y = -99 And p = -99 Then

        Return True

    End If

    Return False

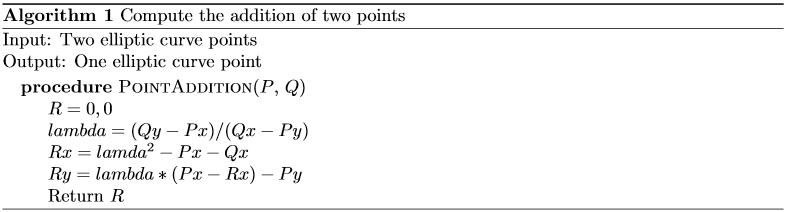
End Function

The first function checks if the point is equal to the point at infinity (0,1), returning true if it is or returning false if not. The next procedure sets the values of the point to zero, so that the memory is overwritten as a safety feature when dealing with the generation of keys for the encryption, it could be possible for an attacker to heap spray the program when the point has been deallocated, returning the value of the public key to the attacker. **IsNull** checks if the value of the point is equal to 0 and **isErroneous** checks if the values are all equal to -99, set when an error occurs in point arithmetic. Lastly the error checking function **IsPointOnCurve** can be called to calculate weather the point is declared on the curve defined by **Domain\_Parameters**. This takes the Weierstrass equation of the curve and checks if it equals zero.

### Point Arithmetic

When dealing with elliptic curves there are three different operations we need to perform on the points to translate them around the curve. The mathematics behind these operations are detailed on page 11. But these equations need to be transformed into some pseudo code like the below:

#### Point Addition



This algorithm was then transformed into some visual basic code:

Public Shared Function Addition(ByVal P1 As ECPoint, ByVal P2 As ECPoint) As ECPoint

    Dim P3 As New ECPoint(P1)

    Dim l, z As BigInteger

    Dim D As BigInteger = P2.x - P1.x

    If D < 0 Then

        D += P1.p

    End If

    z = Inverse(D, P1.p)

    l = ((P2.y - P1.y) \* z) Mod P1.p

    P3.x = ((l \* l) Mod P1.p - P1.x - P2.x) Mod P1.p

    P3.y = ((l \* (P1.x - P3.x)) Mod P1.p - P1.y) Mod P1.p

    Return P3

End Function

This was a very basic function, and the division to calculate lambda has been replaced by an inverse function, which calculates the inverse of a number so that the faster multiplication can be used in our program, therefore making the program much faster.

''' <summary>

''' A structure that contains data returned from Extended\_GCD function

''' </summary>

Friend Structure RetGCD

    Public x As BigInteger

    Public y As BigInteger

    Public GCD As BigInteger

End Structure

''' <summary>

''' Calcualtes the Greatest Common Divisor of two numbers

''' </summary>

Private Shared Function Extended\_GCD(ByVal a As BigInteger, b As BigInteger) As RetGCD

    'The pseudocode was found on https://en.wikipedia.org/wiki/Extended\_Euclidean\_algorithm#Pseudocode

    'Tested with the site http://planetcalc.com/3298/

    Dim r, o\_r As BigInteger

    Dim t, o\_t As BigInteger

    Dim s, o\_s As BigInteger

    Dim p, q As BigInteger

    Dim ret As RetGCD

    s = 0

    t = 1

    r = b

    o\_s = 1

    o\_t = 0

    o\_r = a

    While r <> 0

        q = o\_r / r

        p = r

        r = o\_r - q \* p

        o\_r = p

        p = s

        s = o\_s - q \* p

        o\_s = p

        p = t

        t = o\_t - q \* p

        o\_t = p

    End While

    ret.x = o\_s

    ret.y = o\_t

    ret.GCD = o\_r

    Return ret

End Function

''' <summary>

''' Performs the modular multiplitcative inverse of a number so we can use multiplication instead of division in our arithmetic

''' </summary>

Public Shared Function Inverse(ByVal a As BigInteger, ByVal p As BigInteger) As BigInteger

    'Not explicitely tested but works

    Dim ret As RetGCD

    ret = Extended\_GCD(a, p)

    If ret.GCD <> 1 Then

        Return 0

    End If

    Return ret.x Mod p

End Function

These functions are used when calculating the multiplicative inverse of a large number in the program, its more efficient for a processor to calculate as the multiplication is streamlined microcode available on every processor.

After performing some test on the addition, the values (10,10,47), (20,20,47) were used to test that the addition was correct, the value returned from the program was not equal to the value it should have been, calculated with an EC calculator online. The difference between these two values was equal to the prime number, so the algorithm was adapted to add the value of the prime number onto the x and y values when they were less than zero:

Public Shared Function Addition(ByVal P1 As ECPoint, ByVal P2 As ECPoint) As ECPoint

    Dim P3 As New ECPoint(P1)

    Dim l, z As BigInteger

    Dim D As BigInteger = P2.x - P1.x

    If D < 0 Then

        D += P1.p

    End If

    z = Inverse(D, P1.p)

    l = ((P2.y - P1.y) \* z) Mod P1.p

    P3.x = ((l \* l) Mod P1.p - P1.x - P2.x) Mod P1.p

    P3.y = ((l \* (P1.x - P3.x)) Mod P1.p - P1.y) Mod P1.p

    If P3.x < 0 Then

        P3.x += P1.p

    End If

    If P3.y < 0 Then

        P3.y += P1.p

    End If

    Return P3

End Function

Lastly error checking was added to the program in order to make sure that the provided points were not corrupt in some way:

''' <summary>

''' Performs addition of two points

''' </summary>

Public Shared Function Addition(ByVal P1 As ECPoint, ByVal P2 As ECPoint) As ECPoint

    If P1.IsPointInfinity() Then

        Return P2

    End If

    If P2.IsPointInfinity() Then

        Return P1

    End If

    If P1.p <> P2.p Then

        Return New ECPoint(-99, -99, -99) 'Add custom errors with ENUM

    End If

    If P1.y = -P2.y Then

        Return New ECPoint(-99, -99, -99)

    End If

    Dim Y As BigInteger = -P2.y

    If Y < 0 Then

        Y += P1.p

    End If

    If P1.x = P2.x And P1.y = Y Then

        Dim ret As New ECPoint(0, 1, P1.p)

        Return ret

    End If

    Dim P3 As New ECPoint(P1)

    Dim l, z As BigInteger

    Dim D As BigInteger = P2.x - P1.x

    If D < 0 Then

        D += P1.p

    End If

    z = Inverse(D, P1.p)

    l = ((P2.y - P1.y) \* z) Mod P1.p

    P3.x = ((l \* l) Mod P1.p - P1.x - P2.x) Mod P1.p

    P3.y = ((l \* (P1.x - P3.x)) Mod P1.p - P1.y) Mod P1.p

    If P3.x < 0 Then

        P3.x += P1.p

    End If

    If P3.y < 0 Then

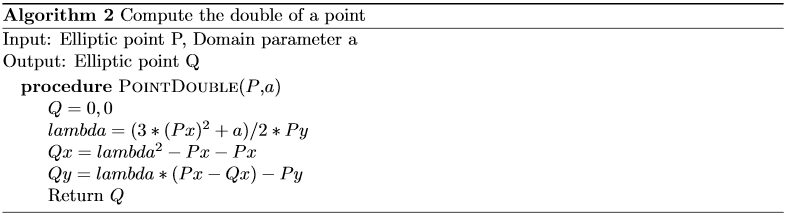
        P3.y += P1.p

    End If

    Return P3

End Function

#### Point Doubling



Like in the addition the algorithm above was transformed into the code below, now including the fix for the negative x and y values:

Public Shared Function PDouble(ByVal P As ECPoint, ByVal a As Domain\_Parameters) As ECPoint

    Dim Q As New ECPoint(P)

    Dim l, z As BigInteger

    z = Inverse(2 \* P.y, P.p)

    l = ((((3 \* P.x \* P.x) Mod P.p + a.a) Mod P.p) \* z) Mod P.p

    Q.x = ((l \* l) Mod P.p - 2 \* P.x) Mod P.p

    Q.y = ((l \* (P.x - Q.x)) Mod P.p - P.y) Mod P.p

    If Q.x < 0 Then

        Q.x += P.p

    End If

    If Q.y < 0 Then

        Q.y += P.p

    End If

    Return Q

End Function

As before we calculate the divide in lambda by using the inverse function listed above.

Finally some error checking was added to the function to try and pre-emptively fail any possible errors:

Public Shared Function PDouble(ByVal P As ECPoint, ByVal a As Domain\_Parameters) As ECPoint

    If P.p = 0 And a.Fp.p = 0 Then

        Return New ECPoint(-99, -99, -99)

    End If

    If P.IsPointInfinity() Then

        Return P

    End If

    Dim Y As BigInteger = -P.y

    If Y < 0 Then

        Y += P.p

    End If

    If P.y = Y Then

        Dim ret As New ECPoint(0, 1, P.p)

        Return ret

    End If

    Dim prime As BigInteger

    If P.p = 0 Then

        prime = a.Fp.p

    Else

        prime = P.p

    End If

    Dim Q As New ECPoint(P)

    Dim l, z As BigInteger

    z = Inverse(2 \* P.y, P.p)

    l = ((((3 \* P.x \* P.x) Mod P.p + a.a) Mod P.p) \* z) Mod P.p

    Q.x = ((l \* l) Mod P.p - 2 \* P.x) Mod P.p

    Q.y = ((l \* (P.x - Q.x)) Mod P.p - P.y) Mod P.p

    If Q.x < 0 Then

        Q.x += P.p

    End If

    If Q.y < 0 Then

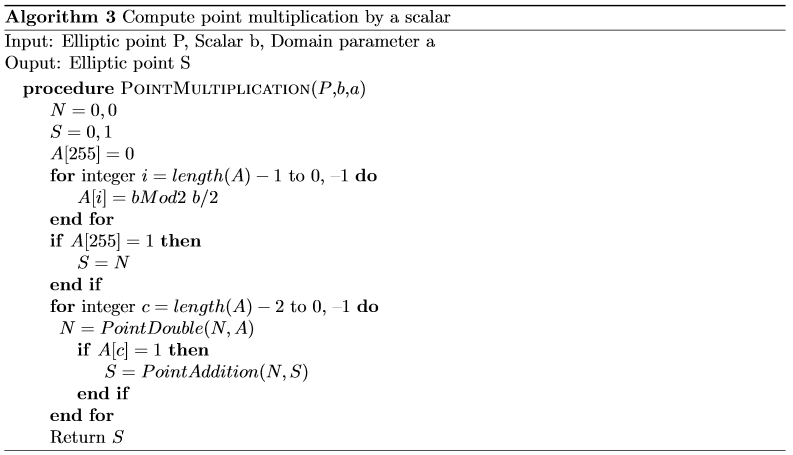
        Q.y += P.p

    End If

    Return Q

End Function

#### Point Multiplication



This was then transformed into a simple procedure:

Public Shared Function Scalar\_Mult(ByVal P As ECPoint, ByVal a As Domain\_Parameters, ByVal Scalar As BigInteger) As ECPoint

    Dim N, S As New ECPoint(P)

    S = New ECPoint(0, 1, prime)

    Dim bytearr(a.Bitlength - 1) As Byte

    Dim k As New BigInteger(Scalar.ToByteArray)

    For i = bytearr.Count - 1 To 0 Step -1

        bytearr(i) = k Mod 2

        k /= 2

    Next

    If bytearr(bytearr.Count - 1) = 1 Then

        S = N

    End If

    For i = bytearr.Count - 2 To 0 Step -1

        N = PDouble(N, a)

        If bytearr(i) = 1 Then

            S = Addition(N, S)

        End If

    Next

    Return S

End Function

Finally, error checking was added to try and catch any problems:

''' <summary>

''' Performs point multiplication with a scalar

''' </summary>

Public Shared Function Scalar\_Mult(ByVal P As ECPoint, ByVal a As Domain\_Parameters, ByVal Scalar As BigInteger) As ECPoint

    If P.p = 0 And a.Fp.p = 0 Then

        Return New ECPoint(-99, -99, -99)

    End If

    If P.IsPointInfinity Then

        Return P

    End If

    Dim prime As BigInteger

    If P.p = 0 Then

        prime = a.Fp.p

    Else

        prime = P.p

    End If

    Dim N, S As New ECPoint(P)

    S = New ECPoint(0, 1, prime)

    Dim bytearr(a.Bitlength - 1) As Byte

    Dim k As New BigInteger(Scalar.ToByteArray)

    For i = bytearr.Count - 1 To 0 Step -1

        bytearr(i) = k Mod 2

        k /= 2

    Next

    If bytearr(bytearr.Count - 1) = 1 Then

        S = N

    End If

    For i = bytearr.Count - 2 To 0 Step -1

        N = PDouble(N, a)

        If bytearr(i) = 1 Then

            S = Addition(N, S)

        End If

    Next

    Return S

End Function

### Representing a Curve

A curve is defined by its domain parameters (Fp, a, b, Base, n, h), as explained on page 12. I created a new class in visual basic to store these parameters and have accessible error checking functions available within. Firstly, when creating a new object, we have to setup the domain parameters:

Public Bitlength As Integer

Public Fp As Fp\_Operations.Fp

Public a As BigInteger

Public b As BigInteger

'Public s As BigInteger

Public Base As ECPoint

Public n As BigInteger

Public h As Integer

'Infinity point has the homogeneous coords (0:1:0), or negative y (0:-1:0)

Public Sub New(ByVal Bitlenght As Integer, ByVal p As Fp\_Operations.Fp, ByVal a As BigInteger, ByVal b As BigInteger, ByVal Base As ECPoint, ByVal n As BigInteger, ByVal h As Integer)

    Me.Bitlength = Bitlength

    Me.Fp = p

    Me.a = a

    Me.b = b

    'Me.s = s Removed the need for a seed, might be needed if we do our own curve generation

    Me.Base = Base

    Me.n = n

    Me.h = h

End Sub

As you can see the domain parameters are all accessible via public attributes and are set with the values specified for the parameters when calling **New**.

I later added a method of checking if the domain parameters were likely to be correct for a curve:

Public Shared Function Validate\_Params(ByVal Params As Domain\_Parameters) As errorEnum

    'We dont need to validate Fp as that is done upon creation

    'Dim p As Fp\_Operations.Fp = Params.Fp

    Dim bitLength As Integer = Params.Bitlength

    Dim a As BigInteger = Params.a

    Dim b As BigInteger = Params.b

    Dim Base As ECPoint = Params.Base

    Dim n As BigInteger = Params.n

    Dim h As BigInteger = Params.h

    If New BitArray(a.ToByteArray).Length <> bitLength Or New BitArray(b.ToByteArray).Length <> bitLength Then

        Return 10

    End If

    If b = 0 Then

        Return 11

    End If

    If 4 \* BigInteger.Pow(a, 3) + 27 \* BigInteger.Pow(b, 2) = 0 Then

        Return 12

    End If

    If Base.IsPointInfinity Then

        Return 3

    End If

    If h <> 1 Then

        Return 13

    End If

    Return Nothing

End Function

We start off by checking if the bitlengths of a and b are equal to the bitlength of the curve, if they are not equal then we return the enum suggesting they are not equal. We also check if b is equal to 0 as b can never be zero, if it is the check is failed. Finally, we check that a and b are correct and make sure that the base point is not infinity and that the cofactor of the size is not equal to 1.

### A Wrapper for a Predefined Curve

I created an xml file containing the variables to some predefined curves and needed a way to read them in, I then created a wrapper for the domain parameters that holds the curve’s name and the domain parameters but also has a way of processing the xml file. In the future this can be adapted to include custom curve generation and default values to rebuild the xml file later on. Currently it only looks like:

'''<summary>

'''Deffines a cruve with the general weistrass formula y^2 = x^3 + ax + b

''' </summary>

Public Class Weierstrass\_Curve

    Public Parameters As New Curve.Domain\_Parameters(0, New Fp\_Operations.Fp(1), 0, 0, New ECPoint(0, 0), 0, 0)

    Public Curve\_ID As String

    Public Sub New(ByVal Filepath As String, ByVal Curve\_Name As String)

        If Not IO.File.Exists(Filepath) Then

            Throw New IO.FileNotFoundException

        End If

        Dim found As Boolean

        Dim xmlr As New Xml.XmlDocument()

        Dim xmlnode As Xml.XmlNodeList

        Dim fs As New IO.FileStream(Filepath, IO.FileMode.Open)

        xmlr.Load(fs)

        xmlnode = xmlr.GetElementsByTagName("Curve-ID")

        For i = 0 To xmlnode.Count - 1

            If xmlnode(i).Attributes(0).InnerText = Curve\_Name Then

                If xmlnode(i).ChildNodes.Count <> 8 Then

                    fs.Close()

                    Throw New Exception("The current file is corrupted or the curve entry is corrupted, rebuild the curves.xml file from the menu")

                End If

                Me.Curve\_ID = Curve\_Name

                Me.Parameters.Bitlength = Convert.ToInt32(xmlnode(i).ChildNodes(0).InnerText)

                Me.Parameters.Fp = New Fp\_Operations.Fp(BigInteger.Parse(xmlnode(i).ChildNodes(1).InnerText))

                Me.Parameters.a = BigInteger.Parse(xmlnode(i).ChildNodes(2).InnerText)

                Me.Parameters.b = BigInteger.Parse(xmlnode(i).ChildNodes(3).InnerText)

                Me.Parameters.Base.x = BigInteger.Parse(xmlnode(i).ChildNodes(4).InnerText)

                Me.Parameters.Base.y = BigInteger.Parse(xmlnode(i).ChildNodes(5).InnerText)

                Me.Parameters.Base.p = Me.Parameters.Fp.p

                Me.Parameters.n = BigInteger.Parse(xmlnode(i).ChildNodes(6).InnerText)

                Me.Parameters.h = BigInteger.Parse(xmlnode(i).ChildNodes(7).InnerText)

                found = True

            End If

        Next

        If found = False Then

            Throw New Exception("Curve could not be found in the xml file")

        End If

        fs.Close()

    End Sub

End Class

Simply this opens up the file with the xml reader that is built into the xml library in visual basic. Then we loop through all of the nodes in the document, finding the one that is equal to the specified name of the curve, once a match is found the values that are listed inside this container are then converted to their respective domain parameters and they are then validated for problems.

The current curve.xml file looks like so:

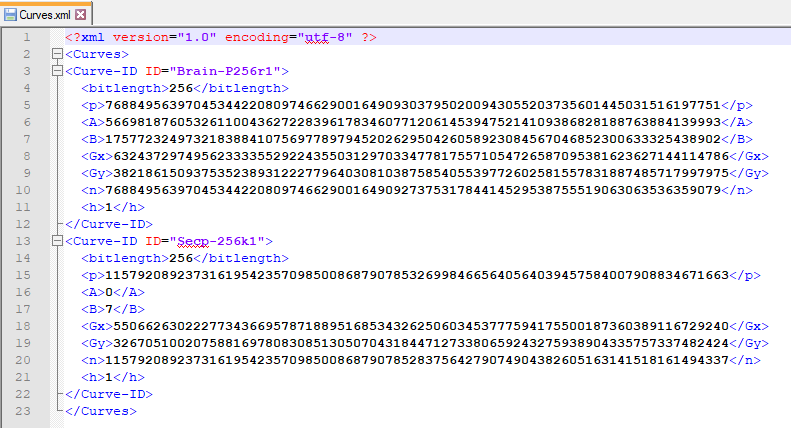


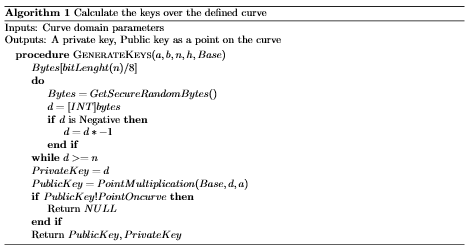
Figure 16: Curves.xml file

As you can see there are currently two entries so far, the 256-bit Brainpool curve and the 256-bit Secp curve. There exists also the NIST curves, but I refrained from add them due to rumours that they are defined with a secret, making it possible for a private key to be easily generated.

### Elliptic Curve Diffie-Hellman

To recount from the research section, Elliptic Curve Diffie-Hellman, also known as ECDH, is a method for two parties to share keys with each other over an insecure channel. This requires us to generate two keys per party and then using those keys generate an elliptic curve point that is used during the encryption stage. In order to generate these two keys a curve must be pre-defined, as the parameters of the curve are later needed in some of the point calculations defined above.

As a basis for my design I developed an algorithm to help me with the design of the key generation:



As of the algorithm above I transformed the pseudocode into visual basic code:

Public Structure Keys

Public PublicKey As ECPoint

Public PrivateKey As BigInteger

End Structure

Public Shared Function generate\_Keys(ByVal Param As Domain\_Parameters) As Keys

Dim \_\_PrivateKey As BigInteger

Dim \_\_PublicKey As ECPoint

Dim d As BigInteger

Dim rng As New RNGCryptoServiceProvider

Dim bytes(Param.n.ToByteArray.Length) As Byte

Do

rng.GetBytes(bytes)

d = New BigInteger(bytes)

If d.Sign = -1 Then

d = d \* -1

End If

Loop While d >= Param.n

\_\_PrivateKey = d

\_\_PublicKey = Point\_Operations.Scalar\_Mult(Param.Base, Param, d)

If \_\_PublicKey.IsPointOnCurve(Param) = False Then

Throw New Exception("Key generation failed")

End If

Dim Ret As New Keys

Ret.PublicKey = \_\_PublicKey

Ret.PrivateKey = \_\_PrivateKey

\_\_PrivateKey = BigInteger.Zero()

d = BigInteger.Zero()

bytes = {0}

Return Ret

End Function

Where in the algorithm **NULL** is returned when its determined that the point is not valid and therefore does not lie on the curve defined by the parameters, we throw an error instead. This error can then be caught by the calling subroutine, and the message can be handled correctly. I added a quick check at the start to determine if the parameters are likely to be valid by calling **Validate\_Params**:

Public Shared Function generate\_Keys(ByVal Param As Domain\_Parameters) As Keys

If Domain\_Parameters.Validate\_Params(Param) <> Nothing Then

Throw New Exception("Provided parameters not valid as a curve")

End If

Dim \_\_PrivateKey As BigInteger

Dim \_\_PublicKey As ECPoint

Dim d As BigInteger

Dim rng As New RNGCryptoServiceProvider

Dim bytes(Param.n.ToByteArray.Length) As Byte

Do

rng.GetBytes(bytes)

d = New BigInteger(bytes)

If d.Sign = -1 Then

d = d \* -1

End If

Loop While d >= Param.n

\_\_PrivateKey = d

\_\_PublicKey = Point\_Operations.Scalar\_Mult(Param.Base, Param, d)

If \_\_PublicKey.IsPointOnCurve(Param) = False Then

Throw New Exception("Key generation failed")

End If

Dim Ret As New Keys

Ret.PublicKey = \_\_PublicKey

Ret.PrivateKey = \_\_PrivateKey

\_\_PrivateKey = BigInteger.Zero()

d = BigInteger.Zero()

bytes = {0}

Return Ret

End Function

Lastly, for the generating keys instead of throwing exceptions, I used the new error handler that I wrote in order to notify the user of the problems:

Public Shared Function generate\_Keys(ByVal Param As Domain\_Parameters) As Keys

Dim validateError As errorEnum = Domain\_Parameters.Validate\_Params(Param)

If validateError <> Nothing Then

errorHappened(Param, "Parameters are not a valid curve", validateError, GetCurrentMethod.Name)

Return Nothing

End If

Dim \_\_PrivateKey As BigInteger

Dim \_\_PublicKey As ECPoint

Dim d As BigInteger

Dim rng As New RNGCryptoServiceProvider

Dim bytes(Param.n.ToByteArray.Length) As Byte

Do

rng.GetBytes(bytes)

d = New BigInteger(bytes)

If d.Sign = -1 Then

d = d \* -1

End If

Loop While d >= Param.n

\_\_PrivateKey = d

\_\_PublicKey = Point\_Operations.Scalar\_Mult(Param.Base, Param, d)

If \_\_PublicKey.IsPointOnCurve(Param) = False Then

errorHappened(\_\_PublicKey, "The generated public key was not valid for the provided parameters", errorEnum.POINT\_NOT\_ON\_CURVE, GetCurrentMethod.Name)

Return Nothing

End If

Dim Ret As New Keys

Ret.PublicKey = \_\_PublicKey

Ret.PrivateKey = \_\_PrivateKey

\_\_PrivateKey = BigInteger.Zero()

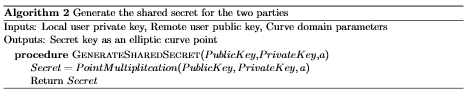
d = BigInteger.Zero()

bytes = {0}

Return Ret

End Function

Next part was to transform the algorithm for generating the shared secret into pseudocode to give me a basis for the design of the function used in the program:



I added some validation to the subroutine to strengthen it, making it more resilient to abuse:

Public Shared Function generate\_SharedSecret(ByVal PrivateKey As BigInteger, ByVal PublicKey As ECPoint, ByVal Curve\_Parameters As Domain\_Parameters) As ECPoint

Dim Secret As ECPoint

If PrivateKey = 0 Or PrivateKey = Nothing Then

errorHappened(PrivateKey, "Private key was null or 0", errorEnum.NULL\_REFERECE\_EXCEPTION, GetCurrentMethod.Name)

Return New ECPoint(-99, -99, -99)

End If

If PublicKey.IsPointInfinity Then

errorHappened(PublicKey, "Public key point was infinity", errorEnum.POINT\_INFINITY, GetCurrentMethod.Name)

Return New ECPoint(-99, -99, -99)

End If

Dim validateParamsError As errorEnum = Domain\_Parameters.Validate\_Params(Curve\_Parameters)

If validateParamsError <> Nothing Then

errorHappened(Curve\_Parameters, "Provided parameters not valid as a curve", validateParamsError, GetCurrentMethod.Name)

Return New ECPoint(-99, -99, -99)

End If

If PublicKey.IsPointOnCurve(Curve\_Parameters) = False Then

errorHappened(PublicKey, "Point was not found on provided curve", errorEnum.POINT\_NOT\_ON\_CURVE, GetCurrentMethod.Name)

Return New ECPoint(-99, -99, -99)

End If

Secret = Point\_Operations.Scalar\_Mult(PublicKey, Curve\_Parameters, PrivateKey)

If Secret.IsPointInfinity Then

errorHappened(Secret, "Generated secret key was at point of infinity", errorEnum.POINT\_INFINITY, GetCurrentMethod.Name)

Return New ECPoint(-99, -99, -99)

End If

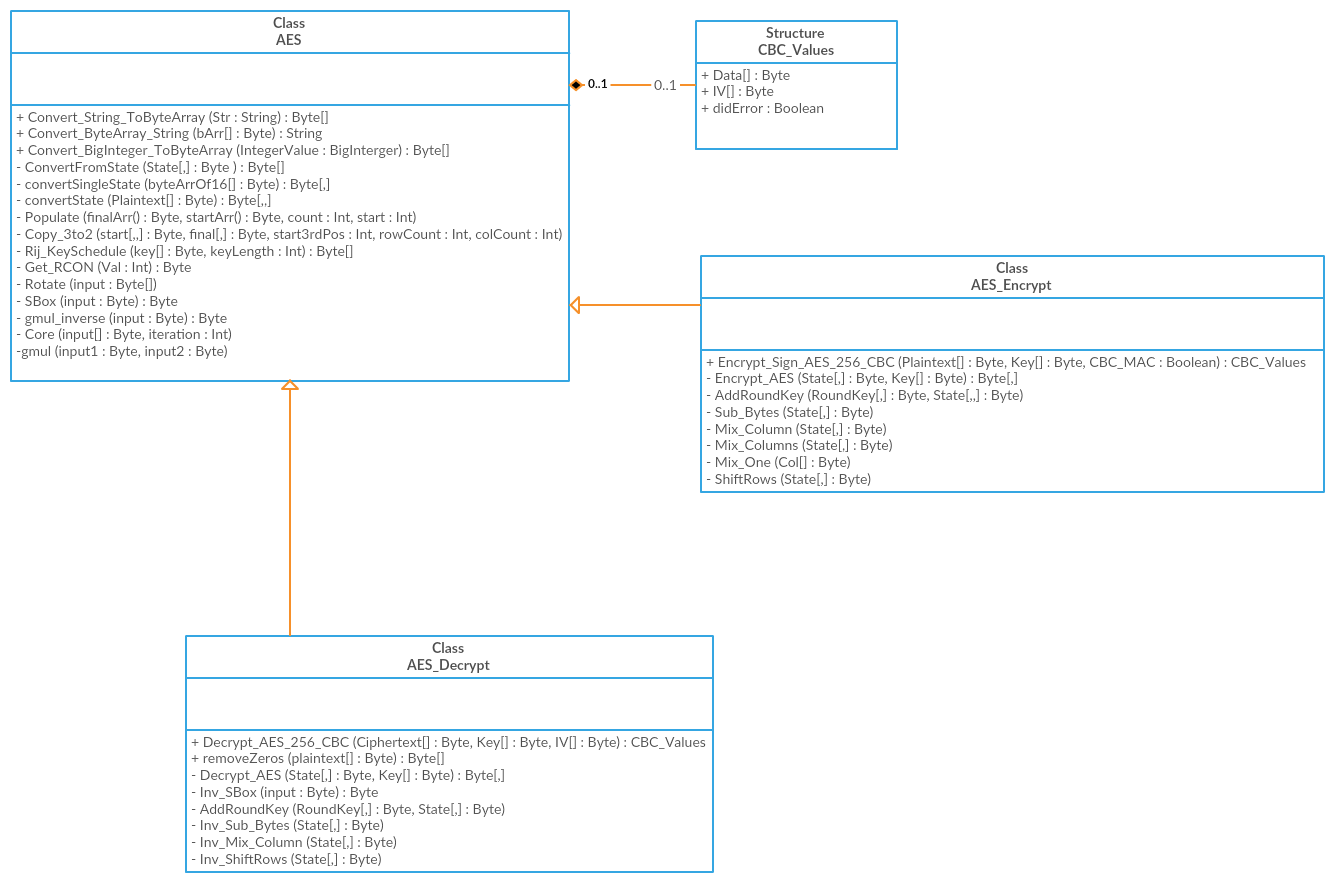
Return Secret

End Function

As above the pseudocode is exactly the same as the visual basic code below it, but it contains routines to check that the arguments provided are correct and that the generated point is not the point at infinity.

### Writing the AES algorithm

Firstly, I started off by writing a base class that is set MustInherit, which provides the AES\_Encrypt and AES\_Decrypt functions with some basic functions, or the Rijndael’s key schedule which is required by both of the encrypt and decrypt functions. The class diagram for this section is shown below:

****

When doing my test with the encryption I needed to use my own big integer, byte array, and string conversion functions to ensure that the endianness of the values was kept consistent. I also had to add routines for converting the data into a State3 and a State2 which are two and three dimensional states. Although a State3 was not explicitly required and could have been done with an array of State2s I deemed it better to use a State3 to make them easily distinguishable. Below is the code behind the base class for the Encryption and decryption routines:

Public MustInherit Class AES

''' <summary>

''' A structure that contains the two values returned from CBC encryption

''' </summary>

Public Structure CBC\_Values

Public Data() As Byte 'The plaintext or chiphertext

Public IV() As Byte 'The Initialisation Vector that was used or generated

Public didError As Boolean

End Structure

''' <summary>

''' Function that converts a string to a byte array

''' </summary>

Public Shared Function Convert\_String\_ToByteArray(ByVal Str As String) As Byte()

Dim chars() As Char = Str.ToCharArray

Dim returnByte(chars.Length - 1) As Byte

'Converts the character to byte in the index location determined by i and store the byte in the byte array at the location i

For i = 0 To chars.Length - 1

returnByte(i) = Convert.ToByte(chars(i))

Next

Return returnByte

End Function

''' <summary>

''' Converts a byte array to string

''' </summary>

Public Shared Function Convert\_ByteArray\_ToString(ByVal bArr() As Byte) As String

Dim str As String

For i = 0 To bArr.Length - 1

str += Convert.ToChar(bArr(i))

Next

Return str

End Function

''' <summary>

''' This function converts big integers to byte arrays rather than using .tobytearray due to the fact that .tobytearray could possibly use the wrong endianess

''' </summary>

Public Shared Function Convert\_BigInteger\_ToByteArray(ByVal IntegerValue As BigInteger) As Byte()

Dim values() As Byte = IntegerValue.ToByteArray

If BitConverter.IsLittleEndian Then 'checks if the computer processor works in little endian as this program must be run in big endian

Array.Reverse(values) 'reverses the byte array

End If

Return values

End Function

''' <summary>

''' Converts the 2 dimensional state array back into a 1 dimensional array

''' </summary>

Friend Function ConvertFromState(ByVal State(,) As Byte) As Byte()

Dim Ret(State.Length - 1) As Byte

Dim i As Integer = Ret.Count - 1

For c = 0 To 3 'column counter

For r = 0 To 3 'row counter

Ret(r + 4 \* c) = State(r, c)

Next

Next

Return Ret

End Function

''' <summary>

''' Converts a byte array of 16 values to a 2 dimensional state rather than a free

''' </summary>

Friend Function convertSingleState(ByVal byteArrOf16 As Byte()) As Byte(,)

If byteArrOf16.Count > 16 Then

errorHappened(byteArrOf16, "Array larger than allowed", 6, GetCurrentMethod.Name)

Return Nothing

End If

Dim state(3, 3) As Byte

For c = 0 To 3

For r = 0 To 3

state(r, c) = byteArrOf16(r + c \* 4)

Next

Next

Return state

End Function

''' <summary>

''' Debugging function

''' </summary>

Friend Sub check\_state\_Val(ByVal State(,) As Byte)

For r = 0 To 3

Dim col(4) As Byte

For c = 0 To 3

col(c) = State(r, c)

Next

Debug.Print(Hex(col(0)) & " " & Hex(col(1)) & " " & Hex(col(2)) & " " & Hex(col(3))) 'prints the hex values in the form FF FF FF FF

Next

Debug.Print(" ")

End Sub

''' <summary>

''' This converts a byte array that is greater than 16 bytes to a state of 3 dimensions

''' </summary>

Friend Function convertState(ByVal Plaintext As Byte()) As Byte(,,)

Dim Nb, Nk As Integer

Nk = 4 'row count

Nb = 4 'column count

Dim SCount As Integer

Dim tmp((Nb \* Nk) - 1) As Byte

If Plaintext.Count = 0 Then 'checks for empty array

errorHappened(Plaintext, "Plaintext is Null", 1, GetCurrentMethod.Name)

End If

'Calculates the number of States needed each state is a 4x4 array

If Plaintext.Length \* 8 > (4 \* Nb \* Nk) Then

SCount = Math.Ceiling(Plaintext.Length / (Nb \* Nk))

End If

'If we need 0 States we need to increase than number to 1 otherwise the algorithm next fails

If SCount = 0 Then

SCount += 1

End If

'We define the new 3 dimensional State array

Dim State(SCount - 1, Nb - 1, Nk - 1) As Byte

'And fill the state array

For i = 0 To SCount - 1

Populate(tmp, Plaintext, (Nb \* Nk), (Nb \* Nk) \* i)

For c = 0 To (Nb \* Nk) - 1

State(i, c Mod Nk, Math.Floor(c / Nk)) = tmp(c)

Next

Array.Clear(tmp, 0, tmp.Count - 1)

Next

Return State

End Function

''' <summary>

''' This function is used to copy as set of specific values from startarray into the finalarray

''' </summary>

Friend Sub Populate(ByRef finalarr() As Byte, ByVal startarr() As Byte, ByVal count As Integer, ByVal start As Integer)

Dim max As Integer

If start + count < startarr.Length Then

max = start + count

Else

max = startarr.Length

End If

For i = start To max - 1

finalarr(i - start) = startarr(i)

Next

End Sub

''' <summary>

''' Copies a selected state from the 3 dimensional State array into the 2 dimensional one for use with AES algorithms

''' </summary>

Friend Sub Copy\_3to2(ByVal start(,,) As Byte, ByRef final(,) As Byte, ByVal start3rdPos As Integer, ByVal rowCount As Integer, ByVal colCount As Integer)

For r = 0 To rowCount - 1

For c = 0 To colCount - 1

final(r, c) = start(start3rdPos, r, c)

Next

Next

End Sub

''' <summary>

''' Function used to extend the provided key for AES into something longer

''' </summary>

Friend Function Rij\_KeySchedule(ByVal Key() As Byte, ByVal keyLength As Integer) As Byte()

Dim n As Integer

Dim b As Integer

'This is depreciated as we are only allowing 256bit curves

If keyLength = 128 Then

n = 16

b = 176

ElseIf keyLength = 192 Then

n = 24

b = 208

ElseIf keyLength = 256 Then

n = 32

b = 240

Else

Throw New Exception("Key Lenght not valid")

End If

'ExpKey is the extended key

Dim ExpKey(b - 1) As Byte

Array.Copy(Key, 0, ExpKey, 0, n)

Dim t(3) As Byte

Dim c, i, a As Integer

c = n

i = 1

'This function extends the key as seen on https://en.wikipedia.org/wiki/Rijndael\_key\_schedule

While c < b

For a = 0 To 3

t(a) = ExpKey(a + c - 4)

Next

If c Mod n = 0 Then

Core(t, i)

i += 1

End If

If c Mod n = 16 Then

For a = 0 To 3

t(a) = SBox(t(a))

Next

End If

For a = 0 To 3

ExpKey(c) = ExpKey(c - n) Xor t(a)

c += 1

Next

End While

Return ExpKey

End Function

''' <summary>

''' This perfroms the exponentation of 2 to a user specified value

''' </summary>

Friend Function Get\_RCON(ByVal Val As Integer) As Byte

Dim c As Byte = 1

If IsNothing(Val) Then

errorHappened(Val, "RCON parameter is Null", 1, GetCurrentMethod.Name)

Return Nothing

End If

While Val <> 1

Dim b As Byte

b = c And 128

c <<= 1

If b = 128 Then

c = c Xor 27

End If

'gmul(one, 2)

Val -= 1

End While

Return c

End Function

''' <summary>

''' This shifts a 4 byte array left by 1 and wraps the end value round to the start like so:

''' Before: 32 AA FF 10

''' After: AA FF 10 32

''' </summary>

Friend Sub Rotate(ByRef input As Byte())

Dim a As Byte

a = input(0)

For b = 0 To 2

input(b) = input(b + 1)

Next

input(3) = a

End Sub

''' <summary>

''' This is a lookup table of multiplicative inverses for a number inside the Rijndaels finite field

''' </summary>

Friend Function SBox(ByVal input As Byte) As Byte

Dim sboxvals(255) As Byte

sboxvals = {99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 118, 202, 130, 201, 125, 250, 89, 71, 240, 173, 212, 162, 175, 156, 164, 114, 192, 183, 253, 147, 38, 54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21, 4, 199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39, 178, 117, 9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214, 179, 41, 227, 47, 132, 83, 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88, 207, 208, 239, 170, 251, 67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168, 81, 163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33, 16, 255, 243, 210, 205, 12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115, 96, 129, 79, 220, 34, 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219, 224, 50, 58, 10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121, 231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8, 186, 120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139, 138, 112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158, 225, 248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206, 85, 40, 223, 140, 161, 137, 13, 191, 230, 66, 104, 65, 153, 45, 15, 176, 84, 187, 22}

Return sboxvals(input)

End Function

''' <summary>

''' This performs the inverse of Galios Field multiplication

''' </summary>

Friend Function gmul\_inverse(ByVal input As Byte) As Byte

If IsNothing(input) Then

'Throw New Exception("gmul\_inverse param is zero")

errorHappened(input, "Parameter is Null", 1, GetCurrentMethod.Name)

Return Nothing

End If

Dim altable(255), ltable(255) As Byte

Dim ret As Byte

ltable = {0, 255, 200, 8, 145, 16, 208, 54, 90, 62, 216, 67, 153, 119, 254, 24, 35, 32, 7, 112, 161, 108, 12, 127, 98, 139, 64, 70, 199, 75, 224, 14, 235, 22, 232, 173, 207, 205, 57, 83, 106, 39, 53, 147, 212, 78, 72, 195, 43, 121, 84, 40, 9, 120, 15, 33, 144, 135, 20, 42, 169, 156, 214, 116, 180, 124, 222, 237, 177, 134, 118, 164, 152, 226, 150, 143, 2, 50, 28, 193, 51, 238, 239, 129, 253, 48, 92, 19, 157, 41, 23, 196, 17, 68, 140, 128, 243, 115, 66, 30, 29, 181, 240, 18, 209, 91, 65, 162, 215, 44, 233, 213, 89, 203, 80, 168, 220, 252, 242, 86, 114, 166, 101, 47, 159, 155, 61, 186, 125, 194, 69, 130, 167, 87, 182, 163, 122, 117, 79, 174, 63, 55, 109, 71, 97, 190, 171, 211, 95, 176, 88, 175, 202, 94, 250, 133, 228, 77, 138, 5, 251, 96, 183, 123, 184, 38, 74, 103, 198, 26, 248, 105, 37, 179, 219, 189, 102, 221, 241, 210, 223, 3, 141, 52, 217, 146, 13, 99, 85, 170, 73, 236, 188, 149, 60, 132, 11, 245, 230, 231, 229, 172, 126, 110, 185, 249, 218, 142, 154, 201, 36, 225, 10, 21, 107, 58, 160, 81, 244, 234, 178, 151, 158, 93, 34, 136, 148, 206, 25, 1, 113, 76, 165, 227, 197, 49, 187, 204, 31, 45, 59, 82, 111, 246, 46, 137, 247, 192, 104, 27, 100, 4, 6, 191, 131, 56}

altable = {1, 229, 76, 181, 251, 159, 252, 18, 3, 52, 212, 196, 22, 186, 31, 54, 5, 92, 103, 87, 58, 213, 33, 90, 15, 228, 169, 249, 78, 100, 99, 238, 17, 55, 224, 16, 210, 172, 165, 41, 51, 89, 59, 48, 109, 239, 244, 123, 85, 235, 77, 80, 183, 42, 7, 141, 255, 38, 215, 240, 194, 126, 9, 140, 26, 106, 98, 11, 93, 130, 27, 143, 46, 190, 166, 29, 231, 157, 45, 138, 114, 217, 241, 39, 50, 188, 119, 133, 150, 112, 8, 105, 86, 223, 153, 148, 161, 144, 24, 187, 250, 122, 176, 167, 248, 171, 40, 214, 21, 142, 203, 242, 19, 230, 120, 97, 63, 137, 70, 13, 53, 49, 136, 163, 65, 128, 202, 23, 95, 83, 131, 254, 195, 155, 69, 57, 225, 245, 158, 25, 94, 182, 207, 75, 56, 4, 185, 43, 226, 193, 74, 221, 72, 12, 208, 125, 61, 88, 222, 124, 216, 20, 107, 135, 71, 232, 121, 132, 115, 60, 189, 146, 201, 35, 139, 151, 149, 68, 220, 173, 64, 101, 134, 162, 164, 204, 127, 236, 192, 175, 145, 253, 247, 79, 129, 47, 91, 234, 168, 28, 2, 209, 152, 113, 237, 37, 227, 36, 6, 104, 179, 147, 44, 111, 62, 108, 10, 184, 206, 174, 116, 177, 66, 180, 30, 211, 73, 233, 156, 200, 198, 199, 34, 110, 219, 32, 191, 67, 81, 82, 102, 178, 118, 96, 218, 197, 243, 246, 170, 205, 154, 160, 117, 84, 14, 1}

ret = Convert.ToByte(altable((256 - ltable(input))))

Return ret

End Function

''' <summary>

''' This is the inner loop of the key schedule

''' </summary>

Friend Sub Core(ByRef input As Byte(), ByVal iteration As Integer)

Rotate(input)

For i = 0 To 3

input(i) = SBox(input(i))

Next

input(0) = input(0) Xor Get\_RCON(iteration)

End Sub

''' <summary>

''' perfoms galios field multiplication using lookup tables

''' </summary>

Friend Function gmul(ByVal input1 As Byte, ByVal input2 As Byte)

If input1 = 0 Or input2 = 0 Then

Return 0

End If

Dim altable(255), ltable(255) As Byte

'log table

ltable = {0, 255, 200, 8, 145, 16, 208, 54, 90, 62, 216, 67, 153, 119, 254, 24, 35, 32, 7, 112, 161, 108, 12, 127, 98, 139, 64, 70, 199, 75, 224, 14, 235, 22, 232, 173, 207, 205, 57, 83, 106, 39, 53, 147, 212, 78, 72, 195, 43, 121, 84, 40, 9, 120, 15, 33, 144, 135, 20, 42, 169, 156, 214, 116, 180, 124, 222, 237, 177, 134, 118, 164, 152, 226, 150, 143, 2, 50, 28, 193, 51, 238, 239, 129, 253, 48, 92, 19, 157, 41, 23, 196, 17, 68, 140, 128, 243, 115, 66, 30, 29, 181, 240, 18, 209, 91, 65, 162, 215, 44, 233, 213, 89, 203, 80, 168, 220, 252, 242, 86, 114, 166, 101, 47, 159, 155, 61, 186, 125, 194, 69, 130, 167, 87, 182, 163, 122, 117, 79, 174, 63, 55, 109, 71, 97, 190, 171, 211, 95, 176, 88, 175, 202, 94, 250, 133, 228, 77, 138, 5, 251, 96, 183, 123, 184, 38, 74, 103, 198, 26, 248, 105, 37, 179, 219, 189, 102, 221, 241, 210, 223, 3, 141, 52, 217, 146, 13, 99, 85, 170, 73, 236, 188, 149, 60, 132, 11, 245, 230, 231, 229, 172, 126, 110, 185, 249, 218, 142, 154, 201, 36, 225, 10, 21, 107, 58, 160, 81, 244, 234, 178, 151, 158, 93, 34, 136, 148, 206, 25, 1, 113, 76, 165, 227, 197, 49, 187, 204, 31, 45, 59, 82, 111, 246, 46, 137, 247, 192, 104, 27, 100, 4, 6, 191, 131, 56}

'antilog table

altable = {1, 229, 76, 181, 251, 159, 252, 18, 3, 52, 212, 196, 22, 186, 31, 54, 5, 92, 103, 87, 58, 213, 33, 90, 15, 228, 169, 249, 78, 100, 99, 238, 17, 55, 224, 16, 210, 172, 165, 41, 51, 89, 59, 48, 109, 239, 244, 123, 85, 235, 77, 80, 183, 42, 7, 141, 255, 38, 215, 240, 194, 126, 9, 140, 26, 106, 98, 11, 93, 130, 27, 143, 46, 190, 166, 29, 231, 157, 45, 138, 114, 217, 241, 39, 50, 188, 119, 133, 150, 112, 8, 105, 86, 223, 153, 148, 161, 144, 24, 187, 250, 122, 176, 167, 248, 171, 40, 214, 21, 142, 203, 242, 19, 230, 120, 97, 63, 137, 70, 13, 53, 49, 136, 163, 65, 128, 202, 23, 95, 83, 131, 254, 195, 155, 69, 57, 225, 245, 158, 25, 94, 182, 207, 75, 56, 4, 185, 43, 226, 193, 74, 221, 72, 12, 208, 125, 61, 88, 222, 124, 216, 20, 107, 135, 71, 232, 121, 132, 115, 60, 189, 146, 201, 35, 139, 151, 149, 68, 220, 173, 64, 101, 134, 162, 164, 204, 127, 236, 192, 175, 145, 253, 247, 79, 129, 47, 91, 234, 168, 28, 2, 209, 152, 113, 237, 37, 227, 36, 6, 104, 179, 147, 44, 111, 62, 108, 10, 184, 206, 174, 116, 177, 66, 180, 30, 211, 73, 233, 156, 200, 198, 199, 34, 110, 219, 32, 191, 67, 81, 82, 102, 178, 118, 96, 218, 197, 243, 246, 170, 205, 154, 160, 117, 84, 14, 1}

Dim a As Integer

a = (Convert.ToInt32(ltable(input1)) + Convert.ToInt32(ltable(input2))) Mod 255

a = altable(a)

Return a

End Function

End Class

The functions from the start up to and including Copy\_3to2 are simple functions that convert the data provided in the arguments to the procedure to the form indicated in the summary of the functions. The last functions in the class are used for key lengthening, as explained on pages 18-20, and some changes were made to the code behind them. The first change was made to fix a problem with generating the log and anti-log tables, and instead, replacing the code to create them with a pre-defined array for the tables:

''' <summary>

''' This performs the inverse of Galios Field multiplication

''' </summary>

Friend Function gmul\_inverse(ByVal input As Byte) As Byte

If IsNothing(input) Then

'Throw New Exception("gmul\_inverse param is zero")

errorHappened(input, "Parameter is Null", 1, GetCurrentMethod.Name)

Return Nothing

End If

'Dim one, c As Byte

Dim altable(255), ltable(255) As Byte

Dim ret As Byte

one = 1

For i = 0 To 255

altable(i) = one

c = one And 128

one <<= 1

If c = 128 Then

one = one Xor 27+ Curkey {Length=16} Byte(,)

End If

one = one Xor altable(i)

ltable(altable(i)) = i

Next

altable(255) = altable(0)

ltable(0) = 0

ret = Convert.ToByte(altable((256 - ltable(input))))

Return ret

End Function

As seen above highlighted in a darker colour shows the section of code that is used to generate the log and antilog tables for use when calculating the inverse of a value of the Galios Filed. This way seemed to produce and error in the values so instead I got a table of value from the Wikipedia site and used an online string manipulation tool to automatically format it correctly for use with visual basic, this replaced the section of code above with:

''' <summary>

''' This performs the inverse of Galios Field multiplication

''' </summary>

Friend Function gmul\_inverse(ByVal input As Byte) As Byte

If IsNothing(input) Then

errorHappened(input, "Parameter is Null", 1, GetCurrentMethod.Name)

Return Nothing

End If

Dim altable(255), ltable(255) As Byte

Dim ret As Byte

ltable = {0, 255, 200, 8, 145, 16, 208, 54, 90, 62, 216, 67, 153, 119, 254, 24, 35, 32, 7, 112, 161, 108, 12, 127, 98, 139, 64, 70, 199, 75, 224, 14, 235, 22, 232, 173, 207, 205, 57, 83, 106, 39, 53, 147, 212, 78, 72, 195, 43, 121, 84, 40, 9, 120, 15, 33, 144, 135, 20, 42, 169, 156, 214, 116, 180, 124, 222, 237, 177, 134, 118, 164, 152, 226, 150, 143, 2, 50, 28, 193, 51, 238, 239, 129, 253, 48, 92, 19, 157, 41, 23, 196, 17, 68, 140, 128, 243, 115, 66, 30, 29, 181, 240, 18, 209, 91, 65, 162, 215, 44, 233, 213, 89, 203, 80, 168, 220, 252, 242, 86, 114, 166, 101, 47, 159, 155, 61, 186, 125, 194, 69, 130, 167, 87, 182, 163, 122, 117, 79, 174, 63, 55, 109, 71, 97, 190, 171, 211, 95, 176, 88, 175, 202, 94, 250, 133, 228, 77, 138, 5, 251, 96, 183, 123, 184, 38, 74, 103, 198, 26, 248, 105, 37, 179, 219, 189, 102, 221, 241, 210, 223, 3, 141, 52, 217, 146, 13, 99, 85, 170, 73, 236, 188, 149, 60, 132, 11, 245, 230, 231, 229, 172, 126, 110, 185, 249, 218, 142, 154, 201, 36, 225, 10, 21, 107, 58, 160, 81, 244, 234, 178, 151, 158, 93, 34, 136, 148, 206, 25, 1, 113, 76, 165, 227, 197, 49, 187, 204, 31, 45, 59, 82, 111, 246, 46, 137, 247, 192, 104, 27, 100, 4, 6, 191, 131, 56}

altable = {1, 229, 76, 181, 251, 159, 252, 18, 3, 52, 212, 196, 22, 186, 31, 54, 5, 92, 103, 87, 58, 213, 33, 90, 15, 228, 169, 249, 78, 100, 99, 238, 17, 55, 224, 16, 210, 172, 165, 41, 51, 89, 59, 48, 109, 239, 244, 123, 85, 235, 77, 80, 183, 42, 7, 141, 255, 38, 215, 240, 194, 126, 9, 140, 26, 106, 98, 11, 93, 130, 27, 143, 46, 190, 166, 29, 231, 157, 45, 138, 114, 217, 241, 39, 50, 188, 119, 133, 150, 112, 8, 105, 86, 223, 153, 148, 161, 144, 24, 187, 250, 122, 176, 167, 248, 171, 40, 214, 21, 142, 203, 242, 19, 230, 120, 97, 63, 137, 70, 13, 53, 49, 136, 163, 65, 128, 202, 23, 95, 83, 131, 254, 195, 155, 69, 57, 225, 245, 158, 25, 94, 182, 207, 75, 56, 4, 185, 43, 226, 193, 74, 221, 72, 12, 208, 125, 61, 88, 222, 124, 216, 20, 107, 135, 71, 232, 121, 132, 115, 60, 189, 146, 201, 35, 139, 151, 149, 68, 220, 173, 64, 101, 134, 162, 164, 204, 127, 236, 192, 175, 145, 253, 247, 79, 129, 47, 91, 234, 168, 28, 2, 209, 152, 113, 237, 37, 227, 36, 6, 104, 179, 147, 44, 111, 62, 108, 10, 184, 206, 174, 116, 177, 66, 180, 30, 211, 73, 233, 156, 200, 198, 199, 34, 110, 219, 32, 191, 67, 81, 82, 102, 178, 118, 96, 218, 197, 243, 246, 170, 205, 154, 160, 117, 84, 14, 1}

ret = Convert.ToByte(altable((256 - ltable(input))))

Return ret

End Function

As you can now see the byte array for the tables are now defined using the {} brackets. This fixed any errors during my testing with calculating the correct inverse for a number.

The next adaption was to fix the SBox routine, rather than trying to calculate the multiplicative inverse of the given number, like above we set a defined array of SBox values that are then used to lookup the input, giving us the interchangeable byte that we need:

''' <summary>

''' This is a lookup table of multiplicative inverses for a number inside the Rijndaels finite field

''' </summary>

Friend Function SBox(ByVal input As Byte) As Byte

Dim b, c As Byte

c = gmul\_inverse(input)

b = c

For i = 0 To 3

b = (b << 1) Or (b >> 7)

c = c Xor b

Next

c = c Xor 99

Return c

End Function

''' <summary>

''' This is a lookup table of multiplicative inverses for a number inside the Rijndaels finite field

''' </summary>

Friend Function SBox(ByVal input As Byte) As Byte

Dim sboxvals(255) As Byte

sboxvals = {99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 118, 202, 130, 201, 125, 250, 89, 71, 240, 173, 212, 162, 175, 156, 164, 114, 192, 183, 253, 147, 38, 54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21, 4, 199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39, 178, 117, 9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214, 179, 41, 227, 47, 132, 83, 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88, 207, 208, 239, 170, 251, 67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168, 81, 163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33, 16, 255, 243, 210, 205, 12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115, 96, 129, 79, 220, 34, 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219, 224, 50, 58, 10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121, 231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8, 186, 120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139, 138, 112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158, 225, 248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206, 85, 40, 223, 140, 161, 137, 13, 191, 230, 66, 104, 65, 153, 45, 15, 176, 84, 187, 22}

Return sboxvals(input)

End Function

As you can see the change above replaces the code with the byte array that needs to be used for the lookup of any given value.

#### AES Encryption

As written in the research above, AES follows a simple procedure for encrypting a state of data, consisting of a number of rounds which slowly transform the data in a reversible process to what was applied. Below is some of the code behind the state transformations:

Private Sub AddRoundKey(ByVal RoundKey(,) As Byte, ByRef State(,) As Byte)

    For c = 0 To 3

        For r = 0 To 3

            State(r, c) = RoundKey(r, c) Xor State(r, c)

        Next

    Next

End Sub

Private Sub Sub\_Bytes(ByRef State(,) As Byte)

    For c = 0 To 3

        For r = 0 To 3

            State(r, c) = SBox(State(r, c))

        Next

    Next

End Sub

Private Sub Mix\_Column(ByRef State(,) As Byte)

    Dim tmpState(3, 3) As Byte

    Dim DoubleState(3, 3) As Byte

    For c = 0 To 3

        For r = 0 To 3

            Dim highBit As Byte

            tmpState(r, c) = State(r, c)

            highBit = State(r, c) And 128

            DoubleState(r, c) = State(r, c) << 1

            If highBit = 128 Then

                DoubleState(r, c) = DoubleState(r, c) Xor 27

            End If

        Next

    Next

    For c = 0 To 3

        State(0, c) = DoubleState(0, c) Xor tmpState(3, c) Xor tmpState(2, c) Xor DoubleState(1, c) Xor tmpState(1, c)

        State(1, c) = DoubleState(1, c) Xor tmpState(0, c) Xor tmpState(3, c) Xor DoubleState(2, c) Xor tmpState(2, c)

        State(2, c) = DoubleState(2, c) Xor tmpState(1, c) Xor tmpState(0, c) Xor DoubleState(3, c) Xor tmpState(3, c)

        State(3, c) = DoubleState(3, c) Xor tmpState(2, c) Xor tmpState(1, c) Xor DoubleState(0, c) Xor tmpState(0, c)

    Next

End Sub

Private Sub Mix\_Columns(ByRef State(,) As Byte)

    Dim Col(3) As Byte

    Dim tmp\_state(3, 3) As Byte

    For c = 0 To 3

        For r = 0 To 3

            Col(r) = State(r, c)

        Next

        Mix\_One(Col)

        For r = 0 To 3

            tmp\_state(r, c) = Col(r)

        Next

    Next

    State = tmp\_state

End Sub

Private Sub Mix\_One(ByRef Col() As Byte)

    Dim tmpCol(3) As Byte

    Dim DCol(3) As Byte

    For i = 0 To 3

        Dim h As Byte

        tmpCol(i) = Col(i)

        h = Col(i) And 128

        DCol(i) = Col(i) << 1

        If h = 128 Then

            DCol(i) = DCol(i) Xor 27

        End If

    Next

    Col(0) = DCol(0) Xor tmpCol(3) Xor tmpCol(2) Xor DCol(1) Xor tmpCol(1)

    Col(1) = DCol(1) Xor tmpCol(0) Xor tmpCol(3) Xor DCol(2) Xor tmpCol(2)

    Col(2) = DCol(2) Xor tmpCol(1) Xor tmpCol(0) Xor DCol(3) Xor tmpCol(3)

    Col(3) = DCol(3) Xor tmpCol(2) Xor tmpCol(1) Xor DCol(0) Xor tmpCol(0)

End Sub

Private Sub ShiftRows(ByRef State(,) As Byte)

    Dim temp As Byte

    For i = 1 To 3

        For c = i - 1 To 0 Step -1

            temp = State(i, 0)

            State(i, 0) = State(i, 1)

            State(i, 1) = State(i, 2)

            State(i, 2) = State(i, 3)

            State(i, 3) = temp

        Next

    Next

End Sub

These are the basic functions, except key scheduling, that forms the basis of the AES encryption, minus the mode of operation. The process of these operations is described on page 15, and it was a simple procedure to convert the algorithm into the code above.

Next, I implemented the functions defined above to form the main algorithm for AES encryption:

''' <summary>

''' AES encryption algorithm

''' </summary>

Private Function Encrypt\_AES(ByVal State(,) As Byte, ByVal Key() As Byte) As Byte(,)

    Dim Rounds As Integer

    Dim IntState(3, 3) As Byte

    Dim IntKey(31) As Byte

    Array.Copy(State, IntState, 16)

    Array.Copy(Key, IntKey, 32)

    Dim RoundKey(,,) As Byte

    Dim Curkey(3, 3) As Byte

    IntKey = Rij\_KeySchedule(IntKey, 256)

    RoundKey = convertState(IntKey)

    Copy\_3to2(RoundKey, Curkey, 0, 4, 4)

    AddRoundKey(Curkey, IntState)

    For i = 1 To Rounds - 1

        Sub\_Bytes(IntState)

        ShiftRows(IntState)

        Mix\_Column(IntState)

        Copy\_3to2(RoundKey, Curkey, i, 4, 4)

        AddRoundKey(Curkey, IntState)

    Next

    Sub\_Bytes(IntState)

    ShiftRows(IntState)

    Copy\_3to2(RoundKey, Curkey, Rounds, 4, 4)

    AddRoundKey(Curkey, IntState)

    Return IntState

End Function

This procedure starts off by defining the maximum rounds that we need to iterate through, we also define the internal state as a State2 with row and column maximum count of 4. The next process is we take the given state in the argument and copy it into our new internal state and we take the key argument and copy its values into the internal key for our algorithm to use. Next, the provided key is stretched providing sufficient data for all of the rounds and then the first 3 transformations are performed. The program now loops through the rounds less one round as the last round is reserved for the same repeat as before except the **Mix\_Columns** transformation is not included. The encrypted data is then returned as a State2 to the calling class.

Finally, I implemented the block cipher mode of operation, Cipher Block Chaining, in its own subroutine which generates the required IV using a set of know integers which are then changed using some random secure numbers that are a mix of the CPU usage, RAM usage, mouse x and y positions and finally a random value from the secure random number generator specific to cryptography applications. Once the IV has been successfully generated the plaintext that was provided as a byte array is transformed into a State3 (simply put, an array of 2 dimensional arrays which hold 16 bytes of data at a time). Taking the first state, we reuse the function **AddRoundKey** because it XORs two of the states together, to XOR the current working state and the generated IV, we then take the value from that and encrypt it using the AES function above. Depending on what mode the CBC is operating in, if the function is being used for MAC generation and thusly the argument **CBC\_MAC** is set to true, then we take the final state after the encryption of the data is complete, otherwise each cipherstate is then converted back to a byte array and appended onto the last encrypted state to form the complete ciphertext as a byte array:

''' <summary>

''' Performs AES Cipher Block Chaining, or by specifiying CBC\_MAC as True we can perform CBC\_MAC for calculating a Message Authentication Code

''' </summary>

Public Function Encrypt\_Sign\_AES\_256\_CBC(ByVal Plaintext As Byte(), ByVal Key As Byte(), Optional ByVal CBC\_MAC As Boolean = False) As CBC\_Values

    Dim ret As New CBC\_Values

    If IsNothing(Plaintext) Or Plaintext.Count = 0 Then

        errorHappened(Plaintext, "Plaintext is Null", 1, GetCurrentMethod.Name)

        ret.didError = True

        Return ret

    End If

    If IsNothing(Key) Or Key.Length < 32 Then

        errorHappened(Key, "Key failed validation checks", 6, GetCurrentMethod.Name)

        ret.didError = True

        Return ret

    End If

    Dim IV() As Byte

    If CBC\_MAC = False Then

        IV = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}

        'We can get current total cpu usage in order to help derrive a new cryptographically secure IV

        Dim cpu As New PerformanceCounter()

        Dim rand As New Random

        With cpu

            .CategoryName = "Processor"

            .CounterName = "% Processor Time"

            .InstanceName = "\_Total"

        End With

        'We repeat the generation 50 times to be secure

        For c = 0 To 50

            'we populate the 16 array values using this secure method

            For i = 0 To 15

                'We can use cpu usage as a percentage, cursor positions, a random number and ram usage as a percentage to generate an cryptographic IV that is secure

                Dim ramusage As Integer = (My.Computer.Info.AvailablePhysicalMemory / My.Computer.Info.TotalPhysicalMemory) \* 100

                Dim tmpNum As Integer = IV(i) Xor Cursor.Position.X Xor Cursor.Position.Y Xor cpu.NextValue Xor ramusage Xor rand.Next

                IV(i) = Math.Abs(tmpNum Mod 255)

            Next

        Next

        writeDebugLog("Finished generating IV for session", GetCurrentMethod.Name)

    Else

        'If siging the message using MAC we can set IV to zero

        IV = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

        writeDebugLog("Set IV to zero for MAC signature generation", GetCurrentMethod.Name)

    End If

    Dim State3(,,) As Byte

    'Converts the plaintext into the states possible

    State3 = convertState(Plaintext)

    Dim State2(3, 3) As Byte

    'Get the first state from the three dimensional holding

    Copy\_3to2(State3, State2, 0, 4, 4)

    Dim operationCount As Integer = Math.Ceiling(Plaintext.Length / 16) 'calculates the number of states that plaintext was converted into

    Dim stateIV(,) As Byte = convertSingleState(IV) 'converts the IV into a single state as the IV is 16 counts long

    Dim AES256 As New AES\_Encrypt

    Dim cipherState2(3, 3) As Byte

    Dim ciphertext((Math.Ceiling(Plaintext.Count / 16) \* 16) - 1) As Byte 'We need to calculate the number of states that the plaintext has used up so that when we copy the encrypted State to the output we have the right number indexes

    For i = 0 To operationCount - 1

        If i = 0 Then

            'function reused from AES as it XORs two states together

            AddRoundKey(stateIV, State2)

        Else

            Copy\_3to2(State3, State2, i, 4, 4)

            AddRoundKey(cipherState2, State2)

        End If

        'Encryption

        cipherState2 = AES256.Encrypt\_AES(State2, Key)

        'convert the ciphertext into a 1D byte array and copy that two the return variable

        If Not CBC\_MAC Then

            'if encrypting then we need all the data back

            Array.Copy(ConvertFromState(cipherState2), 0, ciphertext, i \* 16, 16)

        End If

    Next

    If CBC\_MAC Then

        'if generating a MAC then all we need is the last block of the mix

        Array.Copy(ConvertFromState(cipherState2), 0, ciphertext, 0, 16)

    End If

    If Not CBC\_MAC Then

        writeDebugLog("Finished encrypting the plaintext with CBC, returning.", GetCurrentMethod.Name)

    Else

        writeDebugLog("Finished generating the MAC for the message, returning.", GetCurrentMethod.Name)

    End If

    ret.Data = ciphertext

    ret.IV = IV

    Return ret

End Function

When the IV is being generated, if the function if being used for MAC generation (**CBC\_MAC** is set to true) then the IV is set as a 16-byte array of zeros which is required for MAC generation, instead of a random IV.

#### AES Decryption

As before with AES encryption, I started off by writing the inverse function first, which translate the data from its encrypted form back to decrypted form, following the reverse of the process used in **Encrypt\_AES**:

''' <summary>

       ''' Perfroms the inverse of SBox

       ''' </summary>

       Private Function Inv\_SBox(ByVal input As Byte) As Byte

           Dim invsboxvals(255) As Byte

           invsboxvals = {82, 9, 106, 213, 48, 54, 165, 56, 191, 64, 163, 158, 129, 243, 215, 251, 124, 227, 57, 130, 155, 47, 255, 135, 52, 142, 67, 68, 196, 222, 233, 203, 84, 123, 148, 50, 166, 194, 35, 61, 238, 76, 149, 11, 66, 250, 195, 78, 8, 46, 161, 102, 40, 217, 36, 178, 118, 91, 162, 73, 109, 139, 209, 37, 114, 248, 246, 100, 134, 104, 152, 22, 212, 164, 92, 204, 93, 101, 182, 146, 108, 112, 72, 80, 253, 237, 185, 218, 94, 21, 70, 87, 167, 141, 157, 132, 144, 216, 171, 0, 140, 188, 211, 10, 247, 228, 88, 5, 184, 179, 69, 6, 208, 44, 30, 143, 202, 63, 15, 2, 193, 175, 189, 3, 1, 19, 138, 107, 58, 145, 17, 65, 79, 103, 220, 234, 151, 242, 207, 206, 240, 180, 230, 115, 150, 172, 116, 34, 231, 173, 53, 133, 226, 249, 55, 232, 28, 117, 223, 110, 71, 241, 26, 113, 29, 41, 197, 137, 111, 183, 98, 14, 170, 24, 190, 27, 252, 86, 62, 75, 198, 210, 121, 32, 154, 219, 192, 254, 120, 205, 90, 244, 31, 221, 168, 51, 136, 7, 199, 49, 177, 18, 16, 89, 39, 128, 236, 95, 96, 81, 127, 169, 25, 181, 74, 13, 45, 229, 122, 159, 147, 201, 156, 239, 160, 224, 59, 77, 174, 42, 245, 176, 200, 235, 187, 60, 131, 83, 153, 97, 23, 43, 4, 126, 186, 119, 214, 38, 225, 105, 20, 99, 85, 33, 12, 125}

           Return invsboxvals(input)

       End Function

       ''' <summary>

       ''' XORs the key with the state

       ''' </summary>

       Private Sub AddRoundKey(ByVal RoundKey(,) As Byte, ByRef State(,) As Byte)

           For c = 0 To 3

               For r = 0 To 3

                   State(r, c) = RoundKey(r, c) Xor State(r, c)

               Next

           Next

       End Sub

       ''' <summary>

       ''' Does the inverse of Sub\_Bytes by using inverse SBox to look up the data

       ''' </summary>

       Private Sub Inv\_Sub\_Bytes(ByRef State(,) As Byte)

           For c = 0 To 3

               For r = 0 To 3

                   State(r, c) = Inv\_SBox(State(r, c))

               Next

           Next

       End Sub

       ''' <summary>

       ''' Does the inverse of Mix\_Column

       ''' </summary>

       Private Sub Inv\_Mix\_Column(ByRef State(,) As Byte)

           Dim tmpState(3, 3) As Byte

           For c = 0 To 3

               For r = 0 To 3

                   tmpState(r, c) = State(r, c)

               Next

           Next

           For c = 0 To 3

               State(0, c) = gmul(tmpState(0, c), 14) Xor gmul(tmpState(3, c), 9) Xor gmul(tmpState(2, c), 13) Xor gmul(tmpState(1, c), 11)

               State(1, c) = gmul(tmpState(1, c), 14) Xor gmul(tmpState(0, c), 9) Xor gmul(tmpState(3, c), 13) Xor gmul(tmpState(2, c), 11)

               State(2, c) = gmul(tmpState(2, c), 14) Xor gmul(tmpState(1, c), 9) Xor gmul(tmpState(0, c), 13) Xor gmul(tmpState(3, c), 11)

               State(3, c) = gmul(tmpState(3, c), 14) Xor gmul(tmpState(2, c), 9) Xor gmul(tmpState(1, c), 13) Xor gmul(tmpState(0, c), 11)

           Next

       End Sub

       ''' <summary>

       ''' Does the inverse of shift rows

       ''' </summary>

       Private Sub Inv\_ShiftRows(ByRef State(,) As Byte)

           Dim temp As Byte

           For i = 1 To 3

               For c = i - 1 To 0 Step -1

                   temp = State(i, 3)

                   State(i, 3) = State(i, 2)

                   State(i, 2) = State(i, 1)

                   State(i, 1) = State(i, 0)

                   State(i, 0) = temp

               Next

           Next

       End Sub

Simply the code above performs the inverse of the changes made to encrypt the data, these aren’t always mathematical changes, for example **Inv\_ShiftRows** just undoes the cyclic shifts performed in **Shift\_Rows**.

Next, I used the functions above, like before, to create a subroutine that performs the steps to encrypt one 16-byte block of data.

''' <summary>

''' AES decryption algorithm

''' </summary>

Private Function Decrypt\_AES(ByVal State(,) As Byte, ByVal key() As Byte) As Byte(,)

    Dim Rounds As Integer

    Dim IntState(3, 3) As Byte

    Dim IntKey(31) As Byte

    Array.Copy(State, IntState, 16)

    Array.Copy(key, IntKey, 32)

    Dim RoundKey(,,) As Byte

    Dim Curkey(3, 3) As Byte

    IntKey = Rij\_KeySchedule(IntKey, 256)

    RoundKey = convertState(IntKey)

    Copy\_3to2(RoundKey, Curkey, Rounds, 4, 4)

    AddRoundKey(Curkey, IntState)

    Inv\_ShiftRows(IntState)

    Inv\_Sub\_Bytes(IntState)

    For i = Rounds - 1 To 1 Step -1

        Copy\_3to2(RoundKey, Curkey, i, 4, 4)

        AddRoundKey(Curkey, IntState)

        Inv\_Mix\_Column(IntState)

        Inv\_ShiftRows(IntState)

        Inv\_Sub\_Bytes(IntState)

    Next

    Copy\_3to2(RoundKey, Curkey, 0, 4, 4)

    AddRoundKey(Curkey, IntState)

    Return IntState

End Function

As in encryption this subroutine starts or by generating the round keys for the whole algorithm, it then continues by starting from the end of the encryption algorithm and taking off the changes applied to it, so it starts off by XORing the final round key with the state and then performs the invers of both the shift rows and the substitution vector. After having done this it then performs the loop backwards using the round keys from the end down to the start and finishes off by XORing the state with the first key as seen in encryption.

Finally, it only left the decryption phase of CBC:

''' <summary>

''' Decrypts a byte array encrypted with AES and Cipher Block Chaining

''' </summary>

Public Function Decrypt\_AES\_256\_CBC(ByVal Ciphertext As Byte(), ByVal key As Byte(), ByVal IV() As Byte) As CBC\_Values

    Dim ret As New CBC\_Values

    If IsNothing(Ciphertext) Or Ciphertext.Count = 0 Then

        errorHappened(Ciphertext, "Ciphertext is Null", 1, GetCurrentMethod.Name)

        ret.didError = True

        Return ret

    End If

    If IsNothing(key) Or key.Length < 32 Then

        errorHappened(key, "Key failed validation checks", 6, GetCurrentMethod.Name)

        ret.didError = True

        Return ret

    End If

    Dim Aes256 As New AES\_Decrypt

    Dim oldState2(3, 3) As Byte

    Dim State3(,,) As Byte = convertState(Ciphertext)

    Dim State2(3, 3) As Byte

    Copy\_3to2(State3, State2, 0, 4, 4)

    Dim operationCount As Integer = Math.Ceiling(Ciphertext.Length / 16)

    Dim stateIV(,) As Byte = convertSingleState(IV)

    Dim plainState2(3, 3) As Byte

    Dim Plaintext(Ciphertext.Count - 1) As Byte

    For i = 0 To operationCount - 1

        Copy\_3to2(State3, State2, i, 4, 4)

        plainState2 = Aes256.Decrypt\_AES(State2, key)

        If i = 0 Then

            AddRoundKey(stateIV, plainState2)

        Else

            AddRoundKey(oldState2, plainState2)

        End If

        Array.Copy(ConvertFromState(plainState2), 0, Plaintext, i \* 16, 16)

        oldState2 = State2.Clone()

    Next

    writeDebugLog("Finished decrypting the text", GetCurrentMethod.Name)

    ret.Data = removeZeros(Plaintext)

    ret.IV = IV

    Return ret

End Function

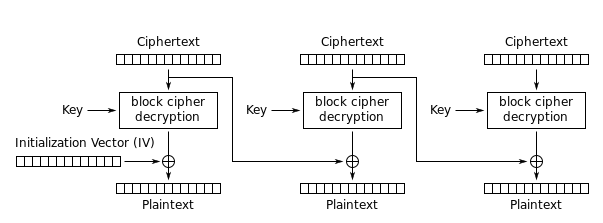


Figure 17: CBC Decryption

The method for decrypting a the cipher text follows the diagram above, it performs the operations of before in opposite to encryption and requires the IV that was generated during the encryption phase. Because the encryption and decryption with AES requires set blocks of 16-bytes there can often be a set of zeros on the end of a decrypted message to pad the message out to fill the complete 16-bytes, this required a subroutine to remove all of the zeros from the end:

''' <summary>

''' Removes the excess zeros from the end of a message

''' </summary>

Private Function removeZeros(ByRef plaintext() As Byte) As Byte()

    Dim number As Integer

    Dim i As Integer = plaintext.Length - 1

    writeDebugLog("Removing zero values from the end of the decrypted message", GetCurrentMethod.Name)

    Do Until plaintext(i) <> 0 Or i = 0

        number += 1

    Loop

    Dim deZeroed(plaintext.Count - 1 - number) As Byte

    i = 0

    For i = 0 To plaintext.Length - 1 - number

        deZeroed(i) = plaintext(i)

    Next

    writeDebugLog("Removed " & number & " zeros from the message", GetCurrentMethod.Name)

    Return deZeroed

End Function

#### ECEIS

Elliptic Curve Integrated Encryption Standard is the complete process of encrypting a message, from key generation to performing ECDH and deriving a hash for the message, it normally produces 4 outputs that are bound together and sent to the receiving party, these consist of the ciphertext, the IV to generate it, a public key of the sender and the MAC of the message to ensure the integrity of the message when it is received by the recipient.

As some of the processes in ECIES produce multiple results I started off by writing the algorithms for those. The first one I implemented was the key derivation function that splits the shared secret into two parts, one key is used for the generation of the MAC and the other is used in the encryption of the plaintext. The process of splitting the key is as simple as appending the values 1 and 2 onto the start of the provided key and generate a SHA256 hash for the values. In order to insert these values, I first created my own subroutine to perform this procedure:

''' <summary>

''' Inserts a byte at a set location in the array

''' </summary>

Private Function insert\_Byte(ByVal Array() As Byte, ByVal Value As Byte, ByVal Position As Integer) As Byte()

    Dim newarr(Array.Count) As Byte

    Dim ins As Boolean

    For i = 0 To newarr.Count - 1

        If i = Position Then

            newarr(i) = Value

            ins = True

        Else

            If ins = False Then

                newarr(i) = Array(i)

            Else

                newarr(i) = Array(i - 1)

            End If

        End If

    Next

    Return newarr

End Function

This simply takes the array we want to add something onto the start of and makes a new array (**newarr**) the size of the array given plus one, it then sets the value when position is equal to **i**, otherwise it copies the values into the new array. After having added this it made it easy to generate the two keys like so:

''' <summary>

''' A hash function which turns the x value of the public key into two secret keys

''' </summary>

Private Function hash\_KDF(ByVal Secret As ECPoint) As HKDF

    Dim hash256 As New SHA256Managed

    Dim hashValue As Byte()

    Dim tmpSec() As Byte

    Dim ret As New HKDF

    If Secret.IsNull() Then

        errorHappened(Secret, "The value of the secret is Null", 1, GetCurrentMethod.Name)

        ret.didError = True

        Return ret

    End If

    tmpSec = insert\_Byte(Secret.x.ToByteArray, Convert.ToByte(1), 0)

    hashValue = hash256.ComputeHash(tmpSec)

    ret.Secret01 = New BigInteger(hashValue)

    tmpSec = insert\_Byte(Secret.x.ToByteArray, Convert.ToByte(2), 0)

    hashValue = hash256.ComputeHash(tmpSec)

    ret.Secret02 = New BigInteger(hashValue)

    Return ret

End Function

In order to be able to return the two keys at once I had to create a structure that contained the two keys then return in instance of that structure:

Private Structure HKDF

    Public Secret01 As BigInteger

    Public Secret02 As BigInteger

    Public didError As Boolean

End Structure

I also added a Boolean value to check if the derivation failed rather than throwing an error and catching it but both work.

Having added these, all I needed were two more functions before starting implementing the ECIES algorithm, one function allows me to easily compare two arrays and the other checks if every value in the array is set to zero. The comparing arrays is used for checking the given MAC and the generated MAC during the decryption phase, and the other checks that the IV used in calculating the first MAC is zero, in order to try and detect any possible errors when generating the MAC.

''' <summary>

''' Check if all the values in an array are zero

''' </summary>

Private Function isArrayZeroed(ByVal byteArray() As Byte) As Boolean

    For i = 0 To byteArray.Length - 1

        If byteArray(i) <> 0 Then

            Return False

        End If

    Next

    Return True

End Function

''' <summary>

''' Compares two arrays of the same length

''' </summary>

Private Function compareArrays(ByVal Array1() As Byte, ByVal Array2() As Byte) As Boolean

    If Array1.Count <> Array2.Count Or Array1.Count = 0 Then

        Return False

    End If

    For i = 0 To Array1.Count - 1

        If Array1(i) <> Array2(i) Then

            Return False

        End If

    Next

    Return True

End Function

After having added the functions necessary for ECIES to be performed correctly it left encryption and decryption to be added, I started off by adding encryption as seen below:

Public Function Encrypt\_AES\_256(ByVal plaintext() As Byte, ByVal localkey As Keys, ByVal Curve As Weierstrass\_Curve, ByVal Recipient\_Public\_key As ECPoint) As TEncryption

            Dim ECIES\_AES\_Return As New TEncryption

            If IsNothing(plaintext) Or IsNothing(localkey) Or IsNothing(Curve) Or IsNothing(Recipient\_Public\_key) Then

                errorHappened(Me, "One of the parameters was Null", 1, GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            If Recipient\_Public\_key.IsNull Or Recipient\_Public\_key.IsPointInfinity Then

                errorHappened(Recipient\_Public\_key, "The recipent key failed NullReference validation", 1, GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            If localkey.PrivateKey.IsZero Or localkey.PublicKey.IsNull Then

                errorHappened(localkey, "The client keys failed NullReference validation", 1, GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            If Curve.Curve\_ID = Nothing Then

                errorHappened(Curve, "The curve failed NullReference validation", 1, GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            If Recipient\_Public\_key.IsPointOnCurve(Curve.Parameters) = False Then

                errorHappened(Recipient\_Public\_key, "Provided public key is not a point on the curve", 4, GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            Dim ex As errorEnum = Domain\_Parameters.Validate\_Params(Curve.Parameters)

            If Not ex = 0 Then

                errorHappened(Curve.Parameters, "Parameters did not pass validation", ex, GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            If IsNothing(plaintext) Then

                errorHappened(plaintext, "The plaintext is empty", 1, GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            Dim secretKey As New ECPoint(0, 0)

            Dim extSecret As New HKDF

            Dim AES256 As New AES\_Encrypt

            Dim CBC\_AES\_Return As New AES.CBC\_Values

            Dim CBC\_MAC\_Return As New AES.CBC\_Values

            ECIES\_AES\_Return.publicKey = localkey.PublicKey

            secretKey = ECDH.generate\_SharedSecret(localkey.PrivateKey, Recipient\_Public\_key, Curve.Parameters)

            If secretKey.isErroneous() Then

                errorHappened(secretKey, "An error occured whilst generating shared secret", 7, GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            extSecret = hash\_KDF(secretKey)

            If extSecret.didError = True Then

                writeDebugLog("Returning from sub, HKDF had an error", GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            CBC\_AES\_Return = AES256.Encrypt\_Sign\_AES\_256\_CBC(plaintext, AES.Convert\_BigInteger\_ToByteArray(extSecret.Secret01))

            If CBC\_AES\_Return.didError Then

                writeDebugLog("Returning from sub, ENCRYPT\_AES\_CBC had an error", GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            ECIES\_AES\_Return.Data = CBC\_AES\_Return.Data

            ECIES\_AES\_Return.IV = CBC\_AES\_Return.IV

            Dim extData() As Byte = addLenghtToEnd(CBC\_AES\_Return.Data)

            CBC\_MAC\_Return = AES256.Encrypt\_Sign\_AES\_256\_CBC(extData, AES.Convert\_BigInteger\_ToByteArray(extSecret.Secret02), True)

            If CBC\_AES\_Return.didError Then

                writeDebugLog("Returning to main, MAC generation had an error", GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            'We can ignore CBC\_MAC\_Return.IV because that should be 0 as the IV when calculating a MAC is zero, we can instead use it for error checking

            If isArrayZeroed(CBC\_MAC\_Return.IV) Then

                ECIES\_AES\_Return.HMAC = CBC\_MAC\_Return.Data

            Else

                errorHappened(CBC\_MAC\_Return, "IV is not zero", 5, GetCurrentMethod.Name)

                ECIES\_AES\_Return.didError = True

                Return ECIES\_AES\_Return

            End If

            ECIES\_AES\_Return.didError = False

            Return ECIES\_AES\_Return

        End Function

The section before all of the **Dim** statements are used for error checking the values of the parameters passed to the function, if any of these checks fail the process of encryption is forced to finish abruptly as encryption cannot continue and the arguments need to be resupplied. After having passed all of the initial checks we start of by generating the shared secret of our private key and the recipients public key. The x value from the **ECPoint** it generates is then used in the function **hashKDF** which takes that secret key and hashes it into two different 32-bit values, the first value is used for the encryption of the plaintext, producing a ciphertext. This ciphertext is then modified and the length of the ciphertext is appended onto the end of the ciphertext so that the MAC generated also covers the length of the data. Generating a MAC uses the same AES function as encryption but we then supply a **True** Boolean value indicating that we are now performing MAC generation using the same algorithm. In order to return the 4 values generated when encrypting the data, I created a structure to contain these values:

Public Structure TEncryption

    Public Data() As Byte

    Public IV() As Byte

    Public publicKey As ECPoint

    Public HMAC() As Byte

    Public didError As Boolean

End Structure

This is also supplied as a parameter to the decryption that we will cover below.

Decryption follows the basic suit of encryption, yet MAC generation is done before the data is decrypted in order to ascertain if the data was tampered with in transit.

Public Function Decrypt\_AES\_256(ByVal CBC\_AES As TEncryption, ByVal localkey As Keys, ByVal Curve As Weierstrass\_Curve, ByVal Recipient\_Public\_key As ECPoint) As Byte()

            If IsNothing(CBC\_AES) Or IsNothing(localkey) Or IsNothing(Curve) Or IsNothing(Recipient\_Public\_key) Then

                errorHappened(Me, "One of the parameters was Null", 1, GetCurrentMethod.Name)

                Return Nothing

            End If

            If CBC\_AES.Data.Count = 0 Then

                errorHappened(CBC\_AES, "Ciphertext was Null", 1, GetCurrentMethod.Name)

                Return Nothing

            End If

            If localkey.PublicKey.isErroneous Or localkey.PublicKey.IsNull Or localkey.PrivateKey.IsZero Or localkey.PublicKey.IsPointInfinity Then

                errorHappened(localkey, "Provided keys were Null", 1, GetCurrentMethod.Name)

                Return Nothing

            End If

            Dim ex As errorEnum = Domain\_Parameters.Validate\_Params(Curve.Parameters)

            If ex <> 0 Then

                errorHappened(Curve.Parameters, "Curve parameters failed validation", ex, GetCurrentMethod.Name)

                Return Nothing

            End If

            If Not localkey.PublicKey.IsPointOnCurve(Curve.Parameters) Then

                errorHappened(localkey, "Point was not calculated correctly, it does not exists as a point on the curve", 4, GetCurrentMethod.Name)

                Return Nothing

            End If

            If Recipient\_Public\_key.isErroneous Or Recipient\_Public\_key.IsNull Then

                errorHappened(Recipient\_Public\_key, "Public key of the recieving party is Null", 1, GetCurrentMethod.Name)

                Return Nothing

            End If

            If Recipient\_Public\_key.IsPointInfinity Then

                errorHappened(Recipient\_Public\_key, "Public key was the point at infinity", 3, GetCurrentMethod.Name)

                Return Nothing

            End If

            If Not Recipient\_Public\_key.IsPointOnCurve(Curve.Parameters) Then

                errorHappened(Recipient\_Public\_key, "Public key was not on the give curve", 4, GetCurrentMethod.Name)

                Return Nothing

            End If

            Dim secretKey As New ECPoint(0, 0)

            Dim extSecret As New HKDF

            Dim AES256 As New AES\_Encrypt

            Dim DAES256 As New AES\_Decrypt

            Dim CBC\_AES\_Return As New AES.CBC\_Values

            Dim CBC\_MAC\_Return As New AES.CBC\_Values

            secretKey = ECDH.generate\_SharedSecret(localkey.PrivateKey, Recipient\_Public\_key, Curve.Parameters)

            If secretKey.isErroneous() Then

                errorHappened(secretKey, "An error occured whilst generating shared secret", 7, GetCurrentMethod.Name)

                Return Nothing

            End If

            extSecret = hash\_KDF(secretKey)

            If extSecret.didError = True Then

                writeDebugLog("Returning from sub, HKDF had an error", GetCurrentMethod.Name)

                Return Nothing

            End If

            Dim extData() As Byte = addLenghtToEnd(CBC\_AES.Data)

            CBC\_MAC\_Return = AES256.Encrypt\_Sign\_AES\_256\_CBC(extData, AES.Convert\_BigInteger\_ToByteArray(extSecret.Secret02), True)

            If CBC\_AES\_Return.didError Then

                writeDebugLog("Returning to main, MAC generation had an error", GetCurrentMethod.Name)

                Return Nothing

            End If

            'We can ignore CBC\_MAC\_Return.IV because that should be 0 as the IV when calculating a MAC is zero, we can instead use it for error checking

            If Not isArrayZeroed(CBC\_MAC\_Return.IV) Then

                errorHappened(CBC\_MAC\_Return, "IV  is not zero", 5, GetCurrentMethod.Name)

                Return Nothing

            End If

            If Not compareArrays(CBC\_AES.HMAC, CBC\_MAC\_Return.Data) Then

                writeDebugLog("Message MAC did not equal the MAC provided by the sender, either the data has been tampered with or something happened in transit", GetCurrentMethod.Name)

                Return Nothing

            End If

            CBC\_AES\_Return = DAES256.Decrypt\_AES\_256\_CBC(CBC\_AES.Data, AES.Convert\_BigInteger\_ToByteArray(extSecret.Secret01), CBC\_AES.IV)

            If CBC\_AES\_Return.didError Then

                writeDebugLog("Returning from sub, DECRYPT\_AES\_CBC had an error", GetCurrentMethod.Name)

                Return Nothing

            End If

            Return CBC\_AES\_Return.Data

        End Function

As in encryption the majority of the code above is error checking in order to try and catch any possible rouge values when encrypting data, therefore preventing failures and crashes.

In order to generate the MAC for the message correctly and not crash I had to swap the **insert\_Byte** statement out for the **addLengthToEnd**, as **insert\_Byte** would crash if the length of the encrypted data was longer than 256, instead **addLengthToEnd** uses a 32-bit signed integer instead, using four bytes:

Private Function addLenghtToEnd(ByVal Array() As Byte) As Byte()

    Dim newarr(Array.Count + 3) As Byte

    Dim len() As Byte = BitConverter.GetBytes(Array.Length)

    Array.CopyTo(newarr, 0)

    len.CopyTo(newarr, Array.Length)

    Return newarr

End Function

## TCP Design Principles

### Server UI Design

I started off with the design of my server UI to make it easy for a simple network admin to use, I started off my design in visual studio like so:

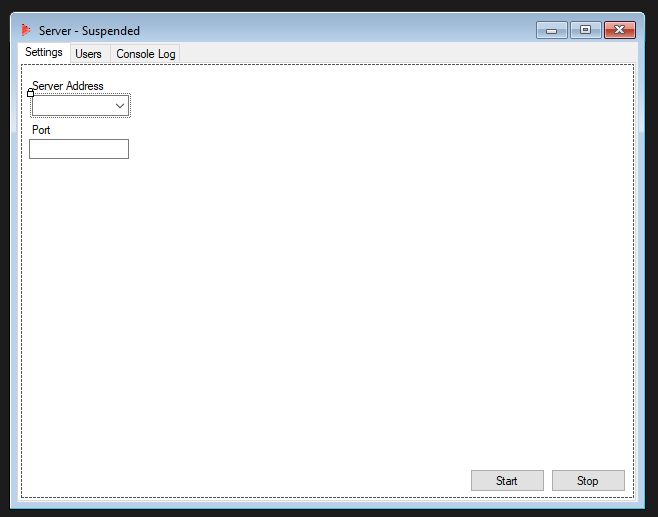


Figure 18: Server Control UI

It featured 3 tabs, the first was just generic server controls, such as starting and stopping the server and easy way of selecting an auto-populated list of local IP addresses on the system. A simple text box was also used to make it easy for the user to choose a port or use the predetermined port 3434. The second tab lists the current logged on users, their corresponding IP addresses and their public key which is a useful tool for debugging to quickly check that data was being received properly. Lastly in the final tab is a log for the console where various different methods are outputted to, using a rich text box.

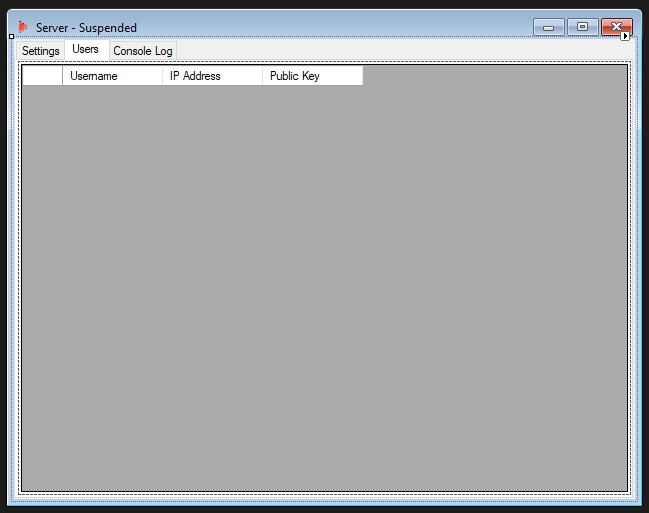


Figure 19: Console Log UI

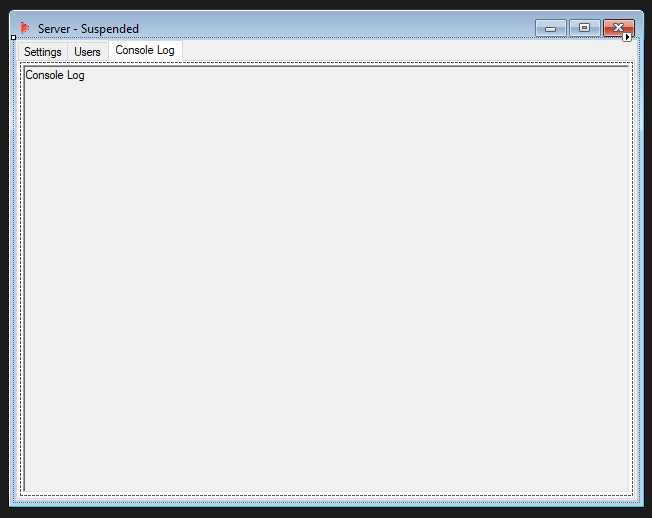


Figure 20: Console Log UI

I modified the first form slightly to try and incorporate a box with a list of the interfaces and the corresponding addresses associated with them:

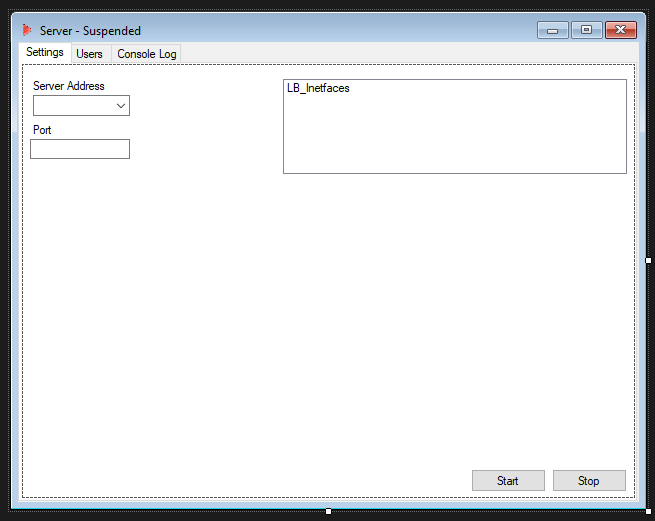


Figure 21: Updated Sever Control UI

The box on the right now lists the interfaces of the computer in the form interface identifier (i.e. loopback) colon IP address. As an example:

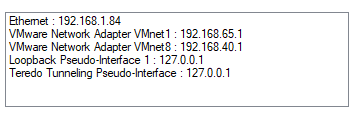


Figure 22: Box listing interfaces

Each of the IP addresses present in the box were also added to the drop-down list under server addresses as each address can be used to create the TCP listener on. Behind the scenes to populate these boxes, I used the **System.NetworkInformation.NetworkInterface** namespace in order to enumerate all of the network interfaces like so:

Dim NICs() As System.Net.NetworkInformation.NetworkInterface = System.Net.NetworkInformation.NetworkInterface.GetAllNetworkInterfaces()

Using NICs I could then loop through all of them making sure that I collected the interface name and the IP address associated with the interface like so:

Private Sub Form1\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

'Get the current time and date so we can datestamp the start of the log

Dim timeDate As DateTime = DateTime.Now

'Add the datestamp to the start of the log in the UI

RTB\_Log.Text = timeDate.Date.ToShortDateString & Space(1) & timeDate.TimeOfDay.ToString & vbNewLine

'Set the style of the machine IP address selection box, there was a problem if a user entered their own value because we get the selected value rather than the text converting the string to IP address would return null error, so to stop this change the box design to drop down instead

CB\_LocalAddress.DropDownStyle = ComboBoxStyle.DropDownList

'Get all of the network interfaces on the system such as Ethernet, wireless, bluetooth, loopback address, etc.

Dim NICs() As System.Net.NetworkInformation.NetworkInterface = System.Net.NetworkInformation.NetworkInterface.GetAllNetworkInterfaces()

RTB\_Log.AppendText("A list of interfaces were found with these IP Addresses: ")

For i = 0 To NICs.Length - 1

Dim IP As String

'Get the interfaces that are currently active and not deactivated

If NICs(i).OperationalStatus = Net.NetworkInformation.OperationalStatus.Up Then

For l = 0 To NICs(i).GetIPProperties.UnicastAddresses.Count - 1

'loop through the addresses, some versions of this list the MACs first and some list the IP first so we need to make sure we are getting the IPs not the MACs

Dim address As New IPAddress({0, 0, 0, 0})

System.Net.IPAddress.TryParse(NICs(i).GetIPProperties.UnicastAddresses(l).Address.ToString, address)

If address.AddressFamily = Sockets.AddressFamily.InterNetwork Then

IP = NICs(i).GetIPProperties.UnicastAddresses(l).Address.ToString()

Exit For

End If

Next

'Add the IP to the selection box

CB\_LocalAddress.Items.Add(IP)

'Add the interface name and the corresponding IP to the list box

LB\_Inetfaces.Items.Add(NICs(i).Name & " : " & IP)

RTB\_Log.AppendText(IP & Space(1))

End If

Next

RTB\_Log.AppendText(vbNewLine)

'set the default port

TXT\_Port.Text = "3434"

End Sub

The process to setup the form follows thusly, the date and time is added to the start of the console log and the style of the combo box is set, this was a problem before because if the user typed their own address into the box, the provided IP was returned null, due to the fact that when getting the needed IP the selected dropdown option was used and because nothing was being selected it caused the error, this change now disallows users to type their own strings in and thus only be able to select an address from the given list. Next, the list of the interfaces of the local system are obtained and looping through those interfaces, the NIC is confirmed to be enabled, if so then the unicast addresses of the NIC (which should only be the MAC and the IP address) are converted to the **IPAddress** object and it’s then possible to confirm if the given address is a MAC or an IPv4 address. This was needed as the index of unicast addresses was not the same across machines, so at points the MAC was listed in index position 0 and sometimes the IP was listed in that index. Lastly, the correct address is the written to the interfaces box along with the corresponding NIC name and the address is also added to the drop-down box.

After the form loads the port text box is set with the value 3434 and as mentioned above all the of the interfaces are added to the combo box and the console log is updated with some new information.

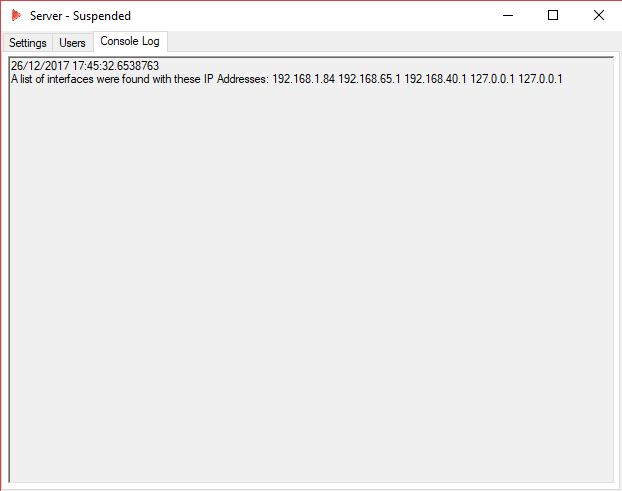
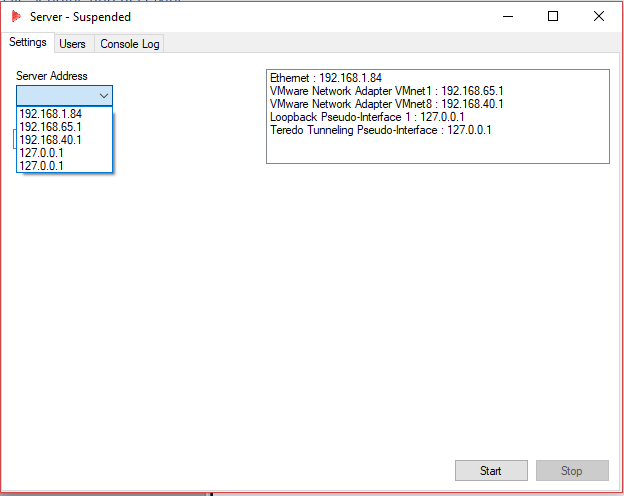


Figure 23: Running UI

My initial design before adding the interfaces box was using a combo box listing the available addresses to the system like so:

Dim hostName As String = System.Net.Dns.GetHostName()

Dim hostIParr() As System.Net.IPAddress = System.Net.Dns.GetHostAddresses(hostName)

For Each hostIp As System.Net.IPAddress In hostIParr

If exc Then

RTB\_Log.AppendText(hostIp.ToString & Space(5))

End If

CB\_LocalAddress.Items.Add(hostIp)

Next

In order to make it easier for a new administrator to use the server I wanted to list the interfaces and the corresponding IP addresses so I modified the code into the version above.

Once the user decides what address they want the server to run on and the port to run on its possible for them to commence running the server using the **START** button in the bottom right side of the main interface. If everything was successful it changes the icon of the server’s UI into a green triangle and the start button is set to disabled and the stop button is then set to be enabled. If the user has not selected an interface a message is written to the log about the error and a message box appears, notifying the user that an error has occurred and that the user must check the server log for details. Here are a few examples:

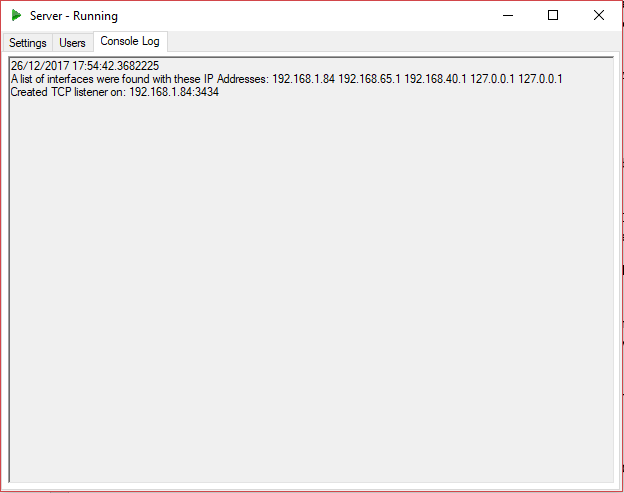
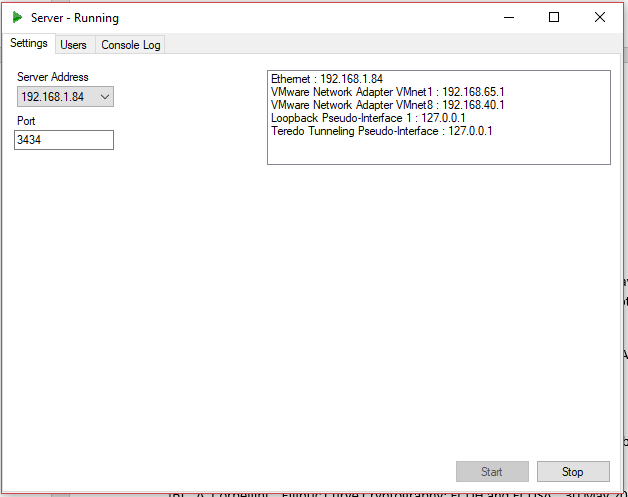


Figure 24: Changes seen when the server is running

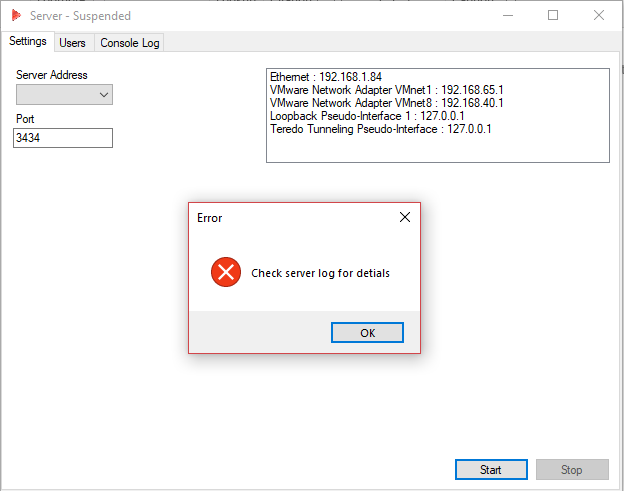


Figure 25: Error when not selecting IP

Some of the code behind the **START** button on the UI seen above takes the form of:

Private Sub BTN\_StartServer\_Click(sender As Object, e As EventArgs) Handles BTN\_StartServer.Click

'Check if the server is currently running and cancel the setup if it already is

If serverSocket IsNot Nothing Then

RTB\_Log.AppendText("Instance of server already running" & vbNewLine)

MessageBox.Show("Check server log for detials", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)

Return

End If

'Get the selected item from the local IP box

Dim IPByte() As Byte = SocketMT.Convert\_IP\_DecimalNotated(CB\_LocalAddress.SelectedItem)

'Just some error checking

If IsNothing(IPByte) Then

MessageBox.Show("Check server log for detials", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)

Return

End If

'Check if the port is numeric, error if its not

If Not IsNumeric(TXT\_Port.Text) Then

RTB\_Log.AppendText("Value for port is not allowed" & vbNewLine)

MessageBox.Show("Check server log for detials", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)

Return

End If

'Parse the IP string to IP

localIP = New System.Net.IPAddress(IPByte)

'New instance of SocketMT

serverSocket = New SocketMT

'Got to make sure that the listener is started correctly before continuing

If Not serverSocket.TcpListener\_Start(TXT\_Port.Text, localIP) Then

MessageBox.Show("Check server log for detials", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)

Return

End If

'Change the desgin so the user cant start the server again

BTN\_StartServer.Enabled = False

BTN\_StopServer.Enabled = True

Me.Icon = GetEmbeddedIcon("Running.ico")

Me.Text = "Server - Running"

End Sub

Private Function GetEmbeddedIcon(ByVal strName As String) As Icon

'Gets the icons for the program stored as an embedded resource in the final executable

Dim p As System.Reflection.Assembly

p = System.Reflection.Assembly.GetExecutingAssembly()

Dim ic As Icon

ic = New System.Drawing.Icon(p.GetManifestResourceStream(Me.GetType(), strName))

Return ic

End Function

There are 2 sets of functions being used here, the first one, called **BTN\_StartServer\_Click**  is called when the **START** button is clicked, the second one, called **GetEmbeddedIcon** returns the imbedded .ico that can be used to change the icon of the server. In the subroutine that handles the click event for the start button starts off by checking if the **serverSocket** instance is not set to nothing, if that statement is valid, the execution must be stopped because an error will be thrown later if we try to start a new instance of **TcpListener** on the same port as an instance that is already running. If **serverSocket** is set to nothing then we can go ahead and convert the IP address in the combo box to an array of 4 bytes, each index in the array is the number in binary that is normally between periods. It wasn’t necessary to write my own function to convert the string to a byte array as its possible to use the functions of the built in **IPAddress** class, but I wanted to run some error check on the IP address. If one of the checks fails then the **IPByte** should be a **NULL** value and therefore execution must be stopped and the user must be prompted to check the server log for details about the problem. The text box containing the port is then checked to see if the value of it is numeric, if so we then convert the byte array IP address to the **IPAddress** class, create a new instance of **SocketMT** which is a class I created that I will explore in further on. The socket can then be started by calling **SocketMT.TcpListener\_Start** with the parameters consisting of the port and the IP address of the host machine, if it was successful then the function returns **true** if not the function returns **false** and the user is told to check the log again. If everything went well, the **START** button is set to disabled, the **STOP** button is set to enabled and the icon of the form is changed using the **GetEmbeddedIcon** function.

Behind the **STOP** button looks something like:

Private Sub BTN\_StopServer\_Click(sender As Object, e As EventArgs) Handles BTN\_StopServer.Click

'stop the server and reset UI stuff

If serverSocket Is Nothing Then

RTB\_Log.AppendText("Server instance has not been set, server is not running" & vbNewLine)

MessageBox.Show("Check server log for detials", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)

Return

End If

serverSocket.TcpListener\_Stop()

serverSocket = Nothing

BTN\_StartServer.Enabled = True

BTN\_StopServer.Enabled = False

Me.Icon = GetEmbeddedIcon("Stopped.ico")

Me.Text = "Server - Suspended"

End Sub

This basically does the opposite of what the start does, it resets the UI, disabling the stop button and re-enabling the start button, sets the **serverSocket** instance equal to nothing so it passes the validation checks that are in **BTN\_StartServer\_Click**, and of course it stops the server by calling **SocketMT.TcpListener\_Stop.**

Upon form closing we check to see if the server is currently running, and if so we stop the server with **SocketMT.TcpListener\_Stop** and then allow the form to completely close.

### Design of the TCP server

#### Extending the functions of TcpListener

Initial thoughts for the server was to use a multithreaded approach similar to that seen on [19], but with an expanded design. In order to have multiple users we would need to make a class of functions that was completely multithreaded in order for users to communicate with the server asynchronously. Starting off I extended the functions of **TcpListener** with my own class that inherits **TcpListener** in order to make it completely multithreaded.

Imports System.Net.Sockets

Imports System.Net

Imports System.Threading

Public Class NewTcpListener

'We are extending the functionality of the TcpListener to make it multithreaded

Inherits System.Net.Sockets.TcpListener

Public Event clientConnecting()

Private ended As Boolean

'Public frmInstance As Form1

'Private WithEvents cl As New ClientConnection

Public Sub New(ByVal port As Integer, ByVal localIP As IPAddress)

'create a new intstance of TcpListener

MyBase.New(localIP, port)

End Sub

Public Sub startClientPoll(ByRef Sender As TcpListener)

'start the poll for new client connetions

Dim TThread As New Thread(New ParameterizedThreadStart(AddressOf pollClientConnect))

TThread.Start(Sender)

End Sub

Private Sub pollClientConnect(tcplistener As TcpListener)

Do

'If a client is pending connection then raise the event and allow socketMT to allow it to connect

If tcplistener.Pending() Then

RaiseEvent clientConnecting()

End If

Loop Until ended = True

End Sub

Public Sub stopListener()

'stop the thread and stop the instance of TcpListener

ended = True

MyBase.Stop()

End Sub

End Class

The class for **NewTcpListener** has the features above, the idea behind this design was that it could always run in a background thread and thus keep waiting until a client connects, which at that point it raises an event which is handled by the caller class. My initial designs featured this as the main thread and upon a client trying to connect a new class for each client could be created housing the required information about the client, the approach I ended up taking differed slightly from what was initially intended and featured a better approach to this situation. Initially there was some teething problems with this design, I was unsure of how to pass parameters to threads and after having figured out that, it seemed as though there was no way of passing a reference to an object:

Private Sub pollClientConnect(tcplistener As TcpListener)

Do

'If a client is pending connection then raise the event and let socketMT deal with it

If tcplistener.Pending() Then

RaiseEvent clientConnecting()

End If

Loop Until ended = True

End Sub

In fact, the natural way of passing parameters to threads is by reference and that’s why an error was thrown when using the keyword byref, which led me to believe that threads could only use byval. Making matters worse, by accident I had put **TcpListener.Stop** at the end of the calling subroutine so each time it got to polling **TcpListener.Pending** I was given an object disposed error, mentioning that I was unable to call a function of a disposed object, but rectifying that it works fine.

#### Managing connected clients

The next design consideration was how we keep a list of all the clients connected, their IP addresses, their usernames, and the reference to the object that is currently containing the multithreaded code that deals with the requests the client has. The first idea was to create an array of this data then every time the data had to be accessed we loop through it to find the information needed. This idea presented two problems, the first was that, short of dimensioning it, the array is a static data structure which would inevitably need to grow and shrink at runtime depending on the number of clients. The other problem was that too look up data using an array was not efficient, it was possible that I could give the client a unique number but then I would have to write a whole class to manage assigning numbers, removing numbers, changing array sizes. A better replacement would be to use a hash table, where a hash function could be applied to a key and give it a position in the table, this object is dynamic which makes it perfect for our situation and also features automatic error checking, if a key is already taken then an error is thrown and we know a client with those details is already connected.

Public Class ClientLookupTable

Private clientHashTable As Hashtable

Public Structure tableValues

Public username As String

Public publicKey() As Byte

Public clientReference As MTClientReciever

End Structure

'An enum of errors depending on the outcome of certain actions

Public Enum clientLookupTableErrors

CLIENT\_DOES\_NOT\_EXIST

TABLE\_CONTAINS\_CLIENT

NO\_ERROR

CLIENT\_CONNECTED

CLIENT\_NOT\_CONNECTED

End Enum

Public Sub New()

clientHashTable = New Hashtable()

End Sub

Public Function AddNewClient(ByVal IPAddr As String, ByVal value As tableValues) As clientLookupTableErrors

'return an error if the table already contains the client

If clientHashTable.ContainsKey(IPAddr) Then

Return clientLookupTableErrors.TABLE\_CONTAINS\_CLIENT

End If

'else add the new client

clientHashTable.Add(IPAddr, value)

Return clientLookupTableErrors.NO\_ERROR

End Function

Public Function CheckClientConnected(ByVal IPAddr As String) As clientLookupTableErrors

'return that the client is connected

If clientHashTable.ContainsKey(IPAddr) Then

Return clientLookupTableErrors.CLIENT\_CONNECTED

End If

'Client probably not connected

Return clientLookupTableErrors.CLIENT\_NOT\_CONNECTED

End Function

Public Function RemoveClient(ByVal IPAddr As String) As clientLookupTableErrors

'Check if the client doesnt exist

If Not clientHashTable.ContainsKey(IPAddr) Then

Return clientLookupTableErrors.CLIENT\_DOES\_NOT\_EXIST

End If

'else remove the client

clientHashTable.Remove(IPAddr)

Return clientLookupTableErrors.NO\_ERROR

End Function

Public Function GetClientDetails(ByVal IPAddr As String) As Object

If Not clientHashTable.ContainsKey(IPAddr) Then

Return clientLookupTableErrors.CLIENT\_DOES\_NOT\_EXIST

End If

'Get the values provided by looking up the IP address in the hashtable

Dim values As tableValues = clientHashTable(IPAddr)

Return values

End Function

Public Function GetClientPublicKey(ByVal IPAddr As String) As Object

If Not clientHashTable.ContainsKey(IPAddr) Then

Return clientLookupTableErrors.CLIENT\_DOES\_NOT\_EXIST

End If

'return the public key

Dim values As tableValues = clientHashTable(IPAddr)

Return values.publicKey

End Function

Public Function GetClientUname(ByVal IPAddr As String) As Object

If Not clientHashTable.ContainsKey(IPAddr) Then

Return clientLookupTableErrors.CLIENT\_DOES\_NOT\_EXIST

End If

'returns the username of the client

Dim values As tableValues = clientHashTable(IPAddr)

Return values.username

End Function

Public Function GetClientReference(ByVal IPAddr As String) As Object

If Not clientHashTable.ContainsKey(IPAddr) Then

Return clientLookupTableErrors.CLIENT\_DOES\_NOT\_EXIST

End If

'returns the reference to the MTClientReciever

Dim values As tableValues = clientHashTable(IPAddr)

Return values.clientReference

End Function

Public Sub CloseAllClientConnections()

'loop through all entrieds and close the connections

For Each entries As DictionaryEntry In clientHashTable

Dim values As tableValues = entries.Value

values.clientReference.closeClientConnection(False)

Next

clientHashTable.Clear()

End Sub

Public Function GetConnectedClients() As tableValues()

'Get all of the clients that are connected

Dim loggenOnUsers(clientHashTable.Count - 1) As tableValues

Dim counter As Integer = 0

For Each entries As DictionaryEntry In clientHashTable

loggenOnUsers(0) = entries.Value

counter += 1

Next

Return loggenOnUsers

End Function

Public Function ResolveUsernameToIP(ByVal username As String) As String

'Loop through all of the entries until we find the value for username that equals the one provided

For Each entries As DictionaryEntry In clientHashTable

Dim values As tableValues = entries.Value

If values.username = username Then

'return the corresponding IP address when we find the username

Return entries.Key

End If

Next

'return nothing if nothing is found

Return Nothing

End Function

Public Sub sendDisconnectedClientBroadcast(ByVal disconnectedClientUsername() As Byte)

'| message type | username |

'This goes through all of the currently connected clients, gets their reference and sends the username of the client that disconnected

Const usernameLen As Integer = 20

Const value As Integer = 220 'DC in hex

Dim message(usernameLen) As Byte

message(0) = CByte(value)

disconnectedClientUsername.CopyTo(message, 1)

For Each entries As DictionaryEntry In clientHashTable

Dim values As tableValues = entries.Value

Dim ref As MTClientReciever = values.clientReference

ref.sendData(message)

Next

End Sub

Public Sub sendConnectedClientBroadcast(ByVal connectedClientUsername() As Byte, ByVal publicKey() As Byte)

'| message type | username | public key |

'This goes through all of the currently connected clients, gets their reference and sends the username of the client that connected

Const usernameLen As Integer = 20

Const value As Integer = 204 'CC in hex

For Each entries As DictionaryEntry In clientHashTable

Dim values As tableValues = entries.Value

Dim ref As MTClientReciever = values.clientReference

Dim message(usernameLen + publicKey.Length) As Byte

message(0) = CByte(value)

connectedClientUsername.CopyTo(message, 1)

publicKey.CopyTo(message, 1 + usernameLen)

ref.sendData(message)

Next

End Sub

End Class

I wrote a class to take care of all the functions that we needed with the lookup table, from searching for usernames, to resolving usernames to IP addresses etc. There are a few changes that could be done to this, some of the functions are declared to return objects because its possible for them to return an error from the enum, or a value from the hash table, instead it would have been better to return just the value from the table, if an error occurred the return value is set to nothing.

#### Interacting with the UI on the Main Thread

During my testing I had a lot of problems with accessing elements correctly on the main forms running inside of the main thread. According to some sources online, they suggested that the default instances of the forms were not adequately suited to deal with requests from across threads. The method to solving these used delegates and invoking commands in order to interact with elements on the UI. My first attempts at this required every function and every subroutine that started a new thread, pass the default instance of the form to it as **byref** via the parameters. This approach to the solution worked but was not the correct way of dealing with this problem. Instead the better approach was to create a public sub in in the forms class which dealt with invoking, on a thread by thread basis.

Private Delegate Sub writeLogDelegate(ByVal message As String)

Private Delegate Sub appendClient(ByVal Username As String, ByVal IPAddr As String, ByVal PublicKey() As Byte)

Private Delegate Sub removeClient(ByVal Row As Integer)

Public Sub writeLog(ByVal Message As String)

'Used for default form interactions from other threads, this uses the main thread to update the form for saftey reasons

If RTB\_Log.InvokeRequired = True Then

Dim Dgate As writeLogDelegate = New writeLogDelegate(AddressOf writeLog)

RTB\_Log.BeginInvoke(Dgate, New Object() {Message})

Else

RTB\_Log.AppendText(Message)

End If

End Sub

Public Sub addNewClient(ByVal Username As String, ByVal IPAddr As String, ByVal PublicKey() As Byte)

'Adds a new client to the data grid viewer in the User tab

If DGV\_Users.InvokeRequired Then

Dim Dgate As appendClient = New appendClient(AddressOf addNewClient)

DGV\_Users.BeginInvoke(Dgate, New Object() {Username, IPAddr, PublicKey})

Else

Dim key As Long = BitConverter.ToInt64(PublicKey, 0)

DGV\_Users.Rows.Add(New String() {Username, IPAddr, key})

End If

End Sub

Public Sub removeClientDGV(ByVal Row As Integer)

'Removes a client from the data grid viewer in the User tab

If DGV\_Users.InvokeRequired Then

Dim Dgate As removeClient = New removeClient(AddressOf removeClientDGV)

DGV\_Users.BeginInvoke(Dgate, New Object() {Row})

Else

Try

DGV\_Users.Rows.RemoveAt(Row)

Catch ex As Exception

'all is well client wasnt added in the first place

Return

End Try

End If

End Sub

The top features 3 global delegates allowing us to pass the subroutine as a parameter to a function we are calling. This is required when calling **[FORM ELEMENT].BeginInvoke**  which needs to know the exact subroutine to execute when control returns to the creation thread. **BeginInvoke** works by taking the subroutine that we need to execute and adds it to a queue on the creator thread so that the thread executes items in the queue successively. In our situation when we call any of the subroutines with code from another thread, **InvokeRequired** should be set to true and we create a delegate of the subroutine that we are currently executing, when the creator thread starts executing items from the queue, **InvokeRequired** should be false and therefore be able to change the UI elements perfectly fine. I created a shared subroutine that allows any thread access to the UI:

Public Class MTMainThreadWriter

Private Shared frmInstance As Form1 = Nothing

Public Enum MTMTWErrors

NO\_ERROR

FRMINSTANCE\_IS\_NULL

EXCEPTION\_HAPPENED

End Enum

Public Shared Sub setInstance(ByRef MainFormUIInstance As Form1)

'must be called first so that we have the instance of the UI

frmInstance = MainFormUIInstance

End Sub

Public Shared Function writeLog(ByVal message As String) As MTMTWErrors

'check that the instance has been set then try adding the message to the log

If frmInstance Is Nothing Then

Return MTMTWErrors.FRMINSTANCE\_IS\_NULL

End If

Try

frmInstance.writeLog(message)

Catch ex As Exception

Return MTMTWErrors.EXCEPTION\_HAPPENED

End Try

Return MTMTWErrors.NO\_ERROR

End Function

Public Shared Function AddClientToDGV(ByVal Username As String, ByVal IPAddr As String, ByVal PublicKey() As Byte) As MTMTWErrors

'check that the instance has been set then try adding the client to the DataGridViewer

If frmInstance Is Nothing Then

Return MTMTWErrors.FRMINSTANCE\_IS\_NULL

End If

Try

frmInstance.addNewClient(Username, IPAddr, PublicKey)

Catch ex As Exception

Return MTMTWErrors.EXCEPTION\_HAPPENED

End Try

Return MTMTWErrors.NO\_ERROR

End Function

Public Shared Function RemoveClientFromDGV(ByVal Row As Integer) As MTMTWErrors

'check that the instance has been set then try removing the client from the DataGridView

If frmInstance Is Nothing Then

Return MTMTWErrors.FRMINSTANCE\_IS\_NULL

End If

Try

frmInstance.removeClientDGV(Row)

Catch ex As Exception

Return MTMTWErrors.EXCEPTION\_HAPPENED

End Try

Return MTMTWErrors.NO\_ERROR

End Function

Public Shared Function GetDGVCount() As Object

'check that the instance has been set then try getting the number of rows in the DataGridView

If frmInstance Is Nothing Then

Return MTMTWErrors.FRMINSTANCE\_IS\_NULL

End If

Try

Return frmInstance.DGV\_Users.Rows.Count

Catch ex As Exception

End Try

Return MTMTWErrors.EXCEPTION\_HAPPENED

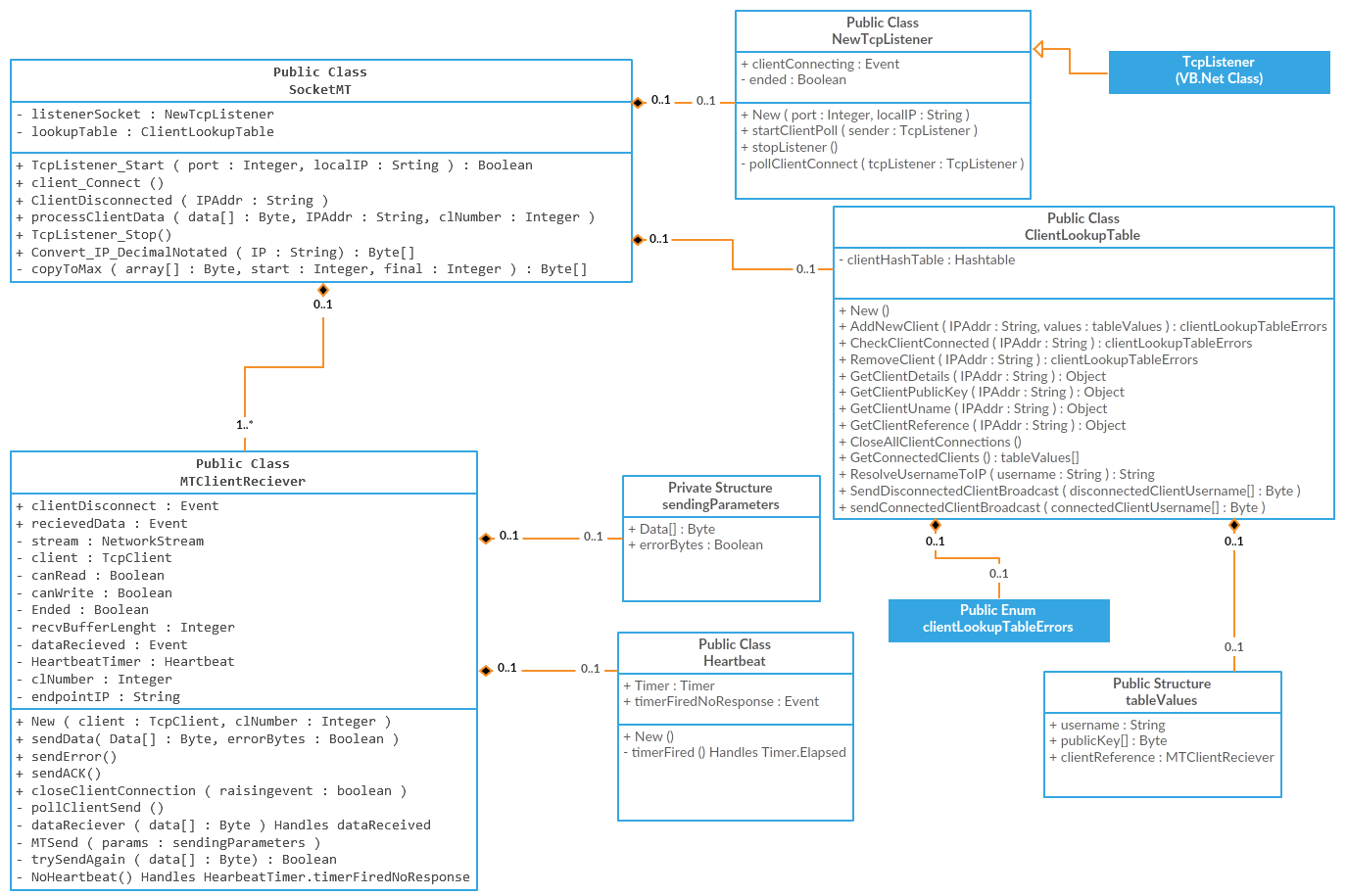
End Function

End Class

For this system to work, **setInstance** must be called by code currently executing on the main thread, this allows correct access to the instantiated **form1** class, if this prerequisite was not met the program would throw an error, exclaiming that the instance of the form has not been set and alas no change would be made to the main UI. Once the instance is set, it is possible for a thread to call any of the shared functions inside the class wrapper. With the **GetDGVCount** function, it is unnecessary to call any invoking functions because no alterations are being made to the UI. As no modifications are being made, a race condition is not possible and in theory should be safe to access it this way.

#### SocketMT Class

The **socketMT** class was initially built to encapsulate all of the server TCP code and acts as a front end for ease of interaction with the internals of the connection between the server and client. A class diagram for the TCP side of the server looks something like:



This is the complete class diagram for how TCP transmission is structured inside of my server project. The top right in the blue box indicates that the **TcpListener** class is inherited by my subclass **NewTcpListener**, which is explained on page 31. The sections below from this page to page 103, covers the details of the class **socketMT** shown in the class diagram above.

##### Forwarding from client to client

I determined this in my head to be a pretty simple task which follows the routine of:

1. Open a TCP port for client connection
2. Allow a client to connect
3. Create client specific threads for sending and receiving
4. Forward the data and change the end user’s username to the sender’s
5. Close the connection when a client finally disconnects

This task became more and more challenging the deeper I delved into the design of it. After having designed the initial structure of the data that was being transferred to and from machines it looked something like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Message Identifier | Packet number | Maximum packet number | Username Length | Username | Data |
| Byte | Byte | Byte | Byte | Bytes | Bytes |

Behind the scenes the server decoded the message like so:

Private Sub processClientData(ByVal data() As Byte, ByVal IPAddr As String, ByVal clNumber As Integer)

Dim dataValue As Integer

Try

'try converting the data to an integer value, but possible that it overflows, so catch this

dataValue = BitConverter.ToInt32(data, 0)

Catch ex As Exception

'value overflowed 32bits, is okay because then we know its not hearbeat or a dead signal

End Try

If dataValue = 57005 Then

'Check for client disconnect value (0xDEAD) and remove any instances of that client

Dim client As MTClientReciever = lookupTable.GetClientReference(IPAddr)

client.sendACK()

client.closeClientConnection(True)

Else

Dim client As MTClientReciever = lookupTable.GetClientReference(IPAddr)

Dim messageType As Byte = data(0)

Dim lenghtOfUsefulData As Integer = data.Length

Dim usernameLength As Integer = data(3)

Dim senderUsername(ASCII.GetByteCount(lookupTable.GetClientUname(IPAddr)) - 1) As Integer

Const offset As Integer = 4

'Get the senders username from our lookup table

Array.Copy(ASCII.GetBytes(lookupTable.GetClientUname(IPAddr)), senderUsername, ASCII.GetByteCount(lookupTable.GetClientUname(IPAddr)))

'We need to reprocess the data for the end user, which means changing the username from the recipients to the senders and therefore altering the username length

For i = data.Length - 1 To 0 Step -1

If data(i) <> 0 Then

Exit For

End If

If lenghtOfUsefulData = usernameLength + 3 Then

Exit For

End If

lenghtOfUsefulData -= 1

Next

Dim x As Byte = data(1)

Dim y As Byte = data(2)

'Get the bytes for the username determined by the value of data(3)

Dim username(usernameLength - 1) As Byte

For i = 0 To usernameLength - 1

username(i) = data(i + offset)

Next

Dim encryptedData(lenghtOfUsefulData - offset - usernameLength - 1) As Byte

encryptedData = copyToMax(data, offset + usernameLength, lenghtOfUsefulData - 1)

'We need to get the recipients IP address from our lookup table so we can get the reference of the instance of MTClientReciever for that client

Dim recpIPAddr As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(username))

If recpIPAddr = Nothing Then

'Got to now store the message for later, that user is not currently logged on

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recpIPAddr & vbNewLine)

Return

End If

'Here we get that clients reference

Dim recpientReference As MTClientReciever = lookupTable.GetClientReference(recpIPAddr)

Dim newData(lenghtOfUsefulData) As Byte

newData(0) = messageType

newData(1) = x

newData(2) = y

newData(3) = senderUsername.Length

senderUsername.CopyTo(newData, offset)

encryptedData.CopyTo(newData, offset + usernameLength + hashLen)

recpientReference.sendData(encryptedData)

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Forwarding data " & IPAddr & " >> " & recpIPAddr & vbNewLine)

End If

End Sub

Private Function copyToMax(ByVal array() As Byte, ByVal start As Integer, ByVal final As Integer) As Byte()

Dim finalisedArray(final - start) As Byte

For i = start To final

finalisedArray(i - start) = array(i)

Next

Return finalisedArray

End Function

The way this works is that firstly it tries to convert the value of the received data to decimal, this is because the data could be a disconnect message from the client indicating that the client has deceased, by comparing the converted value to the decimal 57005, which stands for DEAD in hexadecimal we know if the client has stopped and therefore we can remove the client from the lookup table and issue the shutdown to the class that dealt with the client’s requests. After having checked for that value we can assume it will probably be a message from the client either containing the encrypted IM or the encrypted file. We can setup the variables with some known values that either from the data, or know array boundaries, and create the constants to make our offsets manageable. After having proceeded with that section, the data we received is run through from the end and the length of the data that is useful later is stored. Next, the username of the recipient is gathered from the sender, its size indicated by the byte value at the location 3 in the array. Lastly, we then gather the rest of the message into a buffer which should be the encrypted data the user has sent. Now that everything useful has been obtained, the username of the recipient can be used to get the reference to the class that is dealing with that users requests, and the new data can be reconstructed and sent.

This initial design of the message was fairly simple and it worked, except later I realised that this design was impractical. Given the situation if the receiver’s username was longer than the sender’s username then the username section would overflow into the data section, either cutting off some of the data, making the packets invalid upon receiving, or would throw an error because the array was not long enough to send all of the data. The new method to fix this was to set the maximum length of the username to 20 bytes, I determined that this should be long enough to store most usernames and it could easily be changed in the code as the offsets for the positions of each section of data was coded using constants rather than hardcoded in. With this change we no longer needed to add Username Length as we would always copy across 20 bytes worth of data, therefore the length was never variable. The new design of the data in the packet looked something like:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Message Identifier | Packet number | Maximum packet number | Username | Data |
| Byte | Byte | Byte | Bytes (max 20) | Bytes |

This required an adaption to the server code like so:

Dim username(usernameLength - 1) As Byte

For i = 0 To usernameLength - 1

username(i) = data(i + offset)

Next

Dim lastNonZeroIndex As Integer = 19

'find the last index in the username array that has a byte that is != 0

For i = 19 To 0 Step -1

If username(i) <> 0 Then

Exit For

End If

lastNonZeroIndex -= 1

Next

Dim nonZeroUsername(lastNonZeroIndex) As Byte

'Cut off the zero bytes by copying the userful data into a buffer

Array.Copy(username, nonZeroUsername, lastNonZeroIndex + 1)

This replaces the original method for retrieving the data, and the value of offset can be updated to equal 3 and the value of usernameLength can be set to a constant of 20.

Through some short tests that I did I encountered some unexplainable problems with sending the data. Sometimes when reading the data from the **NetworkStream**, not all of the bytes that I had originally sent were present. Some research online presented a problem that existed with the theory of sending data over the network, sometimes it was possible for not all of the data to be in the buffer after using **NetworkStream.Read**. In order to resolve this problem, I thought about implementing a system of using a hash as a checksum for the data that I was being sent. This lead me on to once again re design the message structure again like so:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Message Identifier | Packet number | Maximum packet number | Username | Data Hash | Data |
| Byte | Byte | Byte | Bytes (max 20) | Bytes (max 32) | Bytes |

It was possible to set the sizing up as a constant for everything except the data section. I determined the size of the data by taking the maximum packet size determined by **TCPClient.ReceiveBufferSize** and taking away an offset of 3 for the first 3 bytes plus the offset of the Username section, of 20, and the offset of the hash section as 32.

My new design presented some new errors, mainly due to determining the size of the data in the buffer and generating a hash for that data. When I received the packets on the server side I looped through the array for the last value down to zero and kept a count of the last index where the data was not equal to zero, in theory this works if when all of the data is something not equal to zero, but when sending data that had a few null values that ended at final indexes of the messages these values were removed and the generated hash for the data was not equal to the hash that was sent, causing a constant loop of errors. In order to compensate for this problem, I only run through the buffer like above when the x value is equal to the y value as the rest of the packets should be full of correct, non-erroneous data, so long as the hashes match. This meant though that before sending the final packet the client had to remove any trailing zeros from the data, this meant that when the same procedure was used by the server, the hash would be the same.

In order for the hash checking to be useful I had to adapt both the server and the clients to send and accept a pair of bytes, determining whether the hash matched or if the client should send the data again.

There was still a problem with this system that I had realised that I had missed out when sending files over the network. Using the same data layout for the messages and the files meant that the user never knew the filename, my initial resolution was to give a default name to all of the files received by a client, but later realised that I didn’t know the file extension of the original copy making my new design useless. I decided to adapt my idea by sending a packet, with an x value of 0, before I sent the actual data of the file. My choice to do this mean that I didn’t have to rewrite all of the server code that decoded the sent data and all it required was a small addition to the code that checked if the x value was equal to 0 when the Message Identifier was set to the decimal value of 2 and decoded the data in almost the same way as before, minus the data hash and the file data. An example of what I meant can be seen below:

Dim x As Byte = data(1)

Dim y As Byte = data(2)

'Dim usernameLength As Byte = data(3)

'retrieve the full 20 bytes of the username from the message

Dim username(usernameLength - 1) As Byte

For i = 0 To usernameLength - 1

username(i) = data(i + offset)

Next

Dim lastNonZeroIndex As Integer = 19

'find the last index in the username array that has a byte that is != 0

For i = 19 To 0 Step -1

If username(i) <> 0 Then

Exit For

End If

lastNonZeroIndex -= 1

Next

Dim nonZeroUsername(lastNonZeroIndex) As Byte

'Cut off the zero bytes by copying the userful data into a buffer

Array.Copy(username, nonZeroUsername, lastNonZeroIndex + 1)

If x = 0 Then

Const filenameLen As Integer = 40

Dim initalMessage(offset + usernameLength + filenameLen - 1) As Byte

Dim fileName(filenameLen - 1) As Byte

For i = 0 To filenameLen - 1

fileName(i) = data(i + offset + usernameLength)

Next

Dim recp As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(nonZeroUsername))

If recp = Nothing Then

'Got to now store the message for later, that user is not currently logged on

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recp & vbNewLine)

Return

End If

client.sendACK()

'Here we get that clients reference

Dim reference As MTClientReciever = lookupTable.GetClientReference(recp)

initalMessage(0) = messageType

initalMessage(1) = x

initalMessage(2) = y

senderUsername.CopyTo(initalMessage, offset)

fileName.CopyTo(initalMessage, offset + usernameLength)

reference.sendData(initalMessage)

Return

End If

The layout of the initial message for the filename is structure like:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Message Identifier | Packet number | Maximum packet number | Username | Filename |
| Byte | Byte | Byte | Bytes (max 20) | Bytes |

Initially I set the maximum length of the filename to be 40 bytes as indicated by the constant **filenameLen**. I extended the length of this to a maximum of 100 bytes in order to incorporate any number of different files, as I determined the maximum length to be too restrictive.

Const filenameLen As Integer = 100

The way we decode the data we are given follows a simple process of retrieving bits from a known offset up to a maximum. Firstly, we set the variables x and y with the information from the corresponding position in the data we received. Next, we loop through the array and fill a temporary array with the 20 bytes of the username, running down form end to start we stop once we reach a byte that is not equal to the value of zero. We can then use that position to copy the temporary array into a finalised array the correct length, and thus remove the extra zeros from the end of the array. Next section is where we take the filename of the file about to be sent in binary, remove any trailing zeros, we then swap the recipients username out for the senders so that the recipient is aware of who sent the data and finally we reconstruct the new message and forward the data onto the client. After having completed this we can exit this subroutine as the rest of the code is specific to decoding the IMs and files as mentioned above.

In retrospect the code could have been condensed a lot more, I could have used the existing function **Array.ConstrainedCopy** which allows the specification of the starting index for each array and the length like any copy. This would have then left me to process the data, removing zeros, etc. like before.

The final design of this incorporating all of the changes above can be represented below:

Private Sub processClientData(ByVal data() As Byte, ByVal IPAddr As String, ByVal clNumber As Integer)

Dim dataValue As Integer

Try

'try converting the data to an integer value, but possible that it overflows, so catch this

dataValue = BitConverter.ToInt32(data, 0)

Catch ex As Exception

'value overflowed 32bits, is okay because then we know its not hearbeat or a dead signal

End Try

If dataValue = 57005 Then

'Check for client disconnect value (0xDEAD) and remove any instances of that client

Dim client As MTClientReciever = lookupTable.GetClientReference(IPAddr)

client.sendACK()

client.closeClientConnection(True)

Else

Dim client As MTClientReciever = lookupTable.GetClientReference(IPAddr)

Dim messageType As Byte = data(0)

Dim lenghtOfUsefulData As Integer = data.Length

Dim senderUsername(19) As Byte

Const usernameLength As Integer = 20

Const offset As Integer = 3

Const hashLen As Integer = 32

Dim sha256 As New SHA256Managed

Array.Copy(ASCII.GetBytes(lookupTable.GetClientUname(IPAddr)), senderUsername, ASCII.GetByteCount(lookupTable.GetClientUname(IPAddr)))

'We need to reprocess the data for the end user, which means changing the username from the recipients to the senders and therefore altering the username length

'| Message Type | x | of y | username | data hash | data |

'| Byte | B | B | 20 bytes | 32 bytes | Bytes|

If data(0) <> 1 And data(0) <> 2 Then

client.sendError()

Return

End If

If data(1) = data(2) Then

For i = data.Length - 1 To 0 Step -1

If data(i) <> 0 Then

Exit For

End If

If lenghtOfUsefulData = usernameLength + 3 Then

Exit For

End If

lenghtOfUsefulData -= 1

Next

End If

Dim x As Byte = data(1)

Dim y As Byte = data(2)

'retrieve the full 20 bytes of the username from the message

Dim username(usernameLength - 1) As Byte

For i = 0 To usernameLength - 1

username(i) = data(i + offset)

Next

Dim lastNonZeroIndex As Integer = 19

'find the last index in the username array that has a byte that is != 0

For i = 19 To 0 Step -1

If username(i) <> 0 Then

Exit For

End If

lastNonZeroIndex -= 1

Next

Dim nonZeroUsername(lastNonZeroIndex) As Byte

'Cut off the zero bytes by copying the userful data into a buffer

Array.Copy(username, nonZeroUsername, lastNonZeroIndex + 1)

If x = 0 Then

Const filenameLen As Integer = 100

Dim initalMessage(offset + usernameLength + filenameLen - 1) As Byte

Dim fileName(filenameLen - 1) As Byte

For i = 0 To filenameLen - 1

fileName(i) = data(i + offset + usernameLength)

Next

Dim recp As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(nonZeroUsername))

If recp = Nothing Then

'Got to now store the message for later, that user is not currently logged on

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recp & vbNewLine)

Return

End If

client.sendACK()

'Here we get that clients reference

Dim reference As MTClientReciever = lookupTable.GetClientReference(recp)

initalMessage(0) = messageType

initalMessage(1) = x

initalMessage(2) = y

senderUsername.CopyTo(initalMessage, offset)

fileName.CopyTo(initalMessage, offset + usernameLength)

reference.sendData(initalMessage)

Return

End If

Dim hashBytes(hashLen - 1) As Byte

For i = 0 To hashLen - 1

hashBytes(i) = data(i + offset + usernameLength)

Next

Dim encryptedData(lenghtOfUsefulData - offset - usernameLength - hashLen - 1) As Byte

encryptedData = copyToMax(data, offset + usernameLength + hashLen, lenghtOfUsefulData - 1)

Dim genHashBytes() As Byte = sha256.ComputeHash(encryptedData)

If genHashBytes.Length > hashLen Then

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "[ERROR] Generated hash for the received data longer than max allowed" & vbNewLine)

'Not much we can do to deal with this, its a programming error

client.sendACK()

Return

End If

'Compare the two hashes of the encrypted data, in order to make sure there were no errors when transmitting the file

For i = 0 To hashLen - 1

If hashBytes(i) <> genHashBytes(i) Then

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "[ERROR] Generated hash did not match the hash received from the client" & vbNewLine)

client.sendError()

Return

End If

Next

'The hashes were equal so we may now send the ACK so the client sends the next set of data

client.sendACK()

'We need to get the recipients IP address from our lookup table so we can get the reference of the instance of MTClientReciever for that client

Dim recpIPAddr As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(nonZeroUsername))

If recpIPAddr = Nothing Then

'Got to now store the message for later, that user is not currently logged on

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recpIPAddr & vbNewLine)

Return

End If

'Here we get that clients reference

Dim recpientReference As MTClientReciever = lookupTable.GetClientReference(recpIPAddr)

Dim newData(lenghtOfUsefulData) As Byte

newData(0) = messageType

newData(1) = x

newData(2) = y

senderUsername.CopyTo(newData, offset)

hashBytes.CopyTo(newData, offset + usernameLength)

encryptedData.CopyTo(newData, offset + usernameLength + hashLen)

recpientReference.sendData(encryptedData)

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Forwarding data " & IPAddr & " >> " & recpIPAddr & vbNewLine)

End If

End Sub

This features all of the necessary changes explained above, especially in the middle:

Dim encryptedData(lenghtOfUsefulData - offset - usernameLength - hashLen - 1) As Byte

encryptedData = copyToMax(data, offset + usernameLength + hashLen, lenghtOfUsefulData - 1)

Dim genHashBytes() As Byte = sha256.ComputeHash(encryptedData)

If genHashBytes.Length > hashLen Then

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "[ERROR] Generated hash for the received data longer than max allowed" & vbNewLine)

'Not much we can do to deal with this, its a programming error

client.sendACK()

Return

End If

'Compare the two hashes of the encrypted data, in order to make sure there were no errors when transmitting the file

For i = 0 To hashLen - 1

If hashBytes(i) <> genHashBytes(i) Then

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "[ERROR] Generated hash did not match the hash Received from the client" & vbNewLine)

client.sendError()

Return

End If

Next

'The hashes were equal so we may now send the ACK so the client sends the next set of data

client.sendACK()

This is the section that retrieves the hash from the received data and generates a new hash for the data that we currently have, if the hashes are not equal then we get the class that deals with the client to send the client an error and if the hashes match the client is sent a no error message and is allowed to proceed with sending the rest of the data if necessary.

##### Processing a new client connecting

Having thought about the requirements of the server detailed 3 pieces of data that need to be transmitted upon connecting to the server. This data must be the first to be exchanged so that the user is able to successfully communicate with other clients after connecting. The 3 forms of transmissions are:

1. The client provides the users username and public key
2. The server sends the currently connected users to and from the server
3. Other connected users are notified of a newly connected user

These three things make it possible for transmission to continue between users, this is because each user needs to know the other connected users, and for when transmission starts the user also needs to obtain the public key of the user they are attempting to talk to, in order to create the encrypted communication channel between them.

Each one of these transmissions has a specific packet data design to accompany it, so that the receiver knows the type of message and the data that is being sent:

|  |  |  |  |
| --- | --- | --- | --- |
| Username length | Username | Public Key Length | Public Key |
| Byte | Bytes | Byte | Bytes |

The second bullet point is a slightly longer approach to sending the data, firstly we sent the count of packets to the user so the user knows how many to expect to receive, then we send the connected usernames followed by the respective public key to the user and we iterate through until all of the users have been sent.

The third is a simpler approach, we use our **clientLookupTable** and we iterate through all of the users listed in the table and send them a packet like so:

|  |  |  |
| --- | --- | --- |
| Message Identifier | Username | Public Key |
| Byte (value of 204 (CC in hex)) | Bytes | Bytes |

Below is the process of sending and receiving the data:

Private Sub client\_Connect()

'Design of the intitial connection message:

'start | UNAMELENGHT | UNAME | PKLENGHT | PK | end

' | BYTE | BYTES | BYTE |BYTES|

Dim username As String

Dim IPaddr As String

Dim endpoint As IPEndPoint

Dim netStream As NetworkStream

Dim unameLength As Integer

Dim pkLength As Integer

MTMainThreadWriter.writeLog("New user awaiting acept" & vbNewLine)

'If there isnt an instance of the lookup table create one

If lookupTable Is Nothing Then

lookupTable = New ClientLookupTable

End If

'Accept the clients connection

Dim client As TcpClient = listenerSocket.AcceptTcpClient()

'Get the remote end point and then the IP address of the client

endpoint = client.Client.RemoteEndPoint

IPaddr = endpoint.Address.ToString

If lookupTable.CheckClientConnected(IPaddr) = ClientLookupTable.clientLookupTableErrors.CLIENT\_CONNECTED Then

MTMainThreadWriter.writeLog("Client already connected" & vbNewLine)

client.Close()

Return

End If

Dim recvBytes(client.ReceiveBufferSize) As Byte

'sync the networkstream with the client stream

netStream = client.GetStream

'Read the data from the stream in order to get the username, public key from the client

netStream.Read(recvBytes, 0, client.ReceiveBufferSize)

'#Region "All of this is incorrect and needs replacing when the logon system is implemented"

'we need to get username length

unameLength = CInt(recvBytes(0))

'loop through the values from point 1 in the Received data array to the end of username, we dont need to minus 1 because we have offset the data by 1

For i = 1 To unameLength

username = username & System.Convert.ToChar(recvBytes(i))

Next

'Now get the public key lenght that is right after the username as shown at the start

pkLength = CInt(recvBytes(unameLength + 1))

Dim publicKey(pkLength - 1) As Byte

'Now just loop through as you normally would and offest the recvBytes array by usernamelength and a value of 2 because of the 2 bytes used for data lengths

For i = 0 To pkLength - 1

publicKey(i) = recvBytes(i + unameLength + 2)

Next

'Check if the details are null

If publicKey Is Nothing Or username = Nothing Then

MTMainThreadWriter.writeLog("Client connection details are NULL" & vbNewLine)

Return

End If

'#End Region

'Get all of the currently connected clients

Dim connectedClientsValues As ClientLookupTable.tableValues() = lookupTable.GetConnectedClients()

Dim numberOfClientsBytes() As Byte = BitConverter.GetBytes(connectedClientsValues.Length)

'send the count of packets that we will send, and transmit that first

netStream.Write(numberOfClientsBytes, 0, numberOfClientsBytes.Length)

netStream.Flush()

Const usernameLen As Integer = 20

'send:

'| username | public key |

For i = 0 To connectedClientsValues.Length - 1

Dim message(usernameLen + connectedClientsValues(i).publicKey.Length) As Byte

Dim usernameBytes() As Byte = ASCII.GetBytes(connectedClientsValues(i).username)

usernameBytes.CopyTo(message, 0)

connectedClientsValues(i).publicKey.CopyTo(message, usernameLen)

netStream.Write(message, 0, message.Length)

netStream.Flush()

Next

'Get the unique client number based off of the number of clients in the DataGridView element

Dim clientCount As Integer = MTMainThreadWriter.GetDGVCount

'Start the instance of the multithreaded client sender and reciever

Dim clientHandler As New MTClientReciever(client, clientCount)

'Add 2 handlers, 1 to deal with removing the client from lookupTable and one to deal with the data once Received

AddHandler clientHandler.clientDisconnect, AddressOf clientDisconnected

AddHandler clientHandler.ReceivedData, AddressOf processClientData

Dim lookupTableValues As New ClientLookupTable.tableValues

lookupTableValues.clientReference = clientHandler

lookupTableValues.publicKey = publicKey

lookupTableValues.username = username

lookupTable.sendConnectedClientBroadcast(ASCII.GetBytes(username), publicKey)

'try and populate the table with the values above and the IP address as the lookup key

If lookupTable.AddNewClient(IPaddr, lookupTableValues) = ClientLookupTable.clientLookupTableErrors.TABLE\_CONTAINS\_CLIENT Then

MTMainThreadWriter.writeLog("Client already connected" & vbNewLine)

clientHandler.closeClientConnection(False)

Return

End If

MTMainThreadWriter.AddClientToDGV(username, IPaddr, publicKey)

MTMainThreadWriter.writeLog("[Client: " & clientCount & "]: " & "Created new client with connection number: " & clientCount & vbNewLine)

End Sub

Here we start off by checking if **lookupTable** has been instantiated, if not then we create an instance of that ready for later. Next, we get the IP address of the client so that we can check if the client is already connected and therefore reject the new connection. Once the client is accepted we read the clients data from the network stream in order to get the transmitted username and public key, in future revisions the hash of the user’s password would need to be sent too, in order to authenticate the user with a list of users in the database, either rejecting or accepting the connection. After having received the client’s information, we enumerate all of the currently logged in users and send a packet for each user, containing the username and public, mentioned above. Finally, assuming that all has gone well so far, a new event handler is created for the two events that are raised by the **MTClientReceiver** class, the currently connected users are sent a broadcast of the username and IP of the new operator and that person is then added to the lookup table for later use.

##### Starting and stopping the TCP server

There are two main functions inside the SocketMT class that creates and halts all of the needed objects for the TCP server to function. The subroutine that started the server was initially create as a **New()** procedure, but this process disallows the use of return values and therefore the instantiating code is unable to tell whether the server functions were initiated successfully. Initially it took the form of:

Public Sub New(ByVal port As Integer, ByVal localIP As IPAddress)

If port = Nothing Or port = 0 Or localIP.Equals(Nothing) Then

Form1.RTB\_Log.AppendText("Failed to start socket, 1 or more parameters were invalid" & vbNewLine)

Return

End If

listenerSocket = New TcpListener(localIP, port)

Try

listenerSocket.Start()

Catch ex As Exception

Form1.RTB\_Log.AppendText(ex.Message)

End Try

Form1.RTB\_Log.AppendText("Created TCP listener on: " & localIP.ToString & ":" & port & vbNewLine)

End Sub

But later this was merged with the function **TcpListener\_Start()** in order to return a Boolean value indicating success or failure.

Public Function TcpListener\_Start(ByVal port As Integer, ByVal LocalIP As IPAddress) As Boolean

'Check if the port is invalid or if the IP address is invalid

If port = Nothing Or port = 0 Or LocalIP.Equals(Nothing) Then

Form1.RTB\_Log.AppendText("Failed to start socket, 1 or more parameters were invalid" & vbNewLine)

Return False

End If

If port > 65535 Then

Form1.RTB\_Log.AppendText("[ERROR]: Port was larger than the maximum allowed" & vbNewLine)

Return False

End If

'Set the instance of the UI updater

MTMainThreadWriter.setInstance(Form1)

'Create a new instance of NewTcpListener

listenerSocket = New NewTcpListener(port, LocalIP)

Try

'try creating the socket and catch any errors thrown

listenerSocket.Start()

Catch ex As Exception

Form1.RTB\_Log.AppendText(ex.Message & vbNewLine)

Return False

End Try

'Now start polling for a client using our new multithreaded version of TcpClient

listenerSocket.startClientPoll(listenerSocket)

Form1.RTB\_Log.AppendText("Created TCP listener on: " & LocalIP.ToString & ":" & port & vbNewLine)

'listenerSocket.Stop()

'Add a new handler that handles when the pollClientConnetion fires and a new user wants to connect

AddHandler listenerSocket.clientConnecting, AddressOf Me.client\_Connect

Return True

End Function

This section starts off with some rudimental error checking, alleviating any form of crashes caused by either the port or IP address being a null value. Having passed the checks, an instance of the main form is set before we change threads and the TCP listener is started, any errors are caught and the message is written out to the log on the main form. Finally, the polling thread is created, waiting for a new user to connect to the server, and adding a handler for the event the poll raises allows this class to deal with that connecting user.

Stopping the server is simpler than starting it:

Public Sub TcpListener\_Stop()

'Kill off everything to stop the server running

If listenerSocket Is Nothing Then

Return

End If

listenerSocket.stopListener()

If lookupTable Is Nothing Then

Return

End If

lookupTable.CloseAllClientConnections()

End Sub

We check to see if the socket is running, and stop if it’s not, we then check to see if the lookup table has any entries and if it does we then call the function that loops through those entries and removes them from the hashtable.

Forwarding the data had to also be updated when the design was changed to ensure that all the data was receive (explained on page 70), this made all of the arrays that formed the data to send it back to the client a fixed length of **recvBufferSize**:

Private Sub processClientData(ByVal data() As Byte, ByVal IPAddr As String, ByVal clNumber As Integer)

Dim dataValue As Integer

Try

'try converting the data to an integer value, but possible that it overflows, so catch this

dataValue = BitConverter.ToInt32(data, 0)

Catch ex As Exception

'value overflowed 32bits, is okay because then we know its not hearbeat or a dead signal

End Try

If dataValue = 57005 Then

'Check for client disconnect value (0xDEAD) and remove any instances of that client

Dim client As MTClientReciever = lookupTable.GetClientReference(IPAddr)

client.closeClientConnection(True)

Else

Dim client As MTClientReciever = lookupTable.GetClientReference(IPAddr)

Dim messageType As Byte = data(0)

Dim lenghtOfUsefulData As Integer = data.Length

Dim senderUsername(19) As Byte

Const usernameLength As Integer = 20

Const offset As Integer = 3

Const hashLen As Integer = 32

Dim sha256 As New SHA256Managed

Array.Copy(ASCII.GetBytes(lookupTable.GetClientUname(IPAddr)), senderUsername, ASCII.GetByteCount(lookupTable.GetClientUname(IPAddr)))

'We need to reprocess the data for the end user, which means changing the username from the recipients to the senders and therefore altering the username length

'| Message Type | x | of y | username | data hash | data |

'| Byte | B | B | 20 bytes | 32 bytes | Bytes|

If data(0) <> 1 And data(0) <> 2 Then

client.sendError()

Return

End If

If data(1) = data(2) Then

For i = data.Length - 1 To 0 Step -1

If data(i) <> 0 Then

Exit For

End If

If lenghtOfUsefulData = usernameLength + 3 Then

Exit For

End If

lenghtOfUsefulData -= 1

Next

End If

Dim x As Byte = data(1)

Dim y As Byte = data(2)

'Dim usernameLength As Byte = data(3)

'retrieve the full 20 bytes of the username from the message

Dim username(usernameLength - 1) As Byte

For i = 0 To usernameLength - 1

username(i) = data(i + offset)

Next

Dim lastNonZeroIndex As Integer = 19

'find the last index in the username array that has a byte that is != 0

For i = 19 To 0 Step -1

If username(i) <> 0 Then

Exit For

End If

lastNonZeroIndex -= 1

Next

Dim nonZeroUsername(lastNonZeroIndex) As Byte

'Cut off the zero bytes by copying the userful data into a buffer

Array.Copy(username, nonZeroUsername, lastNonZeroIndex + 1)

If x = 0 Then

Const filenameLen As Integer = 100

Dim initalMessage(data.Length - 1) As Byte

Dim fileName(filenameLen - 1) As Byte

For i = 0 To filenameLen - 1

fileName(i) = data(i + offset + usernameLength)

Next

Dim recp As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(nonZeroUsername))

If recp = Nothing Then

'Got to now store the message for later, that user is not currently logged on

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recp & vbNewLine)

Return

End If

client.sendACK()

'Here we get that clients reference

Dim reference As MTClientReciever = lookupTable.GetClientReference(recp)

initalMessage(0) = messageType

initalMessage(1) = x

initalMessage(2) = y

senderUsername.CopyTo(initalMessage, offset)

fileName.CopyTo(initalMessage, offset + usernameLength)

reference.sendData(initalMessage)

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Forwarding data " & IPAddr & " >> " & recp & vbNewLine)

Return

End If

'printArray(data, 10)

Dim hashBytes(hashLen - 1) As Byte

For i = 0 To hashLen - 1

hashBytes(i) = data(i + offset + usernameLength)

Next

Dim encryptedData(lenghtOfUsefulData - offset - usernameLength - hashLen - 1) As Byte

'encryptedData = copyToMax(data, offset + usernameLength + hashLen, lenghtOfUsefulData - 1)

Array.ConstrainedCopy(data, offset + usernameLength + hashLen, encryptedData, 0, lenghtOfUsefulData - offset - usernameLength - hashLen)

Dim fs As New IO.FileStream(Environment.CurrentDirectory + "\rnd.txt", IO.FileMode.Create)

fs.Write(encryptedData, 0, encryptedData.Length)

fs.Close()

Dim genHashBytes() As Byte = sha256.ComputeHash(encryptedData)

If genHashBytes.Length > hashLen Then

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "[ERROR] Generated hash for the Received data longer than max allowed" & vbNewLine)

'Not much we can do to deal with this, its a programming error

client.sendACK()

Return

End If

'Compare the two hashes of the encrypted data, in order to make sure there were no errors when transmitting the file

For i = 0 To hashLen - 1

If hashBytes(i) <> genHashBytes(i) Then

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "[ERROR] Generated hash did not match the hash Received from the client" & vbNewLine)

client.sendError()

Debug.Print("hash")

Return

End If

Next

'The hashes were equal so we may now send the ACK so the client sends the next set of data

client.sendACK()

'We need to get the recipients IP address from our lookup table so we can get the reference of the instance of MTClientReciever for that client

Dim recpIPAddr As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(nonZeroUsername))

'Dim s As New IO.FileStream(Environment.CurrentDirectory & "/" & x & ".txt", IO.FileMode.Create)

's.Write(encryptedData, 0, encryptedData.Length)

's.Close()

If recpIPAddr = Nothing Then

'Got to now store the message for later, that user is not currently logged on

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recpIPAddr & vbNewLine)

Return

End If

'Here we get that clients reference

Dim recpientReference As MTClientReciever = lookupTable.GetClientReference(recpIPAddr)

'Send a full frame of bytes, therefore we know the max amount to recieve upon intercept, and therefore no more TCP problems, this also makes hashing now redundant, but keeping it as a backup

Dim newData(data.Length - 1) As Byte

newData(0) = messageType

newData(1) = x

newData(2) = y

'newData(3) = senderUsername.Length

senderUsername.CopyTo(newData, offset)

hashBytes.CopyTo(newData, offset + usernameLength)

encryptedData.CopyTo(newData, offset + usernameLength + hashLen)

recpientReference.sendData(newData)

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Forwarding data " & IPAddr & " >> " & recpIPAddr & vbNewLine)

End If

End Sub

Highlighted in a darker colour shows the updated lines, these make the length of the sending array equal to **recvBufferSize**, therefore if the data received does not fill the buffer entirely then the array is already padded with zeros.

With this final update I decided to leave hashing in the data design, acting as a backup to check that the data sent was correctly.

Overall **socketMT** is structured like so:

Imports System.Net.Sockets

Imports System.Net

Imports System.Text.Encoding

Imports System.Security.Cryptography

Public Class SocketMT

    Private WithEvents listenerSocket As NewTcpListener

    Private lookupTable As ClientLookupTable = Nothing

    'Public Sub New(ByVal port As Integer, ByVal localIP As IPAddress)

    '    If port = Nothing Or port = 0 Or localIP.Equals(Nothing) Then

    '        Form1.RTB\_Log.AppendText("Failed to start socket, 1 or more parameters were invalid" & vbNewLine)

    '        Return

    '    End If

    '    listenerSocket = New TcpListener(localIP, port)

    '    Try

    '        listenerSocket.Start()

    '    Catch ex As Exception

    '        Form1.RTB\_Log.AppendText(ex.Message)

    '    End Try

    '    Form1.RTB\_Log.AppendText("Created TCP listener on: " & localIP.ToString & ":" & port & vbNewLine)

    'End Sub

    Public Function TcpListener\_Start(ByVal port As Integer, ByVal LocalIP As IPAddress) As Boolean

        'Check if the port is invalid or if the IP address is invalid

        If port = Nothing Or port = 0 Or LocalIP.Equals(Nothing) Then

            Form1.RTB\_Log.AppendText("Failed to start socket, 1 or more parameters were invalid" & vbNewLine)

            Return False

        End If

        If port > 65535 Then

            Form1.RTB\_Log.AppendText("[ERROR]: Port was larger than the maximum allowed" & vbNewLine)

            Return False

        End If

        'Set the instance of the UI updater

        MTMainThreadWriter.setInstance(Form1)

        'Create a new instance of NewTcpListener

        listenerSocket = New NewTcpListener(port, LocalIP)

        Try

            'try creating the socket and catch any errors thrown

            listenerSocket.Start()

        Catch ex As Exception

            Form1.RTB\_Log.AppendText(ex.Message & vbNewLine)

            Return False

        End Try

        'Now start polling for a client using our new multithreaded version of TcpClient

        listenerSocket.startClientPoll(listenerSocket)

        Form1.RTB\_Log.AppendText("Created TCP listener on: " & LocalIP.ToString & ":" & port & vbNewLine)

        'listenerSocket.Stop()

        'Add a new handler that handles when the pollClientConnetion fires and a new user wants to connect

        AddHandler listenerSocket.clientConnecting, AddressOf Me.client\_Connect

        Return True

    End Function

    Private Sub client\_Connect()

        'Design of the intitial connection message:

        'start | UNAMELENGHT | UNAME | PKLENGHT | PK  | end

        '      |     BYTE    | BYTES |   BYTE   |BYTES|

        Dim username As String

        Dim IPaddr As String

        Dim endpoint As IPEndPoint

        Dim netStream As NetworkStream

        Dim unameLength As Integer

        Dim pkLength As Integer

        MTMainThreadWriter.writeLog("New user awaiting acept" & vbNewLine)

        'If there isnt an instance of the lookup table create one

        If lookupTable Is Nothing Then

            lookupTable = New ClientLookupTable

        End If

        'Accept the clients connection

        Dim client As TcpClient = listenerSocket.AcceptTcpClient()

        'Get the remote end point and then the IP address of the client

        endpoint = client.Client.RemoteEndPoint

        IPaddr = endpoint.Address.ToString

        If lookupTable.CheckClientConnected(IPaddr) = ClientLookupTable.clientLookupTableErrors.CLIENT\_CONNECTED Then

            MTMainThreadWriter.writeLog("Client already connected" & vbNewLine)

            client.Close()

            Return

        End If

        Dim recvBytes(client.ReceiveBufferSize) As Byte

        'sync the networkstream with the client stream

        netStream = client.GetStream

        'Read the data from the stream in order to get the username, public key from the client

        netStream.Read(recvBytes, 0, client.ReceiveBufferSize)

        '#Region "All of this is incorrect and needs replacing when the logon system is implemented"

        'we need to get username length

        unameLength = CInt(recvBytes(0))

        'loop through the values from point 1 in the Received data array to the end of username, we dont need to minus 1 because we have offset the data by 1

        For i = 1 To unameLength

            username = username & System.Convert.ToChar(recvBytes(i))

        Next

        'Now get the public key lenght that is right after the username as shown at the start

        pkLength = CInt(recvBytes(unameLength + 1))

        Dim publicKey(pkLength - 1) As Byte

        'Now just loop through as you normally would and offest the recvBytes array by usernamelength and a value of 2 because of the 2 bytes used for data lengths

        For i = 0 To pkLength - 1

            publicKey(i) = recvBytes(i + unameLength + 2)

        Next

        'Check if the details are null

        If publicKey Is Nothing Or username = Nothing Then

            MTMainThreadWriter.writeLog("Client connection details are NULL" & vbNewLine)

            Return

        End If

        '#End Region

        'Get all of the currently connected clients

        Dim connectedClientsValues() As ClientLookupTable.tableValues = lookupTable.GetConnectedClients()

        Dim numberOfClientsBytes() As Byte = BitConverter.GetBytes(connectedClientsValues.Length)

        'send the count of packets that we will send, and transmit that first

        netStream.Write(numberOfClientsBytes, 0, numberOfClientsBytes.Length)

        netStream.Flush()

        Const usernameLen As Integer = 20

        'send:

        '| username | public key |

        For i = 0 To connectedClientsValues.Length - 1

            Dim message(usernameLen + connectedClientsValues(i).publicKey.Length) As Byte

            Dim usernameBytes() As Byte = ASCII.GetBytes(connectedClientsValues(i).username)

            usernameBytes.CopyTo(message, 0)

            connectedClientsValues(i).publicKey.CopyTo(message, usernameLen)

            netStream.Write(message, 0, message.Length)

            netStream.Flush()

        Next

        'Get the unique client number based off of the number of clients in the DataGridView element

        Dim clientCount As Integer = MTMainThreadWriter.GetDGVCount

        'Start the instance of the multithreaded client sender and reciever

        Dim clientHandler As New MTClientReciever(client, clientCount)

        'Add 2 handlers, 1 to deal with removing the client from lookupTable and one to deal with the data once Received

        AddHandler clientHandler.clientDisconnect, AddressOf clientDisconnected

        AddHandler clientHandler.ReceivedData, AddressOf processClientData

        Dim lookupTableValues As New ClientLookupTable.tableValues

        lookupTableValues.clientReference = clientHandler

        lookupTableValues.publicKey = publicKey

        lookupTableValues.username = username

        lookupTable.sendConnectedClientBroadcast(ASCII.GetBytes(username), publicKey)

        'try and populate the table with the values above and the IP address as the lookup key

        If lookupTable.AddNewClient(IPaddr, lookupTableValues) = ClientLookupTable.clientLookupTableErrors.TABLE\_CONTAINS\_CLIENT Then

            MTMainThreadWriter.writeLog("Client already connected" & vbNewLine)

            clientHandler.closeClientConnection(False)

            Return

        End If

        MTMainThreadWriter.AddClientToDGV(username, IPaddr, publicKey)

        MTMainThreadWriter.writeLog("[Client: " & clientCount & "]: " & "Created new client with connection number: " & clientCount & vbNewLine)

    End Sub

    Private Sub clientDisconnected(ByVal IPAddr As String)

        'Remove the client from the lookuptable

        Dim username As String = lookupTable.GetClientUname(IPAddr)

        If lookupTable.RemoveClient(IPAddr) = ClientLookupTable.clientLookupTableErrors.CLIENT\_NOT\_CONNECTED Then

            MTMainThreadWriter.writeLog("Tried removing the client " & IPAddr & " but it did not exist")

            Return

        End If

        lookupTable.sendDisconnectedClientBroadcast(ASCII.GetBytes(username))

    End Sub

    Private Sub processClientData(ByVal data() As Byte, ByVal IPAddr As String, ByVal clNumber As Integer)

        Dim dataValue As Integer

        Try

            'try converting the data to an integer value, but possible that it overflows, so catch this

            dataValue = BitConverter.ToInt32(data, 0)

        Catch ex As Exception

            'value overflowed 32bits, is okay because then we know its not hearbeat or a dead signal

        End Try

        If dataValue = 57005 Then

            'Check for client disconnect value (0xDEAD) and remove any instances of that client

            Dim client As MTClientReciever = lookupTable.GetClientReference(IPAddr)

            client.closeClientConnection(True)

        Else

            Dim client As MTClientReciever = lookupTable.GetClientReference(IPAddr)

            Dim messageType As Byte = data(0)

            Dim lenghtOfUsefulData As Integer = data.Length

            Dim senderUsername(19) As Byte

            Const usernameLength As Integer = 20

            Const offset As Integer = 3

            Const hashLen As Integer = 32

            Dim sha256 As New SHA256Managed

            Array.Copy(ASCII.GetBytes(lookupTable.GetClientUname(IPAddr)), senderUsername, ASCII.GetByteCount(lookupTable.GetClientUname(IPAddr)))

            'We need to reprocess the data for the end user, which means changing the username from the recipients to the senders and therefore altering the username length

            '| Message Type | x | of y | username | data hash | data |

            '|     Byte     | B |   B  | 20 bytes | 32 bytes  | Bytes|

            If data(0) <> 1 And data(0) <> 2 Then

                client.sendError()

                Return

            End If

            If data(1) = data(2) Then

                For i = data.Length - 1 To 0 Step -1

                    If data(i) <> 0 Then

                        Exit For

                    End If

                    If lenghtOfUsefulData = usernameLength + 3 Then

                        Exit For

                    End If

                    lenghtOfUsefulData -= 1

                Next

            End If

            Dim x As Byte = data(1)

            Dim y As Byte = data(2)

            'Dim usernameLength As Byte = data(3)

            'retrieve the full 20 bytes of the username from the message

            Dim username(usernameLength - 1) As Byte

            For i = 0 To usernameLength - 1

                username(i) = data(i + offset)

            Next

            Dim lastNonZeroIndex As Integer = 19

            'find the last index in the username array that has a byte that is != 0

            For i = 19 To 0 Step -1

                If username(i) <> 0 Then

                    Exit For

                End If

                lastNonZeroIndex -= 1

            Next

            Dim nonZeroUsername(lastNonZeroIndex) As Byte

            'Cut off the zero bytes by copying the userful data into a buffer

            Array.Copy(username, nonZeroUsername, lastNonZeroIndex + 1)

            If x = 0 Then

                Const filenameLen As Integer = 100

                Dim initalMessage(data.Length - 1) As Byte

                Dim fileName(filenameLen - 1) As Byte

                For i = 0 To filenameLen - 1

                    fileName(i) = data(i + offset + usernameLength)

                Next

                Dim recp As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(nonZeroUsername))

                If recp = Nothing Then

                    'Got to now store the message for later, that user is not currently logged on

                    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recp & vbNewLine)

                    Return

                End If

                client.sendACK()

                'Here we get that clients reference

                Dim reference As MTClientReciever = lookupTable.GetClientReference(recp)

                initalMessage(0) = messageType

                initalMessage(1) = x

                initalMessage(2) = y

                senderUsername.CopyTo(initalMessage, offset)

                fileName.CopyTo(initalMessage, offset + usernameLength)

                reference.sendData(initalMessage)

                MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Forwarding data " & IPAddr & " >> " & recp & vbNewLine)

                Return

            End If

            'printArray(data, 10)

            Dim hashBytes(hashLen - 1) As Byte

            For i = 0 To hashLen - 1

                hashBytes(i) = data(i + offset + usernameLength)

            Next

            Dim encryptedData(lenghtOfUsefulData - offset - usernameLength - hashLen - 1) As Byte

            'encryptedData = copyToMax(data, offset + usernameLength + hashLen, lenghtOfUsefulData - 1)

            Array.ConstrainedCopy(data, offset + usernameLength + hashLen, encryptedData, 0, lenghtOfUsefulData - offset - usernameLength - hashLen)

            Dim fs As New IO.FileStream(Environment.CurrentDirectory + "\rnd.txt", IO.FileMode.Create)

            fs.Write(encryptedData, 0, encryptedData.Length)

            fs.Close()

            Dim genHashBytes() As Byte = sha256.ComputeHash(encryptedData)

            If genHashBytes.Length > hashLen Then

                MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "[ERROR] Generated hash for the Received data longer than max allowed" & vbNewLine)

                'Not much we can do to deal with this, its a programming error

                client.sendACK()

                Return

            End If

            'Compare the two hashes of the encrypted data, in order to make sure there were no errors when transmitting the file

            For i = 0 To hashLen - 1

                If hashBytes(i) <> genHashBytes(i) Then

                    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "[ERROR] Generated hash did not match the hash Received from the client" & vbNewLine)

                    client.sendError()

                    Debug.Print("hash")

                    Return

                End If

            Next

            'The hashes were equal so we may now send the ACK so the client sends the next set of data

            client.sendACK()

            'We need to get the recipients IP address from our lookup table so we can get the reference of the instance of MTClientReciever for that client

            Dim recpIPAddr As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(nonZeroUsername))

            'Dim s As New IO.FileStream(Environment.CurrentDirectory & "/" & x & ".txt", IO.FileMode.Create)

            's.Write(encryptedData, 0, encryptedData.Length)

            's.Close()

            If recpIPAddr = Nothing Then

                'Got to now store the message for later, that user is not currently logged on

                MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recpIPAddr & vbNewLine)

                Return

            End If

            'Here we get that clients reference

            Dim recpientReference As MTClientReciever = lookupTable.GetClientReference(recpIPAddr)

            'Send a full frame of bytes, therefore we know the max amount to recieve upon intercept, and therefore no more TCP problems, this also makes hashing now redundant, but keeping it as a backup

            Dim newData(data.Length - 1) As Byte

            newData(0) = messageType

            newData(1) = x

            newData(2) = y

            'newData(3) = senderUsername.Length

            senderUsername.CopyTo(newData, offset)

            hashBytes.CopyTo(newData, offset + usernameLength)

            encryptedData.CopyTo(newData, offset + usernameLength + hashLen)

            recpientReference.sendData(newData)

            MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Forwarding data " & IPAddr & " >> " & recpIPAddr & vbNewLine)

        End If

    End Sub

    Private Function copyToMax(ByVal array() As Byte, ByVal start As Integer, ByVal final As Integer) As Byte()

        '| offset | username | hash | data |

        '|    3   |    20    |  32  |   ?  |

        '0        2          22     54     ?

        Dim finalisedArray(final - start) As Byte

        Dim i As Integer

        For i = start To final

            finalisedArray(i - start) = array(i)

        Next

        Debug.Print(i)

        Return finalisedArray

    End Function

    Public Sub TcpListener\_Stop()

        'Kill off everything to stop the server running

        If listenerSocket Is Nothing Then

            Return

        End If

        listenerSocket.stopListener()

        If lookupTable Is Nothing Then

            Return

        End If

        lookupTable.CloseAllClientConnections()

    End Sub

    Public Shared Function Convert\_IP\_DecimalNotated(ByVal IP As String) As Byte()

        '0.0.0.0

        'This converts the decimal notated ip like so above to bytes

        If IP = Nothing Then

            Form1.RTB\_Log.AppendText("Provided IP was null" & vbNewLine)

            Return Nothing

        End If

        Dim s(3) As String

        Dim IPBytes(3) As Byte

        s = IP.Split(".")

        If s.Count <> 4 Then

            Form1.RTB\_Log.AppendText("Tried converting IP to byte array, the number of decimals was not equal to 3" & vbNewLine)

            Return Nothing

        End If

        For i = 0 To s.Length - 1

            Try

                IPBytes(i) = System.Convert.ToByte(s(i))

            Catch ex As Exception

                Form1.RTB\_Log.AppendText("Failed to convert to byte, invalid IP address" & vbNewLine)

                Return Nothing

            End Try

        Next

        Return IPBytes

    End Function

End Class

#### Client functions encapsulation

As seen in **SocketMT** above and in the class diagram on page 38, a new object is created for every client that connects. This object contains all of the multithreading and code necessary to deal with communicating to a single client and by multithreading all of the clients, allows the server to asynchronously communicate with the users therefore alleviating the server from backlog. As seen in the procedure **Client\_Connect**, a new instance of this class is created each time a user connects and the reference to this class is stored inside a hashtable, so that when **Client\_Connect** returns the reference to that class is not lost and the object is not disposed of by the garbage collector.

When creating a new object out of the **MTClientReceiver** class, we firstly call sub **New()**:

Public Sub New(ByRef client As TcpClient, ByVal clNumber As Integer)

Dim endpoint As IPEndPoint = client.Client.RemoteEndPoint

endpointIP = endpoint.Address.ToString

'Create the network stream to get data in and out

stream = client.GetStream

'store the tcpClient reference and the clients unique number

Me.client = client

Me.clNumber = clNumber

'start the thread that polls for sent messages

Dim RThread As Threading.Thread = New Threading.Thread(AddressOf pollClientSend)

RThread.Start()

'set the recieve buffer size and start the heartbeat timer to check if the client has timed out

recvBuffLenght = client.ReceiveBufferSize

HeartBeatTimer = New Heartbeat()

End Sub

Firstly we set the corresponding client’s endpoint IP address for use later, we also get the network stream using **client.GetStream** allowing us to send a receive data from the user. Finally, we set the reference to **TcpClient** and set the client’s unique number for use when writing out to logs, start the thread that checks for information that has been sent and set the maximum size of the buffer and start the heartbeat timer.

When we start the new thread in sub **New** it sets a loop going to continuously checks the stream for available data, so long as the resource it’s trying to access is not busy with sending data. This then calls a subroutine to read the net stream:

Private Sub pollClientSend()

    Do

        If stream.DataAvailable And canRead Then

            'we need to raise the event of client has sent data, we must also reset the timer due to the fact that the client is not idle but alive

            HeartBeatTimer.Timer.Stop()

            dataReciever(Nothing)

            Try

                HeartBeatTimer.Timer.Start()

            Catch ex As Exception

                'No need to worry by raising an event we need to return from the event at some point, if the client disconnected we dispose of the timer, therefore meaning when we return from the event the timer is disposed of and we crash

                'Now we dont crash and everything is okay anyway

                'The thread should now end anyway

            End Try

        End If

    Loop Until Ended

End Sub

As you can see, the loop continues to a private global variable for the class is set true, which indicates if the TCP server is being stopped. We stop the heartbeat timer while we receive data because it indicates that the connection is still alive and as well prevents the heartbeat timer writing the heartbeat value in the middle of a packet by accident. Upon starting the timer again, it is possible for the program to crash, due to the fact that if an exception is thrown when the data is sent, the timer is disposed of, and thus when execution returns to the sub above, an **ObjectDisposed** exception can be thrown by calling start timer again.

When the event is raised above, execution vectored to a subroutine called **dataReciever**, where the data is read from the stream and the stream lock is taken off, allowing the network stream to be accessed again.

Private Sub dataReciever(ByVal data() As Byte)

    Dim recvBytes(recvBuffLenght - 1) As Byte

    Dim recvLength As Integer

    Dim endpoint As IPEndPoint = client.Client.RemoteEndPoint

    'lock out the network stream so that a deadlock isnt created, might be a bit pedantic but to be safe

    canWrite = False

    'read the data in

    If data Is Nothing Then

        recvLength = stream.Read(recvBytes, 0, recvBuffLenght)

    Else

        recvBytes = data

    End If

    'check if the vlaue is hearbeat, and change the output to the log

    If BitConverter.ToInt32(recvBytes, 0) = 43775 Then

        MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Heartbeat, client still alive" & vbNewLine)

Return

    Else

        MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Received data from client" & vbNewLine)

    End If

    'set it so that we can write to the stream again

    canWrite = True

    'raise the event for the socketMT class to deal with and decode what the data means

    RaiseEvent ReceivedData(recvBytes, endpoint.Address.ToString, clNumber)

End Sub

This subroutine is merely for reading data from a network stream ready for the **socketMT** class to process the data. The reason why this is required is because **socketMT** has containing methods for sending the data to a receiving client which **MTClientReceiver** does not have access too. So, while the data is gathered from the network stream, we impose a lock on the network stream so that data cannot be written to it while we read from it. We also have the option of passing data to it, in this situation where data is not set to nothing, is a problem with message receiving at the moment, where **MTSend** waits until it receives the ACK from the server. If data that is received that way is not an ACK value or an error value then we called this subroutine to decode what the client had sent. In a future revision this will become depreciated as the problem listed above is due to multithreading and how keep a lock on what is writing data to the clients, the changes made are listed on page.

In order to send data to the client I created a simple multithreaded design to send the data without freezing up the main thread of the server. This design consists of two subroutines one public and one private to send the data, first one initialises the second procedure as a new thread for execution. The public sub looks like so:

Public Sub sendData(ByVal Data() As Byte)

    'start new thread for sending messages and pass the data to it

    Dim SThread As New Threading.Thread(New Threading.ParameterizedThreadStart(AddressOf MTSend))

    SThread.Start(Data)

End Sub

Initially the design was simple like above, but later needed to be adapted to provide a Boolean value to specify whether the data being sent was reply to the client (ACK or error):

Public Sub sendData(ByVal Data() As Byte, Optional ByVal errorBytes As Boolean = False)

    'start new thread for sending messages and pass the data to it

    Dim parameters As New sendingParameters

    parameters.Data = Data

    parameters.errorBytes = errorBytes

    Dim SThread As New Threading.Thread(New Threading.ParameterizedThreadStart(AddressOf MTSend))

    SThread.Start(parameters)

End Sub

As you can see an optional value was added specifying if the data being sent is error bytes, but due to limitations of starting a parameterized thread, a structured had to be created which specified the data being sent to the thread:

Private Structure sendingParameters

    Public Data() As Byte

    Public errorBytes As Boolean

End Structure

By using this we can easily send two or more parameters to the thread without having to remove threading, using async as a replacement.

Private Sub MTSend(ByVal data() As Byte)

    'sleep the thread until the stream becomes available

    Do

        Threading.Thread.Sleep(100)

    Loop Until canWrite And canRead

    'stop the stream from being read until we flush the data out of it

    canRead = False

    'reset the timer like before, if we fail to write the client is already disconnected and the heartbeat wont help us here, possible it could write data to the stream the same time as we do, invalidating the data

    HeartBeatTimer.Timer.Stop()

    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Sending data to client" & vbNewLine)

    Try

        stream.Write(data, 0, data.Length)

    Catch ex As Exception

        'shutdown the clients connection, its dead

        closeClientConnection(True)

        canRead = True

        Return

    End Try

    'flush the stream, start the timer and allow the stream to be read

    stream.Flush()

    HeartBeatTimer.Timer.Start()

    canRead = True

End Sub

This was the initial design of sending data back to the client, before hashes for the data was implemented. This design starts off with a wait loop, which pauses the thread until the lock on the stream has been removed. The thread then locks down the stream for execution and sets the heartbeat timer to disabled, thus resetting it too. Once the timer is stopped, the data provided is written to the stream and the data sent by the client is flushed from the stream, sending it too the client. After this, the timer is started and the lock is then removed and the next thread is able to repeat the data. Once hashes were introduced to the decoding phase the client required a way of determining if the data received was received correctly. We use two different hexadecimal values to represent correctly sent data and then call the **sendData** sub, passing those hex values to it.

Public Sub sendError()

    Dim value As Integer = 61149

    Dim errorByte() As Byte = BitConverter.GetBytes(value)

    sendData(errorByte, True)

End Sub

Public Sub sendACK()

    Dim value As Integer = 43724

    Dim errorByte() As Byte = BitConverter.GetBytes(value)

    sendData(errorByte, True)

End Sub

This then required a change to sending the data, when the client receives the data they will also reply with values like those seen above. This required a new design to be implemented:

Private Sub MTSend(ByVal params As sendingParameters)

    'sleep the thread until the stream becomes available

    Do

        Threading.Thread.Sleep(100)

    Loop Until canWrite And canRead

    'stop the stream from being read until we flush the data out of it

    canRead = False

    'reset the timer like before, if we fail to write the client is already disconnected and the heartbeat wont help us here, possible it could write data to the stream the same time as we do, invalidating the data

    HeartBeatTimer.Timer.Stop()

    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Sending data to client" & vbNewLine)

    Try

        stream.Write(params.Data, 0, params.Data.Length)

    Catch ex As Exception

        'shutdown the clients connection, its dead

        closeClientConnection(True)

        canRead = True

        Return

    End Try

    'flush the stream, start the timer and allow the stream to be read

    stream.Flush()

    If params.errorBytes Then

        canRead = True

        Return

    End If

    Dim serverResponseBytes(recvBuffLenght - 1) As Byte

    Dim value As Integer

    Try

        stream.Read(serverResponseBytes, 0, recvBuffLenght)

    Catch ex As Exception

        canRead = True

        Return

    End Try

    Try

        value = BitConverter.ToInt32(serverResponseBytes, 0)

    Catch ex As Exception

        'Dont worry

    End Try

    If value = 61149 Then

        Dim ret As Boolean = trySendAgain(params.Data)

        If ret = False Then

            closeClientConnection(True)

            canRead = True

            Return

        End If

    ElseIf Not value = 43724 Then

        'Throw New Exception

        dataReciever(serverResponseBytes)

        'trySendAgain(params.Data)

    End If

    HeartBeatTimer.Timer.Start()

    canRead = True

End Sub

Private Function trySendAgain(ByVal data() As Byte) As Boolean

    Try

        stream.Write(data, 0, data.Length)

    Catch ex As Exception

        Return False

    End Try

    stream.Flush()

    Dim serverResponseBytes(recvBuffLenght - 1) As Byte

    Dim value As Integer

    Try

        stream.Read(serverResponseBytes, 0, recvBuffLenght)

    Catch ex As Exception

        Return False

    End Try

    Try

        value = BitConverter.ToInt32(serverResponseBytes, 0)

    Catch ex As Exception

        'Dont worry

    End Try

    If value = 61149 Then

        If trySendAgain(data) = False Then

            Return False

        End If

    End If

    Return True

End Function

The new design is formed of two procedures, one a recursive function and the same subroutine as before with a few code modifications. As before data is sent and the stream is flushed, but checks are made to the value of **params.erroBytes**  this call determines if the data that was sent was a reply to the client, if so then the server doesn’t need to wait for a reply from the client. Otherwise execution continues and thread waits until data is received from the client, this data is then converted to an integer, and checked against two values, the first value (61149) indicates that there was an error of some form when receiving the data, the second (43724) indicates that the server received the data correctly and thus the sub can exit and the data doesn’t have to be resent. If the value of the check is equal to 61149 then the data is passed to the recursive function **trySendAgain**, which in essence repeats the same procedure as in MTSend, except returning a Boolean value depending if the process failed or passed. When the function returns false a call to close connections is made due to the only situation where the false is returned, is when there was an error resending the information.

There is one problem with this design, there is queuing system in place for the threads, but the system is disorderly meaning that its possible for any waiting thread to execute next, which is not the correct approach to this solution. The correct approach to this dilemma is to introduce a linear queue to order the data, insuring that replies are sent at the correct time. This new design fixed some errors where the data was not being sent and received in the correct order, this fix does not correct the problem with my simulation, but alas the simulation’s design flaw would never be intended to work in the implemented system. Design changes required to implement the linear queue firstly required a private global variable to allow access to the queue across subroutines and threads:

Private linearMessageQueue As New Queue

I then adapted the subroutine **sendData** to add the **sendingParameters** to the queue so that the **MTSend** thread can remove those from the queue and send it to the client. Rather than a new thread be created every time we send a message to the client, a thread is created at the start and that thread continuously polls for new data being added to the queue.

Public Sub New(ByRef client As TcpClient, ByVal clNumber As Integer)

Dim endpoint As IPEndPoint = client.Client.RemoteEndPoint

endpointIP = endpoint.Address.ToString

'Create the network stream to get data in and out

stream = client.GetStream

'store the tcpClient reference and the clients unique number

Me.client = client

Me.clNumber = clNumber

'start the thread that polls for sent messages

Dim RThread As Threading.Thread = New Threading.Thread(AddressOf pollClientSend)

RThread.Start()

'Start the thread that polls for new data in the queue

Dim SThread As Threading.Thread = New Threading.Thread(AddressOf MTSend)

SThread.Start()

'set the recieve buffer size and start the heartbeat timer to check if the client has timed out

recvBuffLenght = client.ReceiveBufferSize

HeartBeatTimer = New Heartbeat()

End Sub

**Sub New** was updated with two lines that start the thread that sends the data, this thread was updated slightly but features the same basic mechanics as before:

Private Sub MTSend()

Do

If linearMessageQueue.Count > 0 And canWrite Then

Dim params As sendingParameters = linearMessageQueue.Dequeue()

canRead = False

'reset the timer like before, if we fail to write the client is already disconnected and the heartbeat wont help us here, possible it could write data to the stream the same time as we do, invalidating the data

HeartBeatTimer.Timer.Stop()

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Sending data to client" & vbNewLine)

Try

stream.Write(params.Data, 0, params.Data.Length)

Catch ex As Exception

'shutdown the clients connection, its dead

closeClientConnection(True)

canRead = True

Return

End Try

'flush the stream, start the timer and allow the stream to be read

stream.Flush()

If Not params.errorBytes Then

Dim serverResponseBytes(recvBuffLenght - 1) As Byte

Dim value As Integer

Try

stream.Read(serverResponseBytes, 0, recvBuffLenght)

Catch ex As Exception

canRead = True

Return

End Try

Try

value = BitConverter.ToInt32(serverResponseBytes, 0)

Catch ex As Exception

'Dont worry

End Try

If value = 61149 Then

Dim ret As Boolean = trySendAgain(params.Data)

If ret = False Then

closeClientConnection(True)

canRead = True

Return

End If

ElseIf Not value = 43724 Then

'Throw New Exception

dataReciever(serverResponseBytes)

'trySendAgain(params.Data)

End If

HeartBeatTimer.Timer.Start()

End If

canRead = True

End If

Threading.Thread.Sleep(1000)

Loop Until Ended

End Sub

Like the thread that waits for data being received this is set in a continuous loop until the client closes the connection, whereby the value of **Ended** is set to true and the thread terminates. I thought it was a good idea to add in a 1s sleep after each loop in order to free up some processor time for other executions, there were noticeable resource consumptions on lower powered devices without the use of this the CPUs resources are maxed out and therefore problems are experience with non-responsiveness.

After testing this new implementation there was still problems that I encountered with sending data over TCP, after debugging I found that that the problem was being caused by the error of not waiting till all of the data is in the buffer. My misunderstanding of how the system worked forced me to implement a sha256 hash for the data, which was the incorrect approach to this problem. Due to this, once the rest of the sent data came in it would offset the starting bytes of the resent data, causing the hash check to fail again until the data was received correctly two times in a row. The method to solving this was to implement a looping function that would keep reading in data form the stream until the size of the buffer was filled. This design was correct but only worked for data that was longer than the max size of one packet (roughly 65000 bytes), initially I thought I would have to adapt the packets that have a variable length again by adding an 8 byte value for the length of the data. This would have worked but an easier solution was developed to the problem, rather than have a variable amount for the last packet and the initial packet for file sending, we pad the sending array out to the maximum value that I used for the other packets. Thus when we read in the data at the other end, we know how much we could receive and we can then easily remove the extra zeros from the end of the packet. Below are the respective changes made to the server in order to fix this problem:

Private Sub dataReciever(ByVal data() As Byte)

Dim recvBytes(recvBuffLenght - 1) As Byte

Dim recvLength As Integer

Dim endpoint As IPEndPoint = client.Client.RemoteEndPoint

'lock out the network stream so that a deadlock isnt created, might be a bit pedantic but to be safe

canWrite = False

'read the data in

If data Is Nothing Then

recvLength = stream.Read(recvBytes, 0, recvBuffLenght)

If (recvBytes(0) = 1 Or recvBytes(0) = 2) And recvLength <> recvBuffLenght Then

Dim dataCounter As Integer = recvLength

Do

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: Got " & dataCounter & " out of " & recvBuffLenght & vbNewLine)

recvLength = stream.Read(recvBytes, dataCounter, recvBuffLenght - dataCounter)

dataCounter += recvLength

Loop Until dataCounter = recvBuffLenght

End If

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "All data obtained" & vbNewLine)

Else

recvBytes = data

End If

'check if the vlaue is hearbeat, and change the output to the log

If BitConverter.ToInt32(recvBytes, 0) = 43775 Then

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Heartbeat, client still alive" & vbNewLine)

canWrite = True

Return

Else

MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Received data from client" & vbNewLine)

End If

'set it so that we can write to the stream again

canWrite = True

'raise the event for the socketMT class to deal with and decode what the data means

RaiseEvent ReceivedData(recvBytes, endpoint.Address.ToString, clNumber)

End Sub

Highlighted in the darker yellow is the change, ensuring all the data was received from the client. We initially read in as usual, recording the number of bytes read with **recvLength**, if all was successful then the value of bytes read should equal the value of **recvBuffLenght**, when the type of packet is an IM or a file. If that criteria isn’t met or the length of bytes read in is equal to the buffer size then there’s no need to perform the next step. If the **IF** statement returns true then we create a variable to store the count of bytes read so far, we then read in some more data of the length buffer size minus the byte we have already read, the total counter of bytes read in is then updated with the number of bytes read this time, if the new value of bytes read is still not equal to the buffer size then the read is repeated until this value is met.

Finally in the class **MTClientReciever**, there are two required subroutines to disconnect the client, the first one deals with disconnections by the heartbeat timer, seen below, and the other is a subroutine that is called to disconnect a client by other classes.

    Private Sub NoHeartbeat() Handles HeartBeatTimer.timerFiredNoResponse

        'correctly stop the thread

        Ended = True

        Try

            HeartBeatTimer.Timer.Dispose()

        Catch ex As Exception

            'No need to worry, timer is already stopped

        End Try

        'dispose of the stream and close the tcp connection

        stream.Dispose()

        client.Close()

        'write to the log, remove the client from the DataGridViewer in the UI and finally remove the client from the lookupTable

        MTMainThreadWriter.writeLog("[Client: " & clNumber & "]:" & "No heartbeat packet was Received, or an error happened when writing to the stream, client disconnected" & vbNewLine)

        MTMainThreadWriter.RemoveClientFromDGV(clNumber)

        RaiseEvent clientDisconnect(endpointIP)

    End Sub

    Public Sub closeClientConnection(ByVal raisingevent As Boolean)

        If Ended = True Then

            Return

        End If

        Ended = True

        Try

            HeartBeatTimer.Timer.Dispose()

        Catch ex As Exception

            'No need to worry, timer is already stopped

        End Try

        stream.Dispose()

        client.Close()

        MTMainThreadWriter.writeLog("[Client: " & clNumber & "]:" & " Client disconnected" & vbNewLine)

        MTMainThreadWriter.RemoveClientFromDGV(clNumber)

        'We need to raise an event so that socketMT (which has accesss to the lookuptable) can remove us from logged on users

        'Should be boolean because if clearing out table then we dont need unecessary raiseEvent

        If raisingevent Then

            RaiseEvent clientDisconnect(endpointIP)

        End If

    End Sub

End Class

Simply what both procedures do is, dispose of the timer, the stream and the **TCPClient** object, and remove the client from the table in the main UI. But in **NoHeartbeat** we always raise the event, but in the second the event is raised based on a Boolean value, this is because if the heartbeat client calls it then that’s the first time we know that the client has disconnected, but the Boolean value in the second one is needed because when the **ClientLookupTable** is being cleared out, then we don’t need to call the function that removes it from the table again.

##### Handling client crashes

In a situation where a computer may crash or lose power, the server may not be notified of the client’s disappearance, thus when a client disconnects without notifying the server, it may possible for the user to not be able to log on again due to blocking of connections with the same details as listed already in the server. The server may not even know the client has crashed until tries to send a message to the client, whereabouts an exception occurs and the server closes the connection. In this situation the use of a heartbeat tells the server that the client is still alive and connected. A heartbeat is a small packet of data sent to a server at regular intervals indicating that the client’s connection is still alive, once that time passes and no data was received the server then presumes that the client is deceased and the connection is forcibly closed.

Imports System.Net.Sockets

Public Class Heartbeat

    Public WithEvents Timer As System.Timers.Timer

    Public Event timerFiredNoResponse()

    Public Sub New()

        Timer = New System.Timers.Timer()

        Timer.Interval = 120000 '2 minutes

        'Timer.Interval = 3000 'used in testing

        Timer.Start()

    End Sub

    Private Sub timerFired() Handles Timer.Elapsed

        Timer.Stop()

        RaiseEvent timerFiredNoResponse()

    End Sub

End Class

Here we use the integrated timer class set to the maximum waiting time of 2 minutes, if a message is sent and it contains the data for the heartbeat then the timer is reset back to zero, if the timer expires the 2 minutes then an event is raised and **timerFired** handles that exact event, it then raises its own event for the class **MTClientReciever** to disconnect the client.

#### Databases

As a requirement for the server and the client, either one or the other must use databases to be able to store and retrieve messages on the fly, considering the approach I have taken to building the server and another requirement for a user to access messages from previous sessions and be able to send messages offline, I deemed it appropriate to have databases for both the server and the client, which perform different functions. For the server which this next section is specific to, the database is designed for holding the user’s credentials and also storing records of the messages that were sent offline to the server, by doing this it centralises the system for the time that this project would be put into a real-world scenario, making it easier for administrators to manage all of the clients at once. The database is formed of three tables, one storing the collection of users’ details, for example the passwords, the usernames and the privileges for that specific user, another table to store the offline messages and the final table to store the offline files that were sent. The EER diagram for the database would look something like this:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | TBL\_USERS | |  |  |  |  |  |
|  |  |  |  |  | **PK** | Username | 1 |  |  |  |  |
|  |  |  |  |  |  | PasswordHash |  |  |  |  |  |
|  |  |  |  |  |  | Administrator |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| TBL\_UNSENT\_MESSAGES | |  |  |  |  |  |  |  |  | TBL\_UNSENT\_FILES | |
|  | MessageData |  |  |  |  |  |  |  |  |  | Filepath |
| **FK** | SenderUsername | **∞** |  |  |  |  |  |  | **∞** | **FK** | SenderUsername |
| **FK** | RecipientUsername | **∞** |  |  |  |  |  |  | **∞** | **FK** | RecipientUsername |

Figure 26: Server Database

Above shows roughly what the database for the server will look like, the users table contains the username of the user and the password hash, both as long text and the field Administrator will be a Boolean field suggesting if the user has administrative rights. When the database is created a record will be automatically added, containing the username and password of a generic admin user, which an administrator can use in order to setup the database from a connected client. The values for this user will be something along the lines of Administrator as the username and Admin1 as the password, but rather than hashing the plaintext of the password, we set the known hash instead and convert it to base64 in order to store it in the database, the corresponding base encoded hash is equal to CvsAE42OczSOwf5B/T06j8vZAVayY7+leRug4JX0LPw= which we add to the database using an insert statement.

##### Tools for databases

As a starting platform I created a class that encapsulates the necessary database functions, for example sending the SQL queries and non-queries and checking for and creating a table for the client and server side, the code behind it ended up looking like so:

Imports System.IO

Imports System.Environment

Imports Error\_Handler.Events\_Handler

Imports System.Reflection.MethodInfo

Imports System.Data

Imports System.Data.OleDb

Public Class DatabaseTools

    Private dbWorkingDirectory As String = (CurrentDirectory & "\")

    Private Const dbConnectString As String = "Provider=Microsoft.Jet.OLEDB.4.0;"

    Private DBConnection As OleDbConnection = Nothing

    Public Enum databaseEnum

        ClientDatabase

        ServerDatabase

    End Enum

    Public Sub New(ByVal databaseFile As databaseEnum, Optional ByVal username As String = Nothing)

        If databaseFile = databaseEnum.ClientDatabase And username = Nothing Then

            errorHappened(username, "The username was NULL when checking for the users database", errorEnum.NULL\_REFERECE\_EXCEPTION, GetCurrentMethod.Name)

            Return

        End If

        If databaseFile = databaseEnum.ServerDatabase And Not File.Exists(dbWorkingDirectory & databaseFile.ToString & ".mdb") Then

            writeDebugLog("Database " & databaseFile.ToString & " could not be found", GetCurrentMethod.Name)

            createDatabase(dbWorkingDirectory & databaseFile.ToString & ".mdb", databaseEnum.ServerDatabase)

        ElseIf databaseFile = databaseEnum.ClientDatabase And Not File.Exists(dbWorkingDirectory & username & "\_" & databaseFile.ToString & ".mdb") Then

            writeDebugLog("Database " & username & "\_" & databaseFile.ToString & " could not be found", GetCurrentMethod.Name)

            createDatabase(dbWorkingDirectory & username & "\_" & databaseFile.ToString & ".mdb", databaseEnum.ClientDatabase)

        End If

        If databaseFile = databaseEnum.ServerDatabase Then

            DBConnection = New OleDbConnection(dbConnectString & "Data Source=" & dbWorkingDirectory & databaseFile.ToString & ".mdb")

            DBConnection.Open()

        Else

            DBConnection = New OleDbConnection(dbConnectString & "Data Source=" & dbWorkingDirectory & username & "\_" & databaseFile.ToString & ".mdb")

            DBConnection.Open()

        End If

    End Sub

    Public Sub createDatabase(ByVal filePath As String, ByVal databaseFile As databaseEnum)

        Dim cat As New ADOX.Catalog

        cat.Create(dbConnectString & "Data Source=" & filePath)

        DBConnection = New OleDbConnection(dbConnectString & "Data Source=" & filePath)

        DBConnection.Open()

        Dim SQL As String

        If databaseFile = databaseEnum.ClientDatabase Then

            'SQL = "CREATE DATABASE " & Path.GetFileNameWithoutExtension(filePath) & ";"

            'sendSQLNonQuery(SQL)

            SQL = "CREATE TABLE TBL\_USERS ([Username] TEXT NOT NULL PRIMARY KEY);"

            sendSQLNonQuery(SQL)

            SQL = "CREATE TABLE TBL\_FILES ([FileID] COUNTER NOT NULL UNIQUE PRIMARY KEY, [Filename] TEXT, [Sender\_UserID] TEXT, CONSTRAINT FK\_SenderUsername FOREIGN KEY ([Sender\_UserID]) REFERENCES TBL\_USERS([Username]));"

            sendSQLNonQuery(SQL)

            SQL = "CREATE TABLE TBL\_MESSAGES ([MessageID] COUNTER NOT NULL UNIQUE PRIMARY KEY, [MessageData] LONGTEXT, [ReceivedDate] LONGTEXT, [Sender\_UserID] LONGTEXT, CONSTRAINT FK\_SenderUsername2 FOREIGN KEY ([Sender\_UserID]) REFERENCES TBL\_USERS([Username]));"

            sendSQLNonQuery(SQL)

        Else

            'SQL = "CREATE DATABASE " & Path.GetFileNameWithoutExtension(filePath) & ";"

            'sendSQLNonQuery(SQL)

            SQL = "CREATE TABLE TBL\_USER ([Username] TEXT NOT NULL PRIMARY KEY, [Password\_Hash] LONGTEXT NOT NULL, [Is\_Administrator] YESNO, [Last\_Known\_PublicKey] LONGTEXT);"

            sendSQLNonQuery(SQL)

            SQL = "INSERT INTO TBL\_USER ([Username], [Password\_Hash], [Is\_Administrator]) VALUES ('Administrator', 'CvsAE42OczSOwf5B/T06j8vZAVayY7+leRug4JX0LPw=', TRUE);"

            sendSQLNonQuery(SQL)

            SQL = "CREATE TABLE TBL\_UNSENT\_FILES ([FileID] COUNTER NOT NULL UNIQUE PRIMARY KEY, [Filepath] LONGTEXT, [Sender\_UserID] TEXT, [Recipient\_UserID] TEXT, CONSTRAINT FK\_SenderUsername FOREIGN KEY ([Sender\_UserID]) REFERENCES TBL\_USER([Username]), CONSTRAINT FK\_RecipientUsername FOREIGN KEY ([Recipient\_UserId]) REFERENCES TBL\_USER([Username]));"

            sendSQLNonQuery(SQL)

            SQL = "CREATE TABLE TBL\_UNSENT\_MESSAGES ([MessageID] COUNTER NOT NULL UNIQUE PRIMARY KEY, [MessageData] LONGTEXT, [Sender\_UserID] TEXT, [Recipient\_UserID] TEXT, CONSTRAINT FK\_SenderUsername2 FOREIGN KEY ([Sender\_UserID]) REFERENCES TBL\_USER([Username]), CONSTRAINT FK\_RecipientUsername2 FOREIGN KEY ([Recipient\_UserId]) REFERENCES TBL\_USER([Username]));"

            sendSQLNonQuery(SQL)

        End If

    End Sub

    Public Sub sendSQLNonQuery(ByVal SQL As String)

        Dim cmd As New OleDbCommand(SQL, DBConnection)

        cmd.ExecuteNonQuery()

    End Sub

    Public Function sendSQLQuery(ByVal SQL As String) As OleDbDataReader

        Dim cmd As New OleDbCommand(SQL, DBConnection)

        Dim reader As OleDbDataReader = cmd.ExecuteReader()

        Return reader

    End Function

End Class

As you can see, **sub new** performs some basic error checking on the provided parameters first, for example if the user wants to open a client’s database a username must be specified, as the databases are specific to each user, so that each can be encrypted later on with that user’s session key, the subroutine checks if a username is supplied when the ClientDatabase enumeration is used. Next, we check if the databases exist, if it occurs that the databases could not be found then we call the subroutine that creates the databases, if the databases are found then we open the database as a new **OleDbConneciton** so that we can send queries to the database manager. If the database doesn’t exist the sub **CreateDatabase** is called and it consists of two sections, one has all of the SQL to create the client’s database and one has the SQL to create the server’s database. When creating the databases, I had some problems I encountered, because the databases are in a specific format that I didn’t realise till later I was just running **File.Create(<DatabaseName.mdb>)** this was failing to work because when we opened the database connection an error was thrown and it was not possible to create tables, further research found that there was no simple way of doing it from vb.net and the method of doing it was to add a reference to Microsoft’s ADO COM interface, then I can use the ADOX catalogue to create the new database with the provider being the JET engine. Once the databases are created we can set the private variable **DBConnection** if it wasn’t set already before. This now allows us to use the two procedures, **sendSQLNonQuery** and **sendSQLQuery** allowing us to use the database with one command. As a future update its advisable to add a sync lock around the code inside of these variables and add sync locks to anywhere that uses the **OleDbDataReader**, this just adds a bit of assurance to anywhere where the database is being used, to stop multiple threads creating a race condition on the database.

##### Connecting a user

After having the database tools setup, I moved on to adding authentication for a connecting user in the **SendRecieve** class under the subroutine **ClientConnecting**, the process to authenticating a user meant I had to add the password hash to the data that is sent from the client in order to check it with the hash in the server’s database. In order to get the hash from the database we use the select statement to return the hash where the username equals the one provided by the client, when also retrieving the hash I also obtained the value of **IsAdministrator**, in order to be able to send a value back to the client indicating successful connection. The code behind this looks something like this:

'Get the password hash from the database

Dim SQL As String

SQL = "SELECT Password\_Hash, Is\_Administrator FROM TBL\_USER WHERE Username = '" & username & "';"

Dim dbPWHashReturn As OleDb.OleDbDataReader = databaseConnections.sendSQLQuery(SQL)

dbPWHashReturn.Read()

If Not dbPWHashReturn.HasRows Then

    dbPWHashReturn.Close()

    MTMainThreadWriter.writeLog("Connecting client didnt not provide correct username, password combo, the connection was denied" & vbNewLine)

    netStream.WriteByte(CByte(1))

    client.Close()

    netStream.Dispose()

    Return

End If

Dim dbPWHash() As Byte = System.Convert.FromBase64String(dbPWHashReturn(0).ToString)

Dim isAdmin As Boolean = dbPWHashReturn.GetBoolean(1)

dbPWHashReturn.Close()

'Check if the password hashes are equal

For i = 0 To 31

    If dbPWHash(i) <> passwordHash(i) Then

        MTMainThreadWriter.writeLog("Connecting client didnt not provide correct username, password combo, the connection was denied" & vbNewLine)

        netStream.WriteByte(CByte(1))

        client.Close()

        netStream.Dispose()

        Return

    End If

Next

MTMainThreadWriter.writeLog("User was succesfully authenticated" & vbNewLine)

If lookupTable.CheckClientConnected(IPaddr) = ClientLookupTable.clientLookupTableErrors.CLIENT\_CONNECTED Then

    netStream.WriteByte(CByte(2))

    client.Close()

    Return

End If

If isAdmin Then

    netStream.WriteByte(CByte(4))

Else

    netStream.WriteByte(CByte(3))

End If

The statement:

"SELECT Password\_Hash, Is\_Administrator FROM TBL\_USER WHERE Username = '" & username & "';"

Returns the values of the password hash and the Boolean value for **IsAdministrator** when the **OleDbDataReader** is used. Once the values have been obtained, then they are compared with the ones provided by the client, if there are no values (**dbPWHashReturn.HasRows** is false) or the hashes are not equal then the value of 1 is sent to the client, which indicates that the provided details were wrong, after having check the details against what we have listed, we make sure that a client is not already connected with those details and that being the case, send either a three to the client if the client is a normal user or a four is the client is an administrator, its possible for an attacker to block a value and send back any of those values, but it wouldn’t have any negative repercussions as checks are performed server side for any change of data.

Once having added the login system into the server, we needed to update the section that sends back the list of clients in the same sub, in order for the design to work properly the client needs to know the other users’ public keys and needs to know all of the users on the sever, this required a design change to the database:

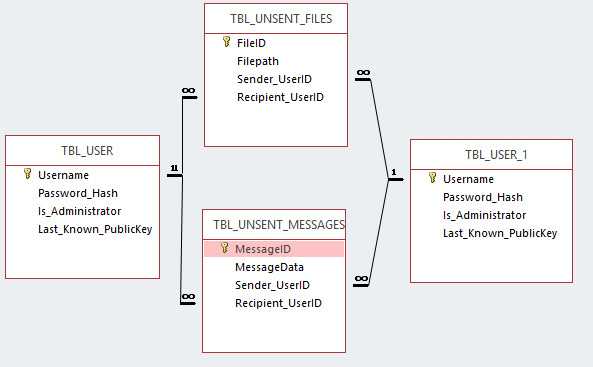


Figure 27: New server ER

As visible in the TBL\_USER and TBL\_USER\_1 which are the same table, a new field known as **Last\_Known\_PublicKey** has been added, this stores the value of the public key of the user from the last session, enabling us to send the offline user an encrypted message and still allow them to decrypt its value. Although when we create a new user, the value of this box cannot be set as the user has previously not had a session, so when sending the list of users back to the client, any client with that field set to NULL will not be enumerated from the database. Having offline users also required the data to be adapted, now when receiving users there is a byte at the end of the stream which indicates if the user is logged on:

'Get all of the currently connected clients

        Dim connectedClientsValues() As ClientLookupTable.tableValues = lookupTable.GetConnectedClients()

        Dim totalClientsReader As OleDb.OleDbDataReader = databaseConnections.sendSQLQuery("SELECT COUNT(Username) FROM TBL\_USER WHERE Username <> '" & username & "' AND Last\_Known\_PublicKey <> 'NULL';")

        totalClientsReader.Read()

        Dim totalClientCount As Integer = totalClientsReader.GetInt32(0)

        totalClientsReader.Close()

        Dim totalClientsInDB(totalClientCount - 1) As databaseUserReturn

        If totalClientCount <> 0 Then

            totalClientsReader = databaseConnections.sendSQLQuery("SELECT [Username],[Last\_Known\_PublicKey] FROM TBL\_USER WHERE Username <> '" & username & "' AND Last\_Known\_PublicKey <> 'NULL';")

            For i = 0 To totalClientCount - 1

                totalClientsReader.Read()

                totalClientsInDB(i).username = totalClientsReader.GetString(0)

                totalClientsInDB(i).publicKey = totalClientsReader.GetString(1)

            Next

        End If

        totalClientsReader.Close()

        Dim numberOfClientsBytes() As Byte = BitConverter.GetBytes(totalClientsInDB.Length)

        'send the count of packets that we will send, and transmit that first

        netStream.Write(numberOfClientsBytes, 0, numberOfClientsBytes.Length)

        netStream.Flush()

        Const usernameLen As Integer = 20

        'send:

        '| username | connected | public key |

        For i = 0 To totalClientsInDB.Length - 1

            Dim usernameBytes() As Byte = ASCII.GetBytes(totalClientsInDB(i).username)

            Dim publicKeyBytes(31) As Byte

            If totalClientsInDB(i).publicKey <> "NULL" Then

                publicKeyBytes = System.Convert.FromBase64String(totalClientsInDB(i).publicKey)

                Dim message(client.ReceiveBufferSize - 1) As Byte

                usernameBytes.CopyTo(message, 0)

                publicKeyBytes.CopyTo(message, usernameLen + 1)

                'See if the username we have is equal to the ones already connected

                For c = 0 To connectedClientsValues.Length - 1

                    If connectedClientsValues(c).username = totalClientsInDB(i).username Then

                        message(usernameLen) = 1

                    Else

                        message(usernameLen) = 0

                    End If

                Next

                netStream.Write(message, 0, message.Length)

                netStream.Flush()

            End If

        Next

Here, like before we get all of the users in the lookuptable, which tells us what users are logged on. Using this information, the server can send a 1 or 0 on the end of the data, representing the connection status. First, we need to get the count of users in the database, in order to set the maximum index of the **databaseReturn** array, this allows us to the get the users from the server and thusly populate that array, without using a **Redim** statement. Using the value of the count of records in the database, we can also send that value back to the user letting them know how many different users they should expect. Once that is done we loop through all of the users in the **databaseReturn** array and send the username, the public key and the boolean value of connected, writing that to the network stream.

Upon sending this information, the user now knows everything it needs to be able to accept any messages or files that were sent while the user was offline, I created a new procedure to send this data, using the correct approach to using TCP (sending the metadata first followed by the data):

As an example, for messages:

|  |  |  |  |
| --- | --- | --- | --- |
| Message Identifier | Username | Length of Data | Data |
| Byte | 20 Bytes | 4 Bytes | Bytes |

And for files:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Message Identifier | Username | Length of Data | Filename | Data |
| Byte | 20 Bytes | 4 Bytes | 100 Bytes | Bytes |

Above shows the new designs of a sending a single message, without having to split it up into 65k chunks and my preference would be to completely rewrite the sending and receiving functions to incorporate this design, as it means that the amount of data that can be sent is almost completely unlimited.

When sending this data back to the connecting client, the server first retrieves the count of IMs and files to send and sends that data back to the client like so:

Private Sub sendUnsentData(ByRef stream As NetworkStream, ByVal username As String)

    Dim SQL As String

    'Get the number of messages to send

    SQL = "SELECT COUNT(MessageID) FROM TBL\_UNSENT\_MESSAGES WHERE [Recipient\_UserID] = '" & username & "';"

    Dim reader As OleDb.OleDbDataReader = databaseConnections.sendSQLQuery(SQL)

    reader.Read()

    Dim messageCount As Integer = reader.GetInt32(0)

    reader.Close()

    'Get the number of files to send

    SQL = "SELECT COUNT(FileID) FROM TBL\_UNSENT\_FILES WHERE [Recipient\_UserID] = '" & username & "';"

    reader = databaseConnections.sendSQLQuery(SQL)

    reader.Read()

    Dim fileCount As Integer = reader.GetInt32(0)

    reader.Close()

    'Send the number of seperate sets of data being sent over the network for recieve

    Dim packetCount() As Byte = BitConverter.GetBytes(messageCount + fileCount)

    stream.Write(packetCount, 0, packetCount.Length)

    stream.Flush()

Once that has been sent we can then send the IMs first followed by the files last:

'Send the IMs

        SQL = "SELECT MessageData, Sender\_UserID FROM TBL\_UNSENT\_MESSAGES WHERE [Recipient\_UserID] = '" & username & "';"

        reader = databaseConnections.sendSQLQuery(SQL)

        reader.Read()

        For i = 0 To messageCount - 1

            '| Message ID | Username | Lenght | Data ... |

            '| Byte       | 20 Bytes |4 Bytes | Bytes    |

            Dim senderUsername As String = reader.GetString(1)

            Dim senderUsernameBytes() As Byte = ASCII.GetBytes(senderUsername)

            Dim message() As Byte = System.Convert.FromBase64String(reader.GetString(0))

            Dim messageLen() As Byte = BitConverter.GetBytes(message.Length)

            Dim fullMessage(1 + 20 + 4 + message.Length - 1) As Byte

            fullMessage(0) = 1

            senderUsernameBytes.CopyTo(fullMessage, 1)

            messageLen.CopyTo(fullMessage, 21)

            message.CopyTo(fullMessage, 25)

            stream.Write(fullMessage, 0, fullMessage.Length)

            stream.Flush()

            reader.Read()

        Next

        reader.Close()

        SQL = "DELETE FROM TBL\_UNSENT\_MESSAGES WHERE [Recipient\_UserID] = '" & username & "';"

        databaseConnections.sendSQLNonQuery(SQL)

        'send the files

        SQL = "SELECT [Filepath], [Sender\_UserID] FROM TBL\_UNSENT\_FILES WHERE [Recipient\_UserID] = '" & username & "';"

        reader = databaseConnections.sendSQLQuery(SQL)

        reader.Read()

        For i = 0 To fileCount - 1

            '| Message ID | Username | Lenght | Filename | Data ... |

            '| Byte       | 20 Bytes |4 Bytes |100 Bytes | Bytes    |

            Dim senderUsername As String = reader.GetString(1)

            Dim senderUsernameBytes() As Byte = ASCII.GetBytes(senderUsername)

            Dim filepath As String = Path.GetFileName(reader.GetString(0))

            Dim filepathBytes() As Byte = ASCII.GetBytes(filepath)

            Dim fs As New FileStream(reader.GetString(0), FileMode.Open)

            Dim message(fs.Length - 1) As Byte

            fs.Read(message, 0, fs.Length)

            fs.Dispose()

            Dim messageLen() As Byte = BitConverter.GetBytes(message.Length)

            Dim fullMessage(1 + 20 + 4 + 100 + message.Length - 1) As Byte

            fullMessage(0) = 2

            senderUsernameBytes.CopyTo(fullMessage, 1)

            messageLen.CopyTo(fullMessage, 21)

            filepathBytes.CopyTo(fullMessage, 25)

            message.CopyTo(fullMessage, 125)

            stream.Write(fullMessage, 0, fullMessage.Length)

            stream.Flush()

            reader.Read()

        Next

        reader.Close()

        SQL = "DELETE FROM TBL\_UNSENT\_FILES WHERE [Recipient\_UserID] = '" & username & "';"

        databaseConnections.sendSQLNonQuery(SQL)

    End Sub

In order to make sure we send all of the unsent data, we loop through the count using a definite loop ensuring that we have sent everything. Once the loops are completed, we remove the records from the table, freeing up the table for when more data is to be added to the table when the user next goes offline. Sending the data each time consists of forming the array with the structure shown above and in the comments of each for loop.

With that done the public key of the user is then set in the database with the SQL:

SQL = "UPDATE TBL\_USER SET [Last\_Known\_PublicKey] = '" & System.Convert.ToBase64String(publicKey) & "' WHERE [Username] = '" & username & "';"

databaseConnections.sendSQLNonQuery(SQL)

With this, the connection of the user is finalised and the user is able to communicate with the server so long as connection terminated successfully.

##### Administrator functions

In order to make it easier for an administrator to manage server functions without direct access to the server, I added an interface to the client that allowed a logged-on administrator to manage users in the server’s database, this also required a new section to be added to the receive data function, in order to be able to deal with these requests. The method that the server takes to identify the different demands was to change the value of the message identifier (the first byte at the start of the message), to accommodate all of the enquiries the client makes we have a set of values that translates to the specific command:

|  |  |
| --- | --- |
| ID | Request |
| 202 | Adding a new user |
| 203 | Updating the password of a specific user |
| 204 | Setting a user’s right |
| 205 | Removing a user |
| 206 | Sync the client with the users in the database |

Every one of these requests are pretty secure, and ensure that the user who is trying to perform them has the needed privileges to perform them, otherwise they will be failed by the server and the user is notified.

Request 202 requires the client to provide not only the correct message identifier but also the Username of the user they want to add, the password hash of the new user and whether that user is an administrator, specified by a Boolean value. In order to know what message ID is being used I created a select case statement which takes the first value of the data received:

Select Case CInt(data(0))

            Case Is = 202

                '| Message ID | Username | Password Hash | Administrator |

                '| Byte       |20 Bytes  | 32 Bytes      | Byte          |

                'The decimal value 202 is equal to the hex value CA, used when adding a new user

                Dim usernameBytes(19) As Byte

                Dim passwordHash(31) As Byte

                Dim admin As Byte

                Dim SQL As String

                Dim reader As OleDb.OleDbDataReader

                Array.ConstrainedCopy(data, 1, usernameBytes, 0, 20)

                Array.ConstrainedCopy(data, 21, passwordHash, 0, 32)

                admin = data(53)

                Dim lastNonZeroIndex As Integer = getLastNonZeroIndex(usernameBytes)

                Dim nonZeroUsername(lastNonZeroIndex) As Byte

                Array.Copy(usernameBytes, nonZeroUsername, lastNonZeroIndex + 1)

                'Check if the username who wanted the new user to be created is an admin

                SQL = "SELECT [Is\_Administrator] FROM TBL\_USER WHERE [Username] = '" & lookupTable.GetClientUname(IPAddr) & "';"

                reader = databaseConnections.sendSQLQuery(SQL)

                reader.Read()

                If Not reader.GetBoolean(0) Then

                    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Tried to add a new user, but was not administrator" & vbNewLine)

                    client.sendError(MTClientReciever.sendingErrorEnum.NOT\_ADMIN)

                    Return

                End If

                reader.Close()

                'Check if the user already exists

                SQL = "SELECT [Username] FROM TBL\_USER WHERE [Username] = '" & ASCII.GetString(nonZeroUsername) & "';"

                reader = databaseConnections.sendSQLQuery(SQL)

                reader.Read()

                If reader.HasRows Then

                    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Tried to add a new user, but the username already exists" & vbNewLine)

                    client.sendError(MTClientReciever.sendingErrorEnum.USER\_EXISTS)

                    Return

                End If

                reader.Close()

                'Add the new user

                SQL = "INSERT INTO TBL\_USER (Username,Password\_Hash,Is\_Administrator,Last\_Known\_PublicKey) VALUES ('" & ASCII.GetString(nonZeroUsername) & "','" & System.Convert.ToBase64String(passwordHash) & "'," & CBool(admin).ToString & ",'NULL');"

                databaseConnections.sendSQLNonQuery(SQL)

                client.sendACK()

There are a few checks that must be applied to ensure that the new username does not already exist in the database and that the user adding the new user is an administrator, these requirements are checked with two SQL statements:

"SELECT [Is\_Administrator] FROM TBL\_USER WHERE [Username] = '" & lookupTable.GetClientUname(IPAddr) & "';"

"SELECT [Username] FROM TBL\_USER WHERE [Username] = '" & ASCII.GetString(nonZeroUsername) & "';"

After the data from the client has been successfully broken down into its constituent parts the first check ensures that the user who is logged onto the current connection is an administrator, failing that check causes the **sendError** to send back a value that corresponds to an enumeration saying that the client is not an administrator. The next audit certifies that the username that has been provided is unique in the database, meaning that it can be used a public key and not cause any errors with adding the new user to the table. Lastly if the checks succeed then the client is added to the database and the ACK value is sent to the client.

Next on the list is updating a user’s password, this should be able to be done by both the administrator and the user so therefore two different situations are needed. If the password is being changed by the user then the user specifies the old password hash and this is checked against the hash that is listen in the record in the database, failing this check or if the user doesn’t exist then the password change request is denied. The second situation is when an administrator wants to change the password, for example if a user can no longer login and doesn’t remember their latest password, by keeping the old password hash as null, a int32 conversion statement will convert the hash to an integer and if the value is equal to 0 then we check that the user requesting this change is listed as an administrator in the database:

Case Is = 203

    'Update password

    '| Message ID | Username | old Password Hash | new Password Hash |

    Dim usernameBytes(19) As Byte

    Dim oldPasswordHash(31) As Byte

    Dim newPasswordHash(31) As Byte

    Dim oldPasswordHashInt As Integer = -99

    Dim SQL As String

    Dim reader As OleDb.OleDbDataReader

    Array.ConstrainedCopy(data, 1, usernameBytes, 0, 20)

    Array.ConstrainedCopy(data, 21, oldPasswordHash, 0, 32)

    Array.ConstrainedCopy(data, 53, newPasswordHash, 0, 32)

    Try

        oldPasswordHashInt = BitConverter.ToInt32(oldPasswordHash, 0)

    Catch ex As Exception

    End Try

    'Get the username without zero bytes

    Dim lastNonZeroIndex As Integer = getLastNonZeroIndex(usernameBytes)

    Dim nonZeroUsername(lastNonZeroIndex) As Byte

    Array.Copy(usernameBytes, nonZeroUsername, lastNonZeroIndex + 1)

    If oldPasswordHashInt = -99 Then

        'Check if the users given password is correct

        SQL = "SELECT [Password\_Hash] FROM TBL\_USER WHERE [Username] = '" & ASCII.GetString(nonZeroUsername) & "';"

        reader = databaseConnections.sendSQLQuery(SQL)

        reader.Read()

        If Not reader.HasRows Then

            MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Tried to change user password but user did not exist" & vbNewLine)

            client.sendError(MTClientReciever.sendingErrorEnum.USER\_DOES\_NOT\_EXIST)

            Return

        End If

        Dim passwordHash() As Byte = System.Convert.FromBase64String(reader.GetString(0))

        For i = 0 To 31

            If passwordHash(i) <> oldPasswordHash(i) Then

                MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Tried to change password, but old password did not match new password " & vbNewLine)

                client.sendError(MTClientReciever.sendingErrorEnum.PASSWORD\_MISMATCH)

                Return

            End If

        Next

        reader.Close()

    Else

        'Administrator tried to change password, check if the person was admin

        SQL = "SELECT [Is\_Administrator] FROM TBL\_USER WHERE [Username] = '" & lookupTable.GetClientUname(IPAddr) & "';"

        reader = databaseConnections.sendSQLQuery(SQL)

        reader.Read()

        If Not reader.GetBoolean(0) Then

            MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Tried to change password, but was not administrator" & vbNewLine)

            client.sendError(MTClientReciever.sendingErrorEnum.NOT\_ADMIN)

            Return

        End If

        reader.Close()

    End If

    'Update the password

    SQL = "UPDATE TBL\_USER SET [Password\_Hash] = '" & System.Convert.ToBase64String(newPasswordHash) & "' WHERE [Username] = '" & ASCII.GetString(nonZeroUsername) & "';"

    databaseConnections.sendSQLNonQuery(SQL)

    client.sendACK()

As before once the checks succeed then the statement:

"UPDATE TBL\_USER SET [Password\_Hash] = '" & System.Convert.ToBase64String(newPasswordHash) & "' WHERE [Username] = '" & ASCII.GetString(nonZeroUsername) & "';"

Will updated the table with the new hash encoded with base 64.

I added ID 204 next which allows an administrator to change a user’s administrator status this just needed two checks to make sure that the user requesting the change was an administrator and that the user we want to perform the change on currently exits:

Case Is = 204

                'Set the users rights

                '| Message ID | Username | Admin |

                Dim usernameBytes(19) As Byte

                Dim admin As Byte

                Dim SQL As String

                Dim reader As OleDb.OleDbDataReader

                Array.ConstrainedCopy(data, 1, usernameBytes, 0, 20)

                admin = data(22)

                'Get the username without zero bytes

                Dim lastNonZeroIndex As Integer = getLastNonZeroIndex(usernameBytes)

                Dim nonZeroUsername(lastNonZeroIndex) As Byte

                Array.Copy(usernameBytes, nonZeroUsername, lastNonZeroIndex + 1)

                'Check if the username who wanted the user to be set admin, is admin

                SQL = "SELECT [Is\_Administrator] FROM TBL\_USER WHERE [Username] = '" & lookupTable.GetClientUname(IPAddr) & "';"

                reader = databaseConnections.sendSQLQuery(SQL)

                reader.Read()

                If Not reader.GetBoolean(0) Then

                    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Tried to add a new user, but was not administrator" & vbNewLine)

                    client.sendError(MTClientReciever.sendingErrorEnum.NOT\_ADMIN)

                    Return

                End If

                reader.Close()

                'Check the user exists

                SQL = "SELECT [Username] FROM TBL\_USER WHERE [Username] = '" & ASCII.GetString(nonZeroUsername) & "';"

                reader = databaseConnections.sendSQLQuery(SQL)

                reader.Read()

                If Not reader.HasRows Then

                    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Tried to change user password but user did not exist" & vbNewLine)

                    client.sendError(MTClientReciever.sendingErrorEnum.USER\_DOES\_NOT\_EXIST)

                    Return

                End If

                reader.Close()

                'Set the user as an admin

                SQL = "UPDATE TBL\_USER SET [Is\_Administrator] = " & CBool(admin) & " WHERE [Username] = '" & ASCII.GetString(nonZeroUsername) & "';"

                databaseConnections.sendSQLNonQuery(SQL)

                client.sendACK()

Every time a test fails the corresponding error message is sent back to the user notifying that it has failed, giving a specific error message for each different failure.

As one of the last requirements I added the checks to remove a user, the same checks as needed for changing a user’s rights:

Case Is = 205

    'Removing a user

    '| Message ID | Username |

    '| Byte       |20 Bytes  |

    Dim usernameBytes(19) As Byte

    Dim SQL As String

    Dim reader As OleDb.OleDbDataReader

    Array.ConstrainedCopy(data, 1, usernameBytes, 0, 20)

    Dim lastNonZeroIndex As Integer = getLastNonZeroIndex(usernameBytes)

    Dim nonZeroUsername(lastNonZeroIndex) As Byte

    Array.Copy(usernameBytes, nonZeroUsername, lastNonZeroIndex + 1)

    'Check if the username who wanted the new user to be removed is an admin

    SQL = "SELECT [Is\_Administrator] FROM TBL\_USER WHERE [Username] = '" & lookupTable.GetClientUname(IPAddr) & "';"

    reader = databaseConnections.sendSQLQuery(SQL)

    reader.Read()

    If Not reader.GetBoolean(0) Then

        MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Tried to remove a user, but was not administrator" & vbNewLine)

        client.sendError(MTClientReciever.sendingErrorEnum.NOT\_ADMIN)

        Return

    End If

    reader.Close()

    'Check the user exists

    SQL = "SELECT [Username] FROM TBL\_USER WHERE [Username] = '" & ASCII.GetString(nonZeroUsername) & "';"

    reader = databaseConnections.sendSQLQuery(SQL)

    reader.Read()

    If Not reader.HasRows Then

        MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Tried to change user password but user did not exist" & vbNewLine)

        client.sendError(MTClientReciever.sendingErrorEnum.USER\_DOES\_NOT\_EXIST)

        Return

    End If

    reader.Close()

    'Add the new user

    SQL = "DELETE FROM TBL\_USER WHERE [Username] = '" & ASCII.GetString(nonZeroUsername) & "';"

    databaseConnections.sendSQLNonQuery(SQL)

    client.sendACK()

Lastly, I added the code necessary for syncing the value from the database with the client, this is slightly different to the previous procedures, as it requires that values be sent back to the user from the database:

Case Is = 206

                'Send back all of the users in the database and whether they are an administrator

                Dim connectedClientsValues() As ClientLookupTable.tableValues = lookupTable.GetConnectedClients()

                Dim totalClientsReader As OleDb.OleDbDataReader = databaseConnections.sendSQLQuery("SELECT COUNT(Username) FROM TBL\_USER;")

                totalClientsReader.Read()

                Dim totalClientCount As Integer = totalClientsReader.GetInt32(0)

                totalClientsReader.Close()

                Dim totalClientsInDB(totalClientCount - 1) As database206Return

                If totalClientCount <> 0 Then

                    totalClientsReader = databaseConnections.sendSQLQuery("SELECT [Username], [Is\_Administrator] FROM TBL\_USER;")

                    For i = 0 To totalClientCount - 1

                        totalClientsReader.Read()

                        totalClientsInDB(i).username = totalClientsReader.GetString(0)

                        totalClientsInDB(i).isAdministrator = totalClientsReader.GetBoolean(1)

                    Next

                End If

                totalClientsReader.Close()

                client.sendACK()

                'send the count of packets that we will send, and transmit that first

                Const usernameLen As Integer = 20

                'send:

                '| MessageID | username | isAdmin |

                For i = 0 To totalClientsInDB.Length - 1

                    Dim usernameBytes() As Byte = ASCII.GetBytes(totalClientsInDB(i).username)

                    Dim message(21) As Byte

                    message(0) = 206

                    usernameBytes.CopyTo(message, 1)

                    message(usernameLen + 1) = CByte(totalClientsInDB(i).isAdministrator)

                    client.sendData(message)

                Next

##### Saving messages for sending later

As a prerequisite the server must be able to store messages for user that are offline, earlier I demonstrated how these messages were sent to the user when the user connects. In order to be able to send them the messages or files must firstly be saved and a record added to the database, informing the server upon using a query who the message was for and who sent it, so that the server can reconstruct the message and send it to the correct user. The way we do that is we use the client hash table to determine if a user is online, if that user could not be found then the message can be stored for later, in the event that the message is a file, we take the first packet which is where x=0 and open a file stream with the filename provided by the user to a temporary directory, this filestream is then accessed when another packet is received and the data for the file can be written out to it. With both IMs and files, the whole set of data must be received before we can write anything out to the database to ensure the transfer continued successfully, when receiving IMs, we re-dimension an array to the size we need so that we can store all of the data in it then write that array out to database.

Firstly, the code below demonstrates creating the filestream when the server receives a file offline:

Dim recp As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(nonZeroUsername))

If recp = Nothing Then

    'Got to now store the message for later, that user is not currently logged on

    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recp & vbNewLine)

    If Not Directory.Exists(Environment.CurrentDirectory & "/TEMP/" & ASCII.GetString(nonZeroUsername)) Then

        Try

            Directory.CreateDirectory(Environment.CurrentDirectory & "/TEMP/" & ASCII.GetString(nonZeroUsername))

        Catch sysex As System.ArgumentException

            MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & sysex.Message & vbNewLine)

            client.sendError()

            Return

        End Try

    End If

    'Open the filestream for the next bytes we are ready to recieve

    Try

        FileStream = New FileStream(Environment.CurrentDirectory & "\TEMP\" & ASCII.GetString(nonZeroUsername) & "\" & ASCII.GetString(fileName), FileMode.Create)

    Catch ex As Exception

    MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & ex.Message & vbNewLine)

    End Try

    client.sendACK()

    Return

End If

This section checks to see if the temporary directory exists and makes sure there is a folder in that directory that corresponds to the user’s name, if these don’t exist then they are automatically created, we then try and open a new file for writing with filestream and then return. The filestream is a global variable as it will need to be accessed multiple times when the user sends a large file.

Lastly is the section that collects the data together and checks if the packet numbers are equal, indicating that the last packet has been received by the server, and therefore either saves the file stream and writes out to the database or takes the total instant message and writes it out to the database:

Dim recpIPAddr As String = lookupTable.ResolveUsernameToIP(ASCII.GetString(nonZeroUsername))

If recpIPAddr = Nothing Then

    'Got to now store the message for later, that user is not currently logged on

    If messageType = 2 Then

        MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Writing encrypted file to /TEMP/ for user " & recpIPAddr & vbNewLine)

        FileStream.Write(encryptedData, 0, encryptedData.Length)

        If x = y Then

            Dim SQL As String

            SQL = "INSERT INTO TBL\_UNSENT\_FILES ([Filepath],[Sender\_UserID],[Recipient\_UserID]) VALUES ('" & FileStream.Name & "','" & lookupTable.GetClientUname(IPAddr) & "','" & ASCII.GetString(nonZeroUsername) & "');"

            databaseConnections.sendSQLNonQuery(SQL)

            FileStream.Close()

            FileStream = Nothing

            MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Finished writing the file out" & vbNewLine)

        End If

    ElseIf messageType = 1 Then

        MTMainThreadWriter.writeLog("[Client: " & clNumber & "]: " & "Storing message for later for the user " & recpIPAddr & vbNewLine)

        If x = 1 And y <> 1 Then

            ReDim totalMessage((data.Length \* y) - 1)

        ElseIf x = 1 And y = 1 Then

            ReDim totalMessage(encryptedData.Length - 1)

        End If

        If y > 1 Then

            encryptedData.CopyTo(totalMessage, data.Length \* (x - 1))

        Else

            encryptedData.CopyTo(totalMessage, 0)

        End If

        If x = y Then

            Dim SQL As String

            SQL = "INSERT INTO TBL\_UNSENT\_MESSAGES ([MessageData],[Sender\_UserID],[Recipient\_UserID]) VALUES ('" & System.Convert.ToBase64String(totalMessage) & "','" & lookupTable.GetClientUname(IPAddr) & "','" & ASCII.GetString(nonZeroUsername) & "');"

            databaseConnections.sendSQLNonQuery(SQL)

        End If

    End If

    Return

End If

Here, like before we check if the user we want is currently online, if that returns nothing then we do perform different actions based on what message was sent, with a file being sent (Message ID = 2) we write the data we received out to the file stream, when the last packet is received we then take the filename from the file stream and write out the values of filename, sender’s username and recipient’s username to the database as a new record, we can then close the file stream and thusly save the file and lastly reset the file stream. If the message type is equal to a 1 and it’s the first packet we have received then the array **totalMessage** is re-dimensioned as this holds all of the data for the current message we a receiving, the data is then written out to that array and upon the final packet being collected, the data in **totalMessage** can be written out to a new record in the **TBL\_UNSENT\_MESSAGES** table. With that being done the messages are saved now until the user connects again at another point in time.

#### Removing Exit Fors

In order to prototype that the server was working correctly, where I removed the zeros from the ends of the array I would have a for loop running from the last index until it reached an index without a zero, at that point the Exit for would stop the loop. In order to remove that I created a new subroutine that uses a do loop to get the last index and also allows me to easily specify a postion as well to check for when removing zeros, making sure the array gets no lower than that position:

Private Function getLastNonZeroIndex(ByVal byteArray() As Byte, Optional ByVal lastAllowedIndex As Integer = 0) As Integer

        Dim lastNonZeroIndex As Integer = byteArray.Length - 1

        Do

            lastNonZeroIndex -= 1

        Loop Until byteArray(lastNonZeroIndex) Or lastNonZeroIndex = lastAllowedIndex

        Return lastNonZeroIndex

    End Function

I also used this method in my client too to get the last index in the array that is not a zero.

### Designs of the Client

#### Backgrounds

The design of the TCP client was to provide the user with access to the encryption routines, message sending procedures and the overall design of the UI to be able to communicate with the server, and therefore other users. Similar to the server the client is multithreaded, providing a single thread for receiving data and another thread for sending data to the server. After having written the server code it made it much easier to write the routines needed for the client to connect and communicate with the server. Like the server the client copies some of the design considerations taken from the server, such as a system to update the main UI when accessing it from another thread, and uses the same Database Tools file as the server does.

#### The UIs

##### Login form

As an introduction to the client side, when launching the executable, the user is presented with a login form shown below:

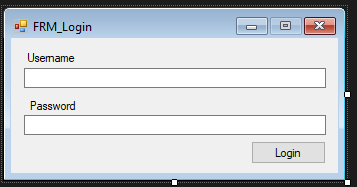


Figure 28: Login Form

This form gathers the required data from the user in order to be able to authenticate itself with the server. The way it does is a simple section of code that runs when the Login button is pressed and the form is submitted:

Imports System.Security.Cryptography

Imports System.Text.Encoding

Public Class FRM\_Login

    Private Sub CMD\_SubmitLoginCreds\_Click(sender As Object, e As EventArgs) Handles CMD\_SubmitLoginCreds.Click

        Dim sha256 As New SHA256Managed

        If TXT\_Username.Text = Nothing Or TXT\_Password.Text = Nothing Then

            MsgBox("Username or password field empty", MsgBoxStyle.Critical)

            Exit Sub

        End If

        If Not FRM\_Messages.SetupUserAfterLogin(sha256.ComputeHash(ASCII.GetBytes(TXT\_Password.Text)), TXT\_Username.Text) Then

            Return

        End If

        TXT\_Username.Clear()

        TXT\_Password.Clear()

        Me.Close()

    End Sub

End Class

Simply this checks to make sure that neither of the boxes are empty and then calls some code on the main form that allows use to start the socket, authenticate with the server and thusly allow the user to communicate with other clients connected to the server.

##### User Settings Form

This form allows an authenticated administrator access to the database, on the serve side his credentials are checked and its virtually impossible for a malicious user to be able to impersonate the administrator without the admins credentials. The form looks something like this:

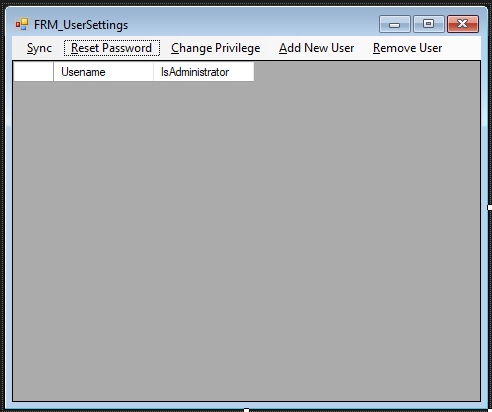


Figure 29:User settings form

This form consists of a dataGridView where the user will be listed upon opening the form or clicking the Sync button in the menu strip at the top of the form. In the box it will list all of the username’s and a check box indicating the privilege of that corresponding user. Along the top in the same menu strip as Sync allow an administrator to change a user’s details by clicking any of these buttons, upon clicking them so long as a user is selected a prompt box should appear asking the administrator to enter some details that the server needs to know. Here is an example of the form when it is synced:

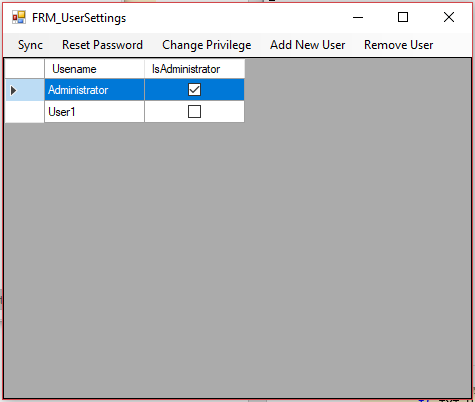


Figure 30: Running UI

As can be noted, two users currently exist in the database on the server, the default Administrator and a normal user called User1, which was created using the Add New User Button.

When any of the toolstrip items are clicked, it invokes the messages that are described on pages 120 to 127. When the form is opened it is invoked by using:

If FormUserSettings Is Nothing Then

    FormUserSettings = New FRM\_UserSettings(srFunction)

    FormUserSettings.ShowDialog()

End If

This means that we require a sub new in the form, instead of using **<form\_name>.show**, this allows us to then pass parameters to the form upon loading:

Public Sub New(ByRef srFunction As SendRecieve)

    ' This call is required by the designer.

    InitializeComponent()

    ' Add any initialization after the InitializeComponent() call.

    Me.srFunction = srFunction

    srFunction.snycWithServer()

End Sub

Here we use the parameter to set an instance of srFunction as a private global variable for that class, thus enabling us to send the requests mention earlier.

Private Sub SyncToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles SyncToolStripMenuItem.Click

        DGV\_Users.Rows.Clear()

        srFunction.snycWithServer()

    End Sub

    Public Sub getUserDetails(ByVal data() As Byte)

        numberOfUsers = BitConverter.ToInt32(data, 1)

        Dim usernameBytes(19) As Byte

        Dim username As String

        Dim isAdmin As Boolean = CBool(data(21))

        Array.ConstrainedCopy(data, 1, usernameBytes, 0, 20)

        Dim usernameLastNonZero As Integer = srFunction.getLastNonZeroIndex(usernameBytes)

        Dim tmpUsername(usernameLastNonZero) As Byte

        Array.Copy(usernameBytes, tmpUsername, usernameLastNonZero + 1)

        username = ASCII.GetString(tmpUsername)

        addRowsToDGV(New Object() {username, isAdmin})

    End Sub

    Private Sub addRowsToDGV(ByVal items As Object())

        If DGV\_Users.InvokeRequired Then

            Dim Dgate As New delegateAddRows(AddressOf addRowsToDGV)

            DGV\_Users.BeginInvoke(Dgate, New Object() {items})

        Else

            DGV\_Users.Rows.Add(items)

        End If

    End Sub

    Private Sub ResetPasswordToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles ResetPasswordToolStripMenuItem.Click

        Dim sha256 As New SHA256Managed

        Dim selectedRows As DataGridViewSelectedRowCollection = DGV\_Users.SelectedRows

        If selectedRows.Count = 0 Then

            MsgBox("Select a user from the box below", MsgBoxStyle.Critical)

            Return

        End If

        Dim username As String = selectedRows.Item(0).Cells.Item(0).Value

        Dim password As String = Interaction.InputBox("Provide a password for the user, its possible for a user to change the password later", "Set password")

        If password = Nothing Then

            Return

        End If

        Dim passwordHash() As Byte = SHA256.ComputeHash(ASCII.GetBytes(password))

        Dim oldPasswordHash(31) As Byte

        password = ""

        srFunction.updatePasswordHash(username, oldPasswordHash, passwordHash)

    End Sub

    Private Sub ChangePrivilegeToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles ChangePrivilegeToolStripMenuItem.Click

        Dim selectedRows As DataGridViewSelectedRowCollection = DGV\_Users.SelectedRows

        If selectedRows.Count = 0 Then

            MsgBox("Select a user from the box below", MsgBoxStyle.Critical)

            Return

        End If

        Dim username As String = selectedRows.Item(0).Cells.Item(0).Value

        Dim value As String = Interaction.InputBox("Type 1 for Administrator and 0 for normal user", "Set privilege")

        If value = Nothing Then

            Return

        End If

        If Not IsNumeric(value) AndAlso (CInt(value) <> 0 Or CInt(value) <> 1) Then

            Do

                value = Interaction.InputBox("Type 1 for Administrator and 0 for normal user", "Set privilege")

            Loop Until IsNumeric(value) AndAlso (CInt(value) = 0 Or CInt(value) = 1)

        End If

        srFunction.updateUsersRights(username, CBool(value))

    End Sub

    Private Sub AddNewUserToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles AddNewUserToolStripMenuItem.Click

        Dim username As String = Interaction.InputBox("Provide a username less than 20 characters long", "Set username")

        Dim sha256 As New SHA256Managed

        If username = Nothing Then

            Return

        End If

        If username.Length > 20 Then

            Do

                username = Interaction.InputBox("Provide a username less than 20 characters long", "Set username")

            Loop Until username.Length <= 20

        End If

        Dim password As String = Interaction.InputBox("Provide a password for the user, its possible for a user to change the password later", "Set password")

        If password = Nothing Then

            Return

        End If

        Dim passwordHash() As Byte = sha256.ComputeHash(ASCII.GetBytes(password))

        password = ""

        Dim value As String = Interaction.InputBox("Type 1 for Administrator and 0 for normal user", "Set privilege")

        If value = Nothing Then

            Return

        End If

        If value = Nothing AndAlso Not IsNumeric(value) AndAlso (CInt(value) <> 0 Or CInt(value) <> 1) Then

            Do

                value = Interaction.InputBox("Type 1 for Administrator and 0 for normal user", "Set privilege")

            Loop Until value <> Nothing AndAlso IsNumeric(value) AndAlso (CInt(value) = 0 Or CInt(value) = 1)

        End If

        srFunction.sendNewUser(username, passwordHash, CBool(value))

    End Sub

    Private Sub RemoveUserToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles RemoveUserToolStripMenuItem.Click

        Dim selectedRows As DataGridViewSelectedRowCollection = DGV\_Users.SelectedRows

        If selectedRows.Count = 0 Then

            MsgBox("Select a user from the box below", MsgBoxStyle.Critical)

            Return

        End If

        Dim username As String = selectedRows.Item(0).Cells.Item(0).Value

        srFunction.removeUser(username)

    End Sub

The subroutines above are the code that allows the menu strip items to function correctly, and it mainly consists of error checking the prompt box known as the **Interactio.InputBox**, trying to catch out any rouge values that could be submitted by a user. The sending of the data is covered in the send receive function that we have on page 148.

##### The Files Form

This form exits to show the user what files they were sent by the other user when connecting or when having an active conversation with them. It consists of a list view that shows the user a green tick or a red cross indicating if the file has been deleted from its know directory followed by the name of the file:

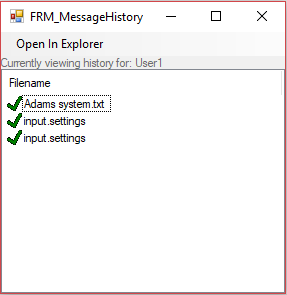


Figure 31: Form user files

As you can see some example files were sent in order to test the system and in the directory <My Documents>\IM Application\User1\ exists the files that are listed above. In the menu strip there is a button allowing a user to open that directory in explorer, this spawns a new explorer process with an argument telling where to point its directory at. The code behind this form is:

Imports System.IO

Public Class FRM\_MessageHistory

    Private imageList As ImageList

    Private username As String

    Public Sub New(ByRef imageList As ImageList, ByVal username As String, ByRef databaseConnection As DatabaseTools)

        ' This call is required by the designer.

        InitializeComponent()

        ' Add any initialization after the InitializeComponent() call.

        LSV\_Files.SmallImageList = imageList

        TXT\_UsersHistory.Text = "Currently viewing history for: " & username

        Me.imageList = imageList

        Me.username = username

        Dim SQL As String

        SQL = "SELECT COUNT([Filename]) FROM TBL\_FILES WHERE [Sender\_UserID] = '" & username & "';"

        Dim reader As OleDb.OleDbDataReader = databaseConnection.sendSQLQuery(SQL)

        reader.Read()

        Dim fileCount As Integer = reader.GetInt32(0)

        If fileCount = 0 Then

            LSV\_Files.Items.Add("No Files")

        Else

            SQL = "SELECT [Filename] FROM TBL\_FILES WHERE [Sender\_UserID] = '" & username & "';"

            reader = databaseConnection.sendSQLQuery(SQL)

            reader.Read()

            For i = 0 To fileCount - 1

                Dim filename As String = reader.GetString(0)

                Dim lvi As New ListViewItem

                lvi.Text = filename

                If File.Exists(FileIO.SpecialDirectories.MyDocuments & "/IM Application/" & username & "/" & filename) Then

                    lvi.ImageIndex = 1

                Else

                    lvi.ImageIndex = 0

                End If

                LSV\_Files.Items.Add(lvi)

                reader.Read()

            Next

        End If

    End Sub

    Private Sub OpenInExplorerToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles OpenInExplorerToolStripMenuItem.Click

        If Not Directory.Exists(FileIO.SpecialDirectories.MyDocuments & "/IM Application/" & username & "/") Then

            Directory.CreateDirectory(FileIO.SpecialDirectories.MyDocuments & "/IM Application/" & username & "/")

        End If

        Process.Start(ControlChars.Quote & FileIO.SpecialDirectories.MyDocuments & "/IM Application/" & username & "/" & ControlChars.Quote)

    End Sub

End Class

In sub **New** we set the text just above the list view to tell the user what user they are currently looking at, we also set the image list that allows us to use the green tick and the red tick icons to indicate if a file exists. After doing this we gather the names of all of the sent files from the database and check that they exist and thusly set the image accordingly form our image list. If there are no files listed in the database then we set the text of the list view to be equal to no files. When we spawn a new windows explorer window, we use process.start and pass it the argument of (“<My Documents>\IM Application\<Current User>”), this will either open the window in that directory or if it doesn’t exist then we create it first and then it will open in the correct directory.

##### Main UI

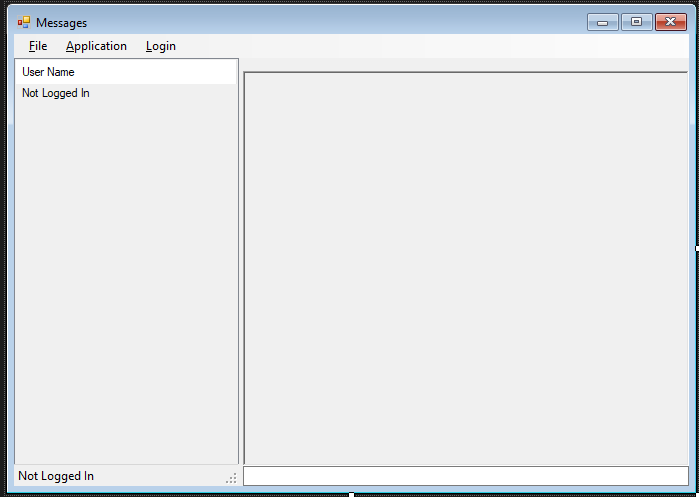
This is the main form that the user will be able to access all of the other forms from, this is also the form that will show all of the connected users, and the message pane allowing them to chat with that user, loading the messages from last time and allow them to view the received file history of that user:  


Figure 32:The main form

When running the users appear in the list view on the side and the messages will appear on the right-hand side allowing the user to use the text box at the bottom of the right hand to side to type their messages. When the form is logged in the design is changes slightly:

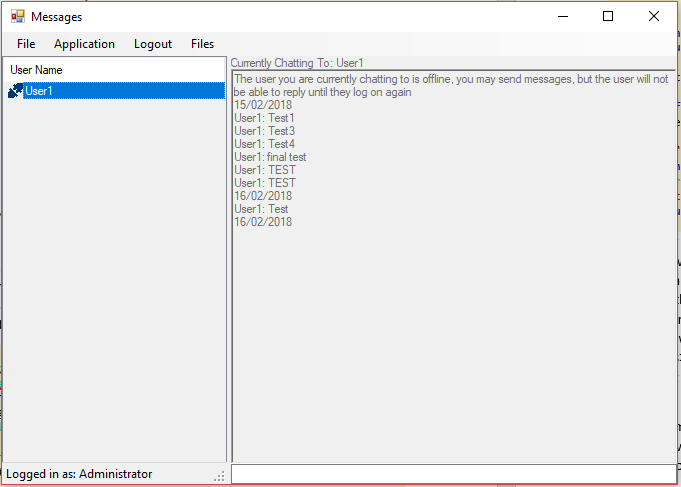


Figure 33:The running main UI

As you can see when a user has been authenticated with the server the user’s username appears in the bottom left and the user table is also populated with any of the users from the server’s database, any connected users will have a different icon indicating a connection which can be seen below:

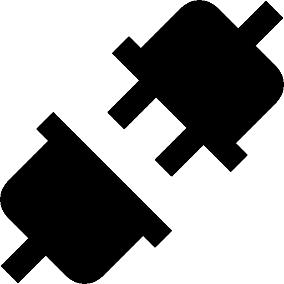
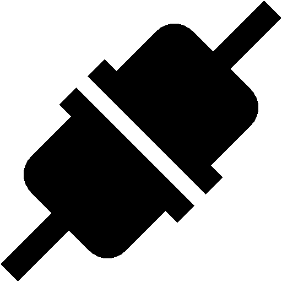


Figure 34:Connected left and Disconnected right

Also, when a user is selected that is currently offline a message is appended to the top of the rich text box stating that “The user you are currently chatting to is offline, you may send messages, but the user will not be able to reply until they log on again”. In order to send files and view the file history the menu strip button Files will be enabled and by having a user currently selected they will get a choice like so:

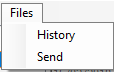


Figure 35: File Choice

By clicking the history button, they will be taken to the file history form showed above and by selecting the send option a **openFileDialog** will appear.

The code behind the form when it loads sets up some necessary parameters:

Private Sub FRM\_Messages\_Load(sender As Object, e As EventArgs) Handles MyBase.Load

    RTXT\_MessagePanel.Enabled = False

    RTXT\_MessagePanel.ForeColor = Color.Black

    LSV\_UserContacts.FullRowSelect = True

    Dim IL As New ImageList

    IL.ImageSize = New Size(15, 15)

    IL.Images.Add(GetEmbeddedIcon("offline.ico"))

    IL.Images.Add(GetEmbeddedIcon("online.ico"))

    LSV\_UserContacts.SmallImageList = IL

    FRM\_Login.Show()

End Sub

Simply it stets the rich text box to disabled and makes sure that the font colour is set to black, it creates a new image list using the two icons above to indicate if the user is online or offline and sets that image list inside of the list views image list parameter, it then finalises by showing the login form.

Upon the form closing it makes sure to save the clients settings file and closes the connection to the server if not done so already:

Private Sub FRM\_Messages\_Closing(sender As Object, e As EventArgs) Handles MyBase.Closing

    If Not srFunction Is Nothing Then

        srFunction.closeConnection()

    End If

    My.Settings.Default.Save()

End Sub

When the user submits a new message to be sent the code below is called:

Private Sub TXT\_MessageEnter\_KeyPress(sender As Object, e As KeyEventArgs) Handles TXT\_MessageEnter.KeyDown

    If e.KeyCode = Keys.Enter Then

        e.SuppressKeyPress = True

        If localUsername = Nothing Or remoteUser = Nothing Then

            MsgBox("You are either currently not logged in or you have not selected a user to chat to.", MsgBoxStyle.Critical)

            Return

        End If

        Try

            srFunction.sendMessage(remoteUser, TXT\_MessageEnter.Text, localUsername)

            WriteMsgToRTX(TXT\_MessageEnter.Text, localUsername, True)

        Catch ex As ObjectDisposedException

            srFunction = Nothing

            clientSocket.Close()

            clientSocket = New TcpClient

        End Try

        TXT\_MessageEnter.Clear()

    End If

End Sub

Which thusly uses this to add it to the rich text box:

Private Sub WriteMsgToRTX(sMessage As String, sUsername As String, bIslocalUsername As Boolean, ByVal Optional file As Boolean = False)

    Dim fBold As New Font(RTXT\_MessagePanel.Font.FontFamily, RTXT\_MessagePanel.Font.Size, FontStyle.Bold)

    Dim sOutString As String

    If Not file Then

        sOutString = sUsername & ": " & sMessage

        RTXT\_MessagePanel.AppendText(sOutString)

        If bIslocalUsername Then

            RTXT\_MessagePanel.Select(RTXT\_MessagePanel.TextLength - sOutString.Length, sOutString.Length)

            RTXT\_MessagePanel.SelectionFont = fBold

        End If

    Else

        RTXT\_MessagePanel.AppendText(sMessage)

        RTXT\_MessagePanel.Select(RTXT\_MessagePanel.TextLength - sMessage.Length, sMessage.Length)

        RTXT\_MessagePanel.SelectionFont = fBold

    End If

    RTXT\_MessagePanel.AppendText(vbNewLine)

End Sub

Here the **TXT\_MessageEnter\_KeyPressed** sub checks that the key that was pressed was the enter key, that being so we block the keypress from being outputted to the text box and we check that the two usernames, ours (local) and the person we are chatting to (remote) have been set with values. We then use the function **WriteMsgToRTX** to write the message to the rich text box, if the condition **bIsLocalUsername** is true then we highlight the text we right out with a bold font otherwise its written out normally, and if the Boolean **file** value is set to true then we write out a different string to what we use for IMs.

Once the text has been written out we try and send the message to the server and if that fails the function encapsulated in the try catch will catch the exception and thusly clear the instance of **srFunction** and reset the instance of **TcpClient**, if the instance of **TcpClient** is not reset then we have to close the program to be able to try reconnecting with the server, as it will return an error indicating that a request has already been made on the socket.

When a user receives a file or a message without that user being selected in the list view then we set the user’s background to a different colour to indicate they have received some data. We use an orange hue for IMs and a light blue hue for any files received. In order to set this colour, the public subroutine **setLSVBackColorOrange** is called:

Public Sub setLSVBackColorOrange(ByVal username As String, ByVal color As Color)

    Dim i As Integer

    Do

        If LSV\_UserContacts.Items.Item(i).Text = username Then

            LSV\_UserContacts.Items.Item(i).BackColor = color

        End If

        i += 1

    Loop Until i > LSV\_UserContacts.Items.Count - 1

End Sub

This loops through the users in the list view box and when the username is equal to the one we need to set then the colour of the background equal to the colour passed in the parameters.

In order to change the rich text box based on what users is selected and therefore load the messages associated with that user from previous conversations we use the event **SelectedIndexChanged** in order to get the selected user:

Private Sub LSV\_UserContacts\_SelectedIndexChanged(sender As Object, e As EventArgs) Handles LSV\_UserContacts.SelectedIndexChanged

        Try

            remoteUser = LSV\_UserContacts.SelectedItems(0).Text

        Catch ex As Exception

            errorHappened(LSV\_UserContacts, "Make sure you select a user", errorEnum.NULL\_REFERECE\_EXCEPTION, GetCurrentMethod.Name)

            Return

        End Try

        LSV\_UserContacts.SelectedItems(0).BackColor = System.Drawing.SystemColors.ButtonFace

        TXT\_ChatTo.Text = "Currently Chatting To: " + LSV\_UserContacts.SelectedItems(0).Text

        RTXT\_MessagePanel.Clear()

        If LSV\_UserContacts.SelectedItems(0).ImageIndex = 0 Then

            RTXT\_MessagePanel.AppendText("The user you are currently chatting to is offline, you may send messages, but the user will not be able to reply until they log on again" & vbNewLine)

        End If

        Dim SQL As String

        SQL = "SELECT COUNT([MessageData]) FROM TBL\_MESSAGES WHERE IsFile = False AND Sender\_UserID = '" & remoteUser & "';"

        Dim reader As OleDb.OleDbDataReader = srFunction.databaseConnection.sendSQLQuery(SQL)

        reader.Read()

        Dim messageCount As Integer = reader.GetInt32(0)

        If messageCount > 0 Then

            Dim dbValues(messageCount - 1) As databaseMessageReturn

            SQL = "SELECT [MessageData], [ReceivedDate], [Sender\_UserID] FROM TBL\_MESSAGES WHERE IsFile = False AND Sender\_UserID = '" & remoteUser & "';"

            reader = srFunction.databaseConnection.sendSQLQuery(SQL)

            For i = 0 To messageCount - 1

                reader.Read()

                dbValues(i).MessageData = reader.GetString(0)

                dbValues(i).ReceivedDate = DateTime.Parse(reader.GetString(1))

                dbValues(i).username = reader.GetString(2)

            Next

            For i = 0 To messageCount - 1

                Dim difference As TimeSpan = DateTime.Today.Date - dbValues(i).ReceivedDate

                Dim lastTotalDays As Double

                'check that years and months are equal

                If difference.TotalDays <= 2 Then

                    If i = 0 Then

                        lastTotalDays = difference.TotalDays

                        RTXT\_MessagePanel.AppendText(dbValues(i).ReceivedDate.ToShortDateString & vbNewLine)

                    ElseIf lastTotalDays <> difference.TotalDays Then

                        RTXT\_MessagePanel.AppendText(dbValues(i).ReceivedDate.ToShortDateString & vbNewLine)

                    End If

                    WriteMsgToRTX(dbValues(i).MessageData, dbValues(i).username, False)

                End If

            Next

        End If

        RTXT\_MessagePanel.AppendText(DateTime.Today.Date.ToShortDateString & vbNewLine)

        'add in here the editing of the box when we get details from the server

    End Sub

Here we firstly try setting the remote user as the selected index, if no indexes were selected then an error is produced notifying the user allowing them to correct their mistake. If this succeeds then we set the text above the rich text box telling the user who they are in communication with, we also reset the rich text box and add the default text for an offline user if the selected user is offline. Next, we use the database connection to get the number of messages in the database from both users, allowing us to set an array upper limit with that count of users, the array uses a user defined structure:

Private Structure databaseMessageReturn

    Public MessageData As String

    Public ReceivedDate As DateTime

    Public username As String

End Structure

This structure allows to get the messages data the received date and the username of the sender, which currently is the same for messages sent by the local user and received, this will need to be updated before production release and the database will need to be update too, but as an example of how it works, it is sufficient. Having retrieved the users from the database we can then run through the messages we have received, we only load the messages from the last two days anything before that is discarded, that check is performed in the **If** statement inside the second **for** loop, we also write out the date of messages when the received date of those messages changes, allowing the user to tell what date the messages were received on. Once the message has been verified to be within the last two days the message is added to the rich text box. Lastly the current date is added to the rich text box to signify the start of a new chat session.

When the login button on the menu strip is available it opens the login form with:

Private Sub LoginToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles LoginToolStripMenuItem.Click

    FRM\_Login.Show()

End Sub

And when the logout button is available then much like when the form closes it will disconnect the client and reset instances:

Private Sub LogoutToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles LogoutToolStripMenuItem.Click

        If localUsername = Nothing Then

            MsgBox("Already logged out", MsgBoxStyle.Critical)

            Exit Sub

        End If

        If Not srFunction Is Nothing Then

            srFunction.closeConnection()

            srFunction = Nothing

        End If

        TLS\_Username.Text = "Not Logged In"

        TXT\_ChatTo.Clear()

        RTXT\_MessagePanel.Clear()

        LSV\_UserContacts.Items.Clear()

        LSV\_UserContacts.Items.Add("Not Logged In")

        localUsername = Nothing

        remoteUser = Nothing

        LoginToolStripMenuItem.Visible = True

        LogoutToolStripMenuItem.Visible = False

        FilesToolStripMenuItem.Visible = False

    End Sub

The next three functions are used in conjunction with the form that allows an administrator to change user’s identifications:

Private Sub UserSettingsToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles UserSettingsToolStripMenuItem.Click

    If FormUserSettings Is Nothing Then

        FormUserSettings = New FRM\_UserSettings(srFunction)

        FormUserSettings.ShowDialog()

    End If

End Sub

Private Sub FormUserSettings\_Closing(sender As Object, e As EventArgs) Handles FormUserSettings.Closed

    FormUserSettings = Nothing

End Sub

Private Sub updateFormUserSettings(ByVal data() As Byte) Handles srFunction.recievedPopulateValues

    FormUserSettings.getUserDetails(data)

End Sub

The first subroutine opens the from when clicked, the second resets the instance to nothing when the form is closed and the third is used to update the form when values are received from the server when the Sync button is used.

When the file history is opened then the code below is run:

Private Sub HistoryToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles HistoryToolStripMenuItem.Click

        Dim imageList As New ImageList

        imageList.ImageSize = New Size(15, 15)

        imageList.Images.Add(GetEmbeddedIcon("no.ico"))

        imageList.Images.Add(GetEmbeddedIcon("yes.ico"))

        If remoteUser = Nothing Then

            MsgBox("Select a user before preceding")

            Return

        End If

        Dim history As New FRM\_MessageHistory(imageList, remoteUser, srFunction.databaseConnection)

        history.ShowDialog()

    End Sub

This simply sets up the image list using the GetEmbeddedIcon function that the server used to get the resources that are compiled within the executable. This image list, the currently selected username and the database connection is sent to the form, enabling the form to populate its list view with data.

When the user sends a file to the receiving user the Send button under the Files menu on the toolstrip is used: Private Sub SendToolStripMenuItem\_Click(sender As Object, e As EventArgs) Handles SendToolStripMenuItem.Click

        If remoteUser = Nothing Then

            MsgBox("Select a user before preceding")

            Return

        End If

        OpenFileDialog1.InitialDirectory = System.Environment.SpecialFolder.MyDocuments

        OpenFileDialog1.ShowDialog()

        Dim filepath As String = OpenFileDialog1.FileName

        If filepath = Nothing Then

            Return

        End If

        Try

            srFunction.sendFile(remoteUser, filepath)

        Catch ex As ObjectDisposedException

            srFunction = Nothing

            clientSocket = New TcpClient

            Return

        Catch ex As Exception

            writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

            Return

        End Try

    End Sub

This makes sure that a user is selected before proceeding, then opens the openFileDialog with the initial directory set to the logged-on user’s documents area. It then opens the dialog and waits till a file is selected, that file is then passed to the function that sends the file to the server.

#### TCP

##### The Main UI Updater for Threads

As in the server the whole ideaology of this class is to make it possible for different threads to write data out to the main UI and the functions below are needed for the whole messaging UI to work:

Imports Error\_Handler.Events\_Handler

Imports System.Reflection.MethodInfo

Public Class MainThreadUIUpdater

    Private Shared frmInstance As FRM\_Messages = Nothing

    Public Shared Sub addInstance(ByRef instance As Form)

        frmInstance = instance

    End Sub

    Public Shared Sub addUserToList(ByVal username As String, ByVal connected As Integer)

        If frmInstance Is Nothing Then

            writeDebugLog("Set an instance of the form with addInstance", GetCurrentMethod.Name, True)

            Return

        End If

        Try

            frmInstance.addUserToListView(username, connected)

        Catch ex As Exception

            writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

        End Try

    End Sub

    Public Shared Sub removeUserFromList(ByVal username As String)

        If frmInstance Is Nothing Then

            writeDebugLog("Set an instance of the form with addInstance", GetCurrentMethod.Name, True)

            Return

        End If

        Try

            frmInstance.removeUserFromListView(username)

        Catch ex As Exception

            writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

        End Try

    End Sub

    Public Shared Sub changeListIcon(ByVal username As String, ByVal connected As Integer)

        If frmInstance Is Nothing Then

            writeDebugLog("Set an instance of the form with addInstance", GetCurrentMethod.Name, True)

            Return

        End If

        Try

            frmInstance.changeUserIcon(username, connected)

        Catch ex As Exception

            writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

        End Try

    End Sub

    Public Shared Sub checkAndAdd(ByVal username As String, ByVal message As String, ByVal Optional file As Boolean = False)

        If frmInstance Is Nothing Then

            writeDebugLog("Set an instance of the form with addInstance", GetCurrentMethod.Name, True)

            Return

        End If

        Try

            frmInstance.checkAndAdd(username, message, file)

        Catch ex As Exception

            writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

        End Try

    End Sub

End Class

These functions then interface with the functions in the main form code file:

When adding a new message to the rich text box we firstly call a subroutine that checks if the currently selected user is the one that a message has been received from:

Public Sub checkAndAdd(ByVal username As String, ByVal message As String, ByVal Optional file As Boolean = False)

    If Me.InvokeRequired Then

        Dim dgate As New Delegate\_checkAndAdd(AddressOf checkAndAdd)

        Me.BeginInvoke(dgate, New Object() {username, message})

    Else

        If remoteUser = username And file Then

            WriteMsgToRTX(message, username, False, True)

        ElseIf remoteUser = username And Not file Then

            WriteMsgToRTX(message, username, False)

        Else

            If file Then

                setLSVBackColorOrange(username, Color.LightBlue)

            Else

                setLSVBackColorOrange(username, Color.Orange)

            End If

        End If

    End If

End Sub

If the currently selected user is the same as the one given by the parameter username then we call the function to add the message to the rich text box and a different Boolean value is passed based on whether the message is a file notification or not. Else if the user provided by **username** is not currently selected then the notification colour is added to the list view box.

When we want to add a new connected user to the list view box then we use the code below:

Public Sub addUserToListView(ByVal username As String, ByVal connected As Integer)

    If LSV\_UserContacts.InvokeRequired Then

        Dim Dgate As New Delegate\_addUserToListView(AddressOf addUserToListView)

        LSV\_UserContacts.BeginInvoke(Dgate, New Object() {username})

    Else

        Dim lvi As New ListViewItem

        lvi.ImageIndex = connected

        lvi.Text = username

        LSV\_UserContacts.Items.Add(lvi)

    End If

End Sub

This sets the new user as a list view item and selects the image index based on whether the user is currently connected provided by a Boolean value from the server. This list view item is then added to this list view.

Public Sub removeUserFromListView(ByVal username As String)

        If LSV\_UserContacts.InvokeRequired Then

            Dim Dgate As New Delegate\_removeUserFromListView(AddressOf removeUserFromListView)

            LSV\_UserContacts.BeginInvoke(Dgate, New Object() {username})

        Else

            For i = 0 To LSV\_UserContacts.Items.Count - 1

                If LSV\_UserContacts.Items(i).Text = username Then

                    LSV\_UserContacts.Items(i).Remove()

                    Return

                End If

            Next

        End If

    End Sub

This here loops through all of the users in the list view and once the text is equal to the username provided the row in the list view will be removed. This is not used anywhere anymore by the client as all users are shown who are provided by the server not just those who are connected.

Public Sub changeUserIcon(ByVal username As String, ByVal connected As Integer)

        If LSV\_UserContacts.InvokeRequired Then

            Dim Dgate As New Delegate\_chageUserIcon(AddressOf changeUserIcon)

            LSV\_UserContacts.BeginInvoke(Dgate, New Object() {username, connected})

        Else

            For i = 0 To LSV\_UserContacts.Items.Count - 1

                If LSV\_UserContacts.Items.Item(i).Text = username Then

                    LSV\_UserContacts.Items.Item(i).ImageIndex = connected

                End If

            Next

        End If

    End Sub

Lastly, this subroutine uses the same searching routine as deleting a user from the list view, except instead of calling the **Items.Item(i).remove**, instead it changes the icon of the connection to that users to the one passed as a parameter, this change is called when a user connects or disconnects.

##### Heartbeat for the Server

This is the code that sends the heartbeat values to the server indicating that the connection between the client and the server is still alive, it functions on exactly the same principles as the server except the client sends the message every 1.5 minutes and the server is set to close every two minutes.

Imports System.Net.Sockets

Public Class Heartbeat

    Public WithEvents Timer As System.Timers.Timer

    Public Event IOException(ByVal serverDead As Boolean)

    Private stream As NetworkStream

    'Public paused As Boolean

    Public Sub New(ByRef stream As NetworkStream)

        Timer = New System.Timers.Timer()

        Timer.Interval = 90000 '1.5 minutes

        Timer.Enabled = True

        Timer.Start()

        Me.stream = stream

    End Sub

    Private Sub timerFired() Handles Timer.Elapsed

        'If Not paused Then

        Dim value As Integer = 43775

        Dim heartbeat As Byte() = BitConverter.GetBytes(value)

        Try

            stream.Write(heartbeat, 0, heartbeat.Length)

        Catch ex As System.IO.IOException

            RaiseEvent IOException(True)

            Return

        End Try

        Timer.Stop()

        Timer.Start()

        'End If

    End Sub

End Class

##### User Lookup Table

Much like the server, which has a client lookup table, the clients also have a similar one that holds the username and associated elliptic curve point that represents that clients public key, there are a few functions that we use:

Public Class UserLookupTable

    Private lookupTable As Hashtable

    Public Enum clientLookupTableErrors

        CLIENT\_DOES\_NOT\_EXIST

        TABLE\_CONTAINS\_CLIENT

        NO\_ERROR

        CLIENT\_CONNECTED

        CLIENT\_NOT\_CONNECTED

    End Enum

    Public Sub New()

        lookupTable = New Hashtable

    End Sub

    Public Function addClient(ByVal username As String, ByVal publicKey As Curve25519\_ECDH.ECPoint) As clientLookupTableErrors

        If lookupTable.ContainsKey(username) Then

            Return clientLookupTableErrors.TABLE\_CONTAINS\_CLIENT

        End If

        lookupTable.Add(username, publicKey)

        Return clientLookupTableErrors.NO\_ERROR

    End Function

    Public Function removeClient(ByVal username As String) As clientLookupTableErrors

        If Not lookupTable.ContainsKey(username) Then

            Return clientLookupTableErrors.CLIENT\_DOES\_NOT\_EXIST

        End If

        lookupTable.Remove(username)

        Return clientLookupTableErrors.NO\_ERROR

    End Function

    Public Function getClientsPK(ByVal username As String) As Curve25519\_ECDH.ECPoint

        If Not lookupTable.ContainsKey(username) Then

            Return Nothing

        End If

        Return lookupTable(username)

    End Function

    Public Sub updateClientsPK(ByVal username As String, ByVal publicKey As Curve25519\_ECDH.ECPoint)

        lookupTable(username) = publicKey

    End Sub

End Class

Before writing about the **SendReceive** class I will first show you the code I needed for encryption purposes:

Imports Microsoft.Win32

Public Class RegistryTools

    Public Shared Sub addValueToRegistry(ByVal value As String, ByVal username As String)

        Dim RegKey As RegistryKey

        RegKey = Registry.CurrentUser.OpenSubKey("SOFTWARE", True)

        RegKey = RegKey.CreateSubKey("IM Application")

        RegKey.SetValue(username, value)

        RegKey.Close()

    End Sub

    Public Shared Function getRegistryValue(ByVal username As String) As String

        Dim RegKey As RegistryKey

        Dim ret As String

        RegKey = Registry.CurrentUser.OpenSubKey("SOFTWARE", True)

        RegKey = RegKey.CreateSubKey("IM Application")

        ret = RegKey.GetValue(username, "NULL")

        RegKey.Close()

        Return ret

    End Function

End Class

The code seen here interacts with the registry in order to store the currently logged on user’s private key so that they are able to decrypt any messages that were sent to them while they were away. The private key is encoded into Base64 format and stored as a string in the current user’s registry, for testing purposes the registry would only hold the current logged on user’s key rather than the any of the users that log into the client, therefore it would be impossible for other users to retrieve the private keys of other sessions.

For storing the public key, I initially thought about using a certificate file enabling me to store the file encrypted and therefore use the user’s password to decrypt the file when the user logged in and import the private key into the program. A better alternative is to use the current user’s key store in the registry (HKEY\_CURRENT\_USER), under HKEY\_CURRENT\_USER\SOFTWARE allows you to store variable specific to that user and to that application. I use the directory HKEY\_CURRENT\_USER\SOFTWARE\IM APPLICATION then I create a new value for every user that logs in, but for release, the program will be synced with Microsoft active directory and only allow the current logged on user to authenticate with the program, and therefore only one public key will be stored per user.

##### SendRecieve Class

This class encapsulates all of the code required for communication to take place with the server, it features two multithreaded sections, one for receiving data and one for sending data added to a linear queue. This class features the most database work plus all of the encryption and decryption for data sent between clients. The class is instanciated from form 1 when **SetupUserAfterLogin** is called by the login form:

Public Function SetupUserAfterLogin(ByVal passwordHash() As Byte, ByVal username As String) As Boolean

        LSV\_UserContacts.Items.Clear()

        Dim IPAddr As String = My.Settings.Default.serverIPAddress

        If IPAddr = "" Then

            IPAddr = Interaction.InputBox("Prompt", "Input Port")

        End If

        If My.Settings.Default.serverPort = 0 Then

            Try

                My.Settings.Default.serverPort = CInt(Interaction.InputBox("Enter the Servers port", "Input Port"))

            Catch ex As Exception

                Dim port As String

                Do

                    port = Interaction.InputBox("Enter the Servers port", "Input Port")

                Loop Until IsNumeric(port)

                My.Settings.Default.serverPort = CInt(port)

            End Try

        End If

        My.Settings.Default.serverIPAddress = IPAddr

        Try

            clientSocket.Connect(IPAddr, My.Settings.Default.serverPort)

        Catch ex As ObjectDisposedException

            clientSocket.Close()

            clientSocket = New TcpClient

            clientSocket.Connect(IPAddr, My.Settings.Default.serverPort)

        Catch ex As Exception

            errorHappened(clientSocket, ex.Message, errorEnum.SOCKET\_EXCEPTION, GetCurrentMethod.Name)

            LSV\_UserContacts.Items.Add("Not Logged In")

            Return False

        End Try

        Try

            srFunction = New SendRecieve(clientSocket, username, passwordHash)

        Catch ex As Exception

        'We cant return a form of failure as sub new cant be a function, therefore when it fails we just throw an error

        Try

            srFunction.closeConnection()

        Catch exx As Exception

        End Try

        srFunction = Nothing

        clientSocket.Close()

        clientSocket = New TcpClient

        LSV\_UserContacts.Items.Add("Not Logged In")

        Return False

        End Try

        localUsername = username

        TLS\_Username.Text = "Logged in as: " + localUsername

        LoginToolStripMenuItem.Visible = False

        LogoutToolStripMenuItem.Visible = True

        FilesToolStripMenuItem.Visible = True

        Return True

    End Function

This clears this list box on the left with the Not Logged In text so that its ready to add the new users to it, having done that if the IP Address is not set to any value in the settings file or if the port is not set to any value in the same file then a prompt will be created asking the users to input this data. We then set the those values to the settings file and create a new socket with the provided values. Once the socket is successfully created we then create a new instance of the **SendReceive** class and pass the socket instance to it plus the login username and the password hash, if that fails then we reset the **srFunction** and the socket and reset the form layout.

Instantiating **SendReceive** calls the **sub New** where the client details are sent to the server, the server will reply with a byte indicating the result of the logon, the server will then proceed to send the users to the client form the database and then finally send any unsent messages to the client:

Public Sub New(ByRef clientSocket As TcpClient, ByVal username As String, ByVal passwordHash() As Byte)

    Me.client = clientSocket

    stream = clientSocket.GetStream

    recBuffSize = clientSocket.ReceiveBufferSize

    lookupTable = New UserLookupTable

    MainThreadUIUpdater.addInstance(FRM\_Messages)

   localKeys = ECDH.generate\_Keys(curve.Parameters)

    Dim publicKeyx() As Byte = Curve25519\_ECDH.Crypto.AES.Convert\_BigInteger\_ToByteArray(localKeys.PublicKey.x)

    Dim publicKeyy() As Byte = Curve25519\_ECDH.Crypto.AES.Convert\_BigInteger\_ToByteArray(localKeys.PublicKey.y)

    Dim byteUsername() As Byte = ASCII.GetBytes(username)

    Dim data(client.ReceiveBufferSize - 1) As Byte

    byteUsername.CopyTo(data, 0)

    data(20) = publicKeyx.Length

    data(21) = publicKeyy.Length

    publicKeyx.CopyTo(data, 22)

    publicKeyy.CopyTo(data, 22 + publicKeyx.Length)

    passwordHash.CopyTo(data, 22 + publicKeyx.Length + publicKeyy.Length)

    Try

        stream.Write(data, 0, data.Length)

        stream.Flush()

    Catch ex As Exception

        errorHappened(stream, ex.Message, errorEnum.STREAM\_EXCEPTION, GetCurrentMethod.Name)

        Throw New Exception

    End Try

    writeDebugLog("Sending preliminary data to the server", GetCurrentMethod.Name)

    Dim authentication(1) As Byte

    stream.Read(authentication, 0, 1)

    Select Case CInt(authentication(0))

        Case Is = 1

            'Error in authentication

            errorHappened(authentication, "Either the username or the password was incorrect", errorEnum.FAILED\_SERVER\_AUTH, GetCurrentMethod.Name)

            Throw New Exception

        Case Is = 2

            'Already connected

            errorHappened(authentication, "A client is already connected with the provided details", errorEnum.CLIENT\_ALREADY\_CONNECTED, GetCurrentMethod.Name)

            Throw New Exception

        Case Is = 3

            'Normal

        Case Is = 4

            'Admin

    End Select

    databaseConnection = New DatabaseTools(DatabaseTools.databaseEnum.ClientDatabase, username)

    Dim clientCountByte(3) As Byte

    Try

        stream.Read(clientCountByte, 0, 4)

    Catch ex As Exception

        writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

        Throw New Exception

    End Try

    Dim clientCount As Int32 = BitConverter.ToInt32(clientCountByte, 0)

    writeDebugLog("Recieved currently connected users from the server with the count of: " & clientCount, GetCurrentMethod.Name)

    '| username | connected | x len | y len |public key x | public key y |

    For i = 0 To clientCount - 1

        Dim message(recBuffSize - 1) As Byte

        Dim clientUsernameByte(19) As Byte

        Dim lastNonZeroIndex As Integer = recBuffSize - 1

        stream.Read(message, 0, recBuffSize)

        lastNonZeroIndex = getLastNonZeroIndex(message)

        If lastNonZeroIndex - 19 <= 0 Then

            writeDebugLog("Recieved preliminary data from the server was not formatted correctly", GetCurrentMethod.Name, True)

            Throw New Exception

        End If

        If Not lastNonZeroIndex > 20 Then

            writeDebugLog("Recieved preliminary data from the server was not formatted correctly", GetCurrentMethod.Name, True)

            Throw New Exception

        End If

        For c = 0 To 19

            clientUsernameByte(c) = message(c)

        Next

        Dim usernameLastIndex As Integer = getLastNonZeroIndex(clientUsernameByte)

        Dim newUsername(usernameLastIndex) As Byte

        Array.Copy(clientUsernameByte, newUsername, usernameLastIndex + 1)

        Dim pkxLen As Integer = message(21)

        Dim pkylen As Integer = message(22)

        Dim pkx(pkxLen - 1) As Byte

        Dim pky(pkylen - 1) As Byte

        Array.ConstrainedCopy(message, 23, pkx, 0, pkxLen)

        Array.ConstrainedCopy(message, 23 + pkxLen, pky, 0, pkylen)

        Dim online As Boolean = CBool(message(20))

        Dim remotePK As New ECPoint(Curve25519\_ECDH.Crypto.AES.Convert\_IntegerByteArray\_ToBigInteger(pkx), Curve25519\_ECDH.Crypto.AES.Convert\_IntegerByteArray\_ToBigInteger(pky), curve.Parameters.Fp.p)

        lookupTable.addClient(ASCII.GetString(newUsername), remotePK)

        MainThreadUIUpdater.addUserToList(ASCII.GetString(newUsername), CInt(message(20)))

        Dim SQL As String

        SQL = "SELECT \* FROM TBL\_USERS WHERE [Username] = '" & ASCII.GetString(newUsername) & "';"

        Dim reader As OleDb.OleDbDataReader = databaseConnection.sendSQLQuery(SQL)

        reader.Read()

        If Not reader.HasRows Then

            SQL = "INSERT INTO TBL\_USERS VALUES ('" & ASCII.GetString(newUsername) & "');"

            databaseConnection.sendSQLNonQuery(SQL)

        End If

    Next

    recieveUnsentFiles(stream, username)

    RegistryTools.addValueToRegistry(System.Convert.ToBase64String(Curve25519\_ECDH.Crypto.AES.Convert\_BigInteger\_ToByteArray(localKeys.PrivateKey)), username)

    writeDebugLog("Populated the lookup table with the other users", GetCurrentMethod.Name)

    HeartbeatTimer = New Heartbeat(stream)

    Dim RThread As New Threading.Thread(AddressOf pollClientRecieve)

    RThread.Start()

    Dim SThread As New Threading.Thread(AddressOf MTsend)

    SThread.Start()

End Sub

We start off by setting the private variables to values provided by the parameters of the sub, we then commence by generating the keys used for this session, enabling us to send the new private key to the server, having generated the keys we commence by building up the initial packet that authenticates the user with the server, if authentication fails then the server does not store the public key so we don’t store the private key until all of the meta data has been sent first. Once the values are received by the server and checked it will return one of 4 values, as of yet the admin and normal user value do not do anything to change the functions of the program, but they are in place to easily allow any future changes to the client that require these values. After having confirmed that the user details were correct, the database connection is created, allowing for when the list of clients are received to be added to the database if any of the usernames are missing. Transmitted along with the username is the client’s public key, and a Boolean value indicating if the user is currently authenticated with the server. The **lookupTable** is then filled with the clients details, the key being their username as it’s a unique value and the values being the associated private key. Having received all the clients we then call **receiveUnsentFiles:**

Private Sub recieveUnsentFiles(ByVal stream As NetworkStream, ByVal username As String)

       Dim packetsExpected As Integer

       Dim packetsExpectedBytes(3) As Byte

       Dim readLength As Integer

       readLength = stream.Read(packetsExpectedBytes, 0, 4)

       If readLength <> 4 Then

           Dim currentReadLength As Integer = readLength

           Do

               readLength = stream.Read(packetsExpectedBytes, currentReadLength, 3 - currentReadLength)

               currentReadLength += readLength

           Loop Until currentReadLength = 3

       End If

       packetsExpected = BitConverter.ToInt32(packetsExpectedBytes, 0)

       For i = 0 To packetsExpected - 1

           Dim neededData(24) As Byte

           readLength = stream.Read(neededData, 0, 25)

           If readLength <> 25 Then

               Dim currentReadLength As Integer = readLength

               Do

                   readLength = stream.Read(neededData, currentReadLength, 25 - currentReadLength)

                   currentReadLength += readLength

               Loop Until currentReadLength = 25

           End If

           Dim usernameByteZeros(19) As Byte

           Array.ConstrainedCopy(neededData, 1, usernameByteZeros, 0, 20)

           Dim lastNonZeroUsernameIndex As Integer = getLastNonZeroIndex(usernameByteZeros)

           Dim usernameBytes(lastNonZeroUsernameIndex) As Byte

           Array.Copy(usernameByteZeros, usernameBytes, lastNonZeroUsernameIndex + 1)

           Dim dataLen As Integer = BitConverter.ToInt32(neededData, 21)

           Dim data(dataLen - 1) As Byte

           Dim filenameBytes(99) As Byte

           Dim filename As String

           If neededData(0) = 2 Then

               readLength = stream.Read(filenameBytes, 0, 100)

               If readLength <> 100 Then

                   Dim currentReadLength As Integer = readLength

                   Do

                       readLength = stream.Read(filenameBytes, currentReadLength, 100 - currentReadLength)

                       currentReadLength += readLength

                   Loop Until currentReadLength = 100

               End If

               Dim lastIndex As Integer = getLastNonZeroIndex(filenameBytes)

               Dim filenameNonZero(lastIndex) As Byte

               Array.Copy(filenameBytes, filenameNonZero, lastIndex + 1)

               filename = ASCII.GetString(filenameNonZero)

           End If

           readLength = stream.Read(data, 0, dataLen)

           If readLength <> dataLen Then

               Dim currentReadLength As Integer = readLength

               Do

                   readLength = stream.Read(data, currentReadLength, dataLen - currentReadLength)

                   currentReadLength += readLength

               Loop Until currentReadLength = dataLen

           End If

           Dim ECIES\_AES As New ECIES

           Dim CBC\_AES As New ECIES.TEncryption

           Dim MAC(MACBytesLen - 1) As Byte

           Dim IV(IVBytesLen - 1) As Byte

           Dim tmpData(data.Length - MACBytesLen - IVBytesLen - 1) As Byte

           Dim decryptedData() As Byte

           Dim privateKey As String = RegistryTools.getRegistryValue(username)

           If privateKey = "NULL" Then

               writeDebugLog("Could not find privatekey in the registry no messages could have been sent because we had no details", GetCurrentMethod.Name)

               Return

           End If

           Dim lastLocalKeys As Keys = ECDH.regeneratePublicKey(curve.Parameters, Curve25519\_ECDH.Crypto.AES.Convert\_IntegerByteArray\_ToBigInteger(System.Convert.FromBase64String(privateKey)))

           Array.ConstrainedCopy(data, data.Length - MACBytesLen, MAC, 0, MACBytesLen)

           Array.ConstrainedCopy(data, data.Length - MACBytesLen - IVBytesLen, IV, 0, IVBytesLen)

           Array.ConstrainedCopy(data, 0, tmpData, 0, data.Length - MACBytesLen - IVBytesLen)

           CBC\_AES.HMAC = MAC

           CBC\_AES.IV = IV

           CBC\_AES.Data = tmpData

           decryptedData = ECIES\_AES.Decrypt\_AES\_256(CBC\_AES, lastLocalKeys, curve, lookupTable.getClientsPK(ASCII.GetString(usernameBytes)))

           If decryptedData IsNot Nothing Then

               If neededData(0) = 1 Then

                   Dim SQL As String

                   SQL = "INSERT INTO TBL\_MESSAGES ([MessageData],[ReceivedDate],[IsFile],[Sender\_UserID]) VALUES ('" & ASCII.GetString(decryptedData) & "','" & DateTime.Today.Date.ToShortDateString & "',False,'" & ASCII.GetString(usernameBytes) & "');"

                   databaseConnection.sendSQLNonQuery(SQL)

                   MainThreadUIUpdater.checkAndAdd(ASCII.GetString(usernameBytes), ASCII.GetString(data))

               ElseIf neededData(0) = 2 Then

                   MainThreadUIUpdater.checkAndAdd(ASCII.GetString(usernameBytes), "The user sent you a file: " & filename, True)

                   Dim SQL As String

                   SQL = "INSERT INTO TBL\_FILES ([Filename],[Sender\_UserID]) VALUES ('" & filename & "','" & ASCII.GetString(usernameBytes) & "');"

                   databaseConnection.sendSQLNonQuery(SQL)

                   SQL = "SELECT COUNT([FileID]) FROM TBL\_FILES WHERE [Filename] = '" & filename & "' AND [Sender\_UserID] = '" & ASCII.GetString(usernameBytes) & "';"

                   Dim reader As OleDb.OleDbDataReader = databaseConnection.sendSQLQuery(SQL)

                   reader.Read()

                   Dim countOfFiles As Integer = reader.GetInt32(0)

                   reader.Close()

                   SQL = "SELECT [FileID] FROM TBL\_FILES WHERE [Filename] = '" & filename & "' AND [Sender\_UserID] = '" & ASCII.GetString(usernameBytes) & "';"

                   reader = databaseConnection.sendSQLQuery(SQL)

                   For c = 0 To countOfFiles - 1

                       reader.Read()

                   Next

                   Dim fileID As Integer = reader.GetInt32(0)

                   reader.Close()

                   SQL = "INSERT INTO TBL\_MESSAGES ([ReceivedDate],[IsFile],[Received\_FileID],[Sender\_UserID]) VALUES ('" & DateTime.Today.Date.ToShortDateString & "',True," & fileID & ",'" & ASCII.GetString(usernameBytes) & "');"

                   databaseConnection.sendSQLNonQuery(SQL)

                   Dim fs As New FileStream(FileIO.SpecialDirectories.MyDocuments & "/IM Application/" & ASCII.GetString(usernameBytes) & "/" & filename, FileMode.Create)

                   fs.Write(decryptedData, 0, decryptedData.Length)

                   fs.Close()

               End If

           End If

       Next

   End Sub

This subroutine starts of by gathering the count of packets we are expected to receive, once it has the 4 bytes that signify the integer, it the starts the for loop that reads those packets in, we start of by making sure that we read in the 25 bytes that signify the metadata of the packet that has been sent, once that has been received, we can get the packets length and therefore read in the final bytes that store the data, if the packet is a file then we remove the first 100 bytes that holds the file name of the file being sent we can then take the remaining data and populate the TEncryption structure with the values that are used for the decryption routine, if the decryption returns a null value then we cancel that loop as an error occurred and there are currently no ways of determining the error except from the messages written to the debug log. Having decrypted the data we can then either insert the message into the database if the message identifier at the start of the packet signifies that the data is an IM, else we can writeout the file using the **filestream** and add the file name to the database and add the fileID to the messages table. Once all of the packets have been received the control passes back to the calling function and the private key can then be added to the registry, with the two threads being started later on.

When starting the poll for data it starts a thread which runs this code in a continuous loop until **closeConnection** is called:

Private Sub pollClientRecieve()

    Do

        If stream.DataAvailable And canRead Then

            HeartbeatTimer.Timer.Stop()

            dataRecieved(Nothing)

            HeartbeatTimer.Timer.Start()

        End If

        Threading.Thread.Sleep(1000)

    Loop Until Ended

End Sub

Here this keeps looping through until data is available and the value canRead is set to true, this then allows the sub that deals with the data to be called and the heartbeat timer to be stopped.

When data is received the sub **dataRecieved** is called, which deals gathers the data from the network stream and then is passed through a series of statements that determine what the data consists of. The subroutine can be split down into different sections, where the data is read in and checked to try and catch some values and three sections where the data is deciphered differently depending on what data is available:

If \_disposed Then

    'Throw New ObjectDisposedException(Me.ToString)

    writeDebugLog("Object has already been disposed, reinitialise before accessing", GetCurrentMethod.Name, True)

    Return

End If

Dim recvBytes(recBuffSize - 1) As Byte

Dim identifier As Integer

Dim recvLength As Integer

'Initiate a networkstream lock so that writer cannot access it while the stream is being read

canWrite = False

If data Is Nothing Then

    recvLength = stream.Read(recvBytes, 0, recBuffSize)

    If (recvBytes(0) = 1 Or recvBytes(0) = 2 Or recvBytes(0) = 206) And recvLength <> recBuffSize Then

        Dim dataCounter As Integer = recvLength

        Do

            writeDebugLog("Got " & dataCounter & " out of " & recBuffSize, GetCurrentMethod.Name)

            recvLength = stream.Read(recvBytes, dataCounter, recBuffSize - dataCounter)

            dataCounter += recvLength

        Loop Until dataCounter = recBuffSize

    End If

    writeDebugLog("All data obtained", GetCurrentMethod.Name)

Else

    recvBytes = data

End If

'Try and catch any chance or recieveing ACK or ERROR values from the server

identifier = convertSetAmountToInt32(recvBytes, 2)

writeDebugLog("Recieved data from the server", GetCurrentMethod.Name)

If identifier = 43724 Or identifier = 61149 Then

    canWrite = True

    Return

End If

The first section above will take read in the data from the stream until we have around 65k bytes of data, I would like to change this section later on given the time and make it correct, as now with my wider knowledge of TCP the data being sent is unlimited and my successful implementation of this in **recieveUnsentFiles** shows that is the correct approach to using the TCP stream. Having read in the full number of bytes from the sender, when then finally perform a check to make sure that the first two bytes of the array are not the identifier for the ACK value or the error values from the server, if that was the case then we exist the sub as those rougue values are not useful to us. After having received the data the message identifier at the start of any sent data will be converted to integer, and four checks are performed.

Check one consists of seeing the data has the identifier 220 which communicates what user is disconnected:

identifier = CInt(recvBytes(0))

Select Case identifier

    Case Is = 220

        Dim usernameByte(20) As Byte

        Dim lastNonZeroIndex As Integer = 19

        For i = 0 To 19

            usernameByte(i) = recvBytes(i + 1)

        Next

        lastNonZeroIndex = getLastNonZeroIndex(usernameByte)

        If lastNonZeroIndex = 0 Then

            sendError()

            canWrite = True

            Return

        End If

        Dim correctUsername(lastNonZeroIndex) As Byte

        Array.Copy(usernameByte, correctUsername, lastNonZeroIndex + 1)

        'lookupTable.removeClient(ASCII.GetString(correctUsername))

        'MainThreadUIUpdater.removeUserFromList(ASCII.GetString(correctUsername))

        MainThreadUIUpdater.changeListIcon(ASCII.GetString(correctUsername), 0)

        writeDebugLog("Another user disconnected: " & ASCII.GetString(correctUsername), GetCurrentMethod.Name)

        sendACK()

        canWrite = True

        Return

Transmitted with the disconnect statemtne is the username of the user who has disconnected, allowing us to pass the username to the UI updater in order to change the icon in the user list view box to the disconnected icon.

The next check is to see if the data holds the information for a connecting user:

Case Is = 204

    '| message type | username | x len | y len | pkx | pky |

    Dim usernameByte(20) As Byte

    Dim lastNonZeroIndex As Integer = 19

    For i = 0 To 19

        usernameByte(i) = recvBytes(i + 1)

    Next

    lastNonZeroIndex = getLastNonZeroIndex(usernameByte)

    If lastNonZeroIndex = 0 Then

        sendError()

        canWrite = True

        Return

    End If

    Dim correctUsername(lastNonZeroIndex) As Byte

    Array.Copy(usernameByte, correctUsername, lastNonZeroIndex + 1)

    Dim pkxlen As Integer = recvBytes(21)

    Dim pkylen As Integer = recvBytes(22)

    Dim pkx(pkxlen - 1) As Byte

    Dim pky(pkylen - 1) As Byte

    Array.ConstrainedCopy(recvBytes, 23, pkx, 0, pkxlen)

    Array.ConstrainedCopy(recvBytes, 23 + pkxlen, pky, 0, pkylen)

    Dim remotePK As New ECPoint(Curve25519\_ECDH.Crypto.AES.Convert\_IntegerByteArray\_ToBigInteger(pkx), Curve25519\_ECDH.Crypto.AES.Convert\_IntegerByteArray\_ToBigInteger(pky), curve.Parameters.Fp.p)

    lookupTable.updateClientsPK(ASCII.GetString(correctUsername), remotePK)

    MainThreadUIUpdater.changeListIcon(ASCII.GetString(correctUsername), 1)

    writeDebugLog("Another user connected: " & ASCII.GetString(correctUsername), GetCurrentMethod.Name)

    sendACK()

    canWrite = True

    Return

Transmitted with a user connecting notification is the user’s username and their public key, we then obtain the username and the public key from the data and update the lookupTable with the new public key so that when sending a message, we use the most up to date public key to encrypt the user’s messages. Having done that we then change the icon in the list view box to the connected icon in order to notify the user that the client’s connection status has changed and that now they will be able to have a live conversation with them.

            Case Is = 206

        sendACK()

        RaiseEvent recievedPopulateValues(recvBytes)

        canWrite = True

        Return

End Select

The second to last identifier set is a reply from the server with a list of all the users in the database and their administrator status, this is used in the user settings form, allowing a logged in administrator to update credentials.

The final ones are the identifiers 1 and 2, these indicate either a message or a file respectively and the decoding routine for the messages (except where the x value =0) are the same for both identifiers. The point where x=0 is checked for explicitely and this packet conatins the filename for the file about to be sent by the user:  
I shal split this section into parts:

The first part checks wether the received data has the correct identifier then sets up the variables needed through the decoding phase of the message:

'| Message Type | x | of y | username | data hash | data |

'|     Byte     | B |   B  | 20 bytes | 32 bytes  | Bytes|

If recvBytes(0) <> 1 And recvBytes(0) <> 2 Then

    writeDebugLog("Recieved a packet with message type not equal to the know values", GetCurrentMethod.Name, True)

    canWrite = True

    Return

End If

Const offset As Integer = 3

Const usernameLen As Integer = 20

Const hashLen As Integer = 32

Dim SHA256 As New SHA256Managed()

Dim messageType As Byte = recvBytes(0)

Dim x As Byte = recvBytes(1)

Dim y As Byte = recvBytes(2)

Dim lengthOfUsefulData As Integer = recBuffSize - 1

If x = 0 Then

    writeDebugLog("Got filename", GetCurrentMethod.Name)

    Dim usernameByte(usernameLen - 1) As Byte

    For i = 0 To usernameLen - 1

        usernameByte(i) = recvBytes(i + offset)

    Next

    Dim lastNonZeroIndex As Integer = getLastNonZeroIndex(usernameByte)

    Dim finalUsername(lastNonZeroIndex) As Byte

    'Cut off the zero bytes by copying the userful data into a buffer

    Array.Copy(usernameByte, finalUsername, lastNonZeroIndex + 1)

    Dim tmpFileName(99) As Byte

    For i = 0 To 99

        tmpFileName(i) = recvBytes(i + offset + usernameLen)

    Next

    Dim fileNameLen As Integer = getLastNonZeroIndex(tmpFileName)

    Dim fileName(fileNameLen) As Byte

    If fileName.Length = 0 Then

        sendError()

        canWrite = True

        Return

    End If

    Array.Copy(tmpFileName, fileName, fileNameLen + 1)

    If Not Directory.Exists(FileIO.SpecialDirectories.MyDocuments & "/IM Application/" & ASCII.GetString(finalUsername)) Then

        Try

            Directory.CreateDirectory(FileIO.SpecialDirectories.MyDocuments & "/IM Application/" & ASCII.GetString(finalUsername))

        Catch sysex As System.ArgumentException

            writeDebugLog(sysex.Message, GetCurrentMethod.Name, True)

            sendError()

            canWrite = True

            Return

        End Try

    End If

    'Open the filestream for the next bytes we are ready to recieve

    Try

        fileStream = New FileStream(FileIO.SpecialDirectories.MyDocuments & "/IM Application/" & ASCII.GetString(finalUsername) & "/" & ASCII.GetString(fileName), FileMode.Create)

    Catch ex As Exception

        writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

    End Try

    sendACK()

    canWrite = True

    Return

End If

Once completing that we check to see if the x identifier of the packet is equal to 0, we then extract the username from the packet and remove the zeros off the end of the array, allowing us to convert the username to the correct ASCII string minus the extra spaces at the end caused by the zeros being present. We then read the 100 bytes containing the filename from the packet and remove the zeros from the end of the filename byte array for the same reason as with the username. Once we have the filename we then check to see if the default directory of <My Documents>/IM Application/<Username>/ exists and create it if not. We then open the filestream ready for when the rest of the bytes are receieved and then return. One of the benefits of changing the way we send the messges and files in a future update we will not have to use a global variable to store the messages and slowly fill it as we receive the packets, or with files open the filestream when we receive the x=0 packet and close it when we receive the last packet. Another positive to this change is alos we will be able to sent more data, currently we are limited by the maximum value of 256 for the y packet identifier, as we store the value as a byte thusly we have a theoretical maximum of 256 \* 65k bytes. By changing the method to sending a file in theory we have no maximum value, but in practice it would mean we would have to adapt our encryption routine, as currently we read the total maximum of the file in so therefore in that respect we are limited by the amount of memory the system has, and loading a file in greater than that memory limit will crash the whole system. Instead we would have to load 16 byte chunks of the file in at a time which could be done asynchronously while the program was encrypting. Aswell with bigger file sending the method for receiving data would have to be changed, and the asynchronous **BenginRead** routine would be needed so that a user with slower internet speeds would be able to use the program while the download occurs, this would then also necessitate the need for multiple network streams one for file downloads, one for message downloads and one for sending data to the server, in order to keep the program flowing.

writeDebugLog("Packet" & x & " of " & y, GetCurrentMethod.Name)

If x = y Then

    lengthOfUsefulData = getLastNonZeroIndex(recvBytes, usernameLen + 3)

End If

Dim usernameBytes(usernameLen - 1) As Byte

For i = 0 To usernameLen - 1

    usernameBytes(i) = recvBytes(i + offset)

Next

Dim lastNonZeroUnameIndex As Integer = getLastNonZeroIndex(usernameBytes)

'find the last index in the username array that has a byte that is != 0

Dim nonZeroUsername(lastNonZeroUnameIndex) As Byte

'Cut off the zero bytes by copying the userful data into a buffer

Array.Copy(usernameBytes, nonZeroUsername, lastNonZeroUnameIndex + 1)

Dim hashBytes(hashLen - 1) As Byte

For i = 0 To hashLen - 1

    hashBytes(i) = recvBytes(i + offset + usernameLen)

Next

Dim encryptedData(lengthOfUsefulData - offset - usernameLen - hashLen) As Byte

encryptedData = copyToMax(recvBytes, offset + usernameLen + hashLen, lengthOfUsefulData)

Dim genHashBytes() As Byte = SHA256.ComputeHash(encryptedData)

If genHashBytes.Length > hashLen Then

    'Not much we can do to deal with this, its a programming error

    writeDebugLog("Hash length longer than allowed", GetCurrentMethod.Name)

    sendACK()

    canWrite = True

    Return

ElseIf genHashBytes.Length < hashLen Then

    writeDebugLog("Hash lenght shorter than allowed", GetCurrentMethod.Name)

    sendError()

    canWrite = True

    Return

End If

'Compare the two hashes of the encrypted data, in order to make sure there were no errors when transmitting the file

For i = 0 To hashLen - 1

    If hashBytes(i) <> genHashBytes(i) Then

        writeDebugLog("Hash mismatch", GetCurrentMethod.Name)

        sendError()

        canWrite = True

        Return

    End If

Next

'The hashes were equal so we may now send the ACK so the client sends the next set of data

sendACK()

'Now we can take this data we have here and update the UI, save files, and such

If x = 1 And y <> 1 Then

    ReDim totalMessage((recvBytes.Length \* y) - 1)

ElseIf x = 1 And y = 1 Then

    ReDim totalMessage(encryptedData.Length - 1)

End If

If y > 1 Then

    encryptedData.CopyTo(totalMessage, (recvBytes.Length - offset - usernameLen - hashLen) \* (x - 1))

Else

    encryptedData.CopyTo(totalMessage, 0)

End If

This is the section that splits a packet down when it is received, it first gathers the username from the data and removes the zeros from the end of it. Having completed this, it continues by getting the hash from the data as well to check that the data recived was correct, and as with the server although this section is now depreciated as my understanding of the TCP has grown, I deemed it profitable to leave the data in there for error checking reasons, and to stop and malicious activities. After gathering the hash for the data, we then collect the data from the array and calculate the hash for that data, we then compare the arrays to make sure that the hashes are the same and proceed by writing the data to the **totalMessage** array which acts as a collection buffer beause of our method of sending data.

If messageType = 2 Then

    writeDebugLog("Writing out data to filestream", GetCurrentMethod.Name)

    If x = y Then

        Dim totalMessageLastNonZero As Integer = getLastNonZeroIndex(totalMessage)

        Dim message(totalMessageLastNonZero) As Byte

        Array.Copy(totalMessage, message, totalMessageLastNonZero + 1)

        Dim ECIES\_AES As New ECIES

        Dim CBC\_AES As New ECIES.TEncryption

        Dim MAC(MACBytesLen - 1) As Byte

        Dim IV(IVBytesLen - 1) As Byte

        Dim tmpData(message.Length - MACBytesLen - IVBytesLen - 1) As Byte

        Dim decrypteddata() As Byte

        Array.ConstrainedCopy(message, message.Length - MACBytesLen, MAC, 0, MACBytesLen)

        Array.ConstrainedCopy(message, message.Length - MACBytesLen - IVBytesLen, IV, 0, IVBytesLen)

        Array.ConstrainedCopy(message, 0, tmpData, 0, message.Length - MACBytesLen - IVBytesLen)

        CBC\_AES.HMAC = MAC

        CBC\_AES.IV = IV

        CBC\_AES.Data = tmpData

        decrypteddata = ECIES\_AES.Decrypt\_AES\_256(CBC\_AES, localKeys, curve, lookupTable.getClientsPK(ASCII.GetString(nonZeroUsername)))

        fileStream.Write(decrypteddata, 0, decrypteddata.Length)

        Dim SQL As String

        SQL = "INSERT INTO TBL\_FILES ([Filename],[Sender\_UserID]) VALUES ('" & Path.GetFileName(fileStream.Name) & "','" & ASCII.GetString(nonZeroUsername) & "');"

        databaseConnection.sendSQLNonQuery(SQL)

        SQL = "SELECT COUNT([FileID]) FROM TBL\_FILES WHERE [Filename] = '" & Path.GetFileName(fileStream.Name) & "' AND [Sender\_UserID] = '" & ASCII.GetString(nonZeroUsername) & "';"

        Dim reader As OleDb.OleDbDataReader = databaseConnection.sendSQLQuery(SQL)

        reader.Read()

        Dim countOfFiles As Integer = reader.GetInt32(0)

        reader.Close()

        SQL = "SELECT [FileID] FROM TBL\_FILES WHERE [Filename] = '" & Path.GetFileName(fileStream.Name) & "' AND [Sender\_UserID] = '" & ASCII.GetString(nonZeroUsername) & "';"

        reader = databaseConnection.sendSQLQuery(SQL)

        For c = 0 To countOfFiles - 1

            reader.Read()

        Next

        Dim fileID As Integer = reader.GetInt32(0)

        reader.Close()

        SQL = "INSERT INTO TBL\_MESSAGES ([ReceivedDate],[IsFile],[Received\_FileID],[Sender\_UserID]) VALUES ('" & DateTime.Today.Date.ToShortDateString & "',True," & fileID & ",'" & ASCII.GetString(nonZeroUsername) & "');"

        databaseConnection.sendSQLNonQuery(SQL)

        MainThreadUIUpdater.checkAndAdd(ASCII.GetString(nonZeroUsername), "The user sent you a file: " & Path.GetFileName(fileStream.Name), True)

        writeDebugLog("Closing filestream", GetCurrentMethod.Name)

        fileStream.Close()

        fileStream = Nothing

    End If

Else

    writeDebugLog("Collecting the message", GetCurrentMethod.Name)

    If x = y Then

        Dim totalMessageLastNonZero As Integer = getLastNonZeroIndex(totalMessage)

        Dim message(totalMessageLastNonZero) As Byte

        Array.Copy(totalMessage, message, totalMessageLastNonZero + 1)

        Dim ECIES\_AES As New ECIES

        Dim CBC\_AES As New ECIES.TEncryption

        Dim MAC(MACBytesLen - 1) As Byte

        Dim IV(IVBytesLen - 1) As Byte

        Dim tmpData(message.Length - MACBytesLen - IVBytesLen - 1) As Byte

        Dim decrypteddata() As Byte

        Array.ConstrainedCopy(message, message.Length - MACBytesLen, MAC, 0, MACBytesLen)

        Array.ConstrainedCopy(message, message.Length - MACBytesLen - IVBytesLen, IV, 0, IVBytesLen)

        Array.ConstrainedCopy(message, 0, tmpData, 0, message.Length - MACBytesLen - IVBytesLen)

        CBC\_AES.HMAC = MAC

        CBC\_AES.IV = IV

        CBC\_AES.Data = tmpData

        decrypteddata = ECIES\_AES.Decrypt\_AES\_256(CBC\_AES, localKeys, curve, lookupTable.getClientsPK(ASCII.GetString(nonZeroUsername)))

        Dim SQL As String

        SQL = "INSERT INTO TBL\_MESSAGES ([MessageData],[ReceivedDate],[IsFile],[Sender\_UserID]) VALUES ('" & ASCII.GetString(decrypteddata) & "','" & DateTime.Today.Date.ToShortDateString & "',False,'" & ASCII.GetString(nonZeroUsername) & "');"

        databaseConnection.sendSQLNonQuery(SQL)

        MainThreadUIUpdater.checkAndAdd(ASCII.GetString(nonZeroUsername), ASCII.GetString(decrypteddata))

    End If

End If

canWrite = True  'allow the writer to now access the stream

Once the last packet is received we can the take the buffer array **totalMessage** and use the data in there to reform the **TEncryption** structure allowing us to attempt to decrypt the message, with our decryption routines. If decryption succeesds then we can take the message or file and either write it out to the database of the filestream depending and then add a notification to the UI in order for the user to see the data.

###### Sending Data

The process of sending the data is an exact replica of the server’s process so I will not delve too deep into its workings:

Public Sub sendData(ByVal data() As Byte, Optional ByVal errorBytes As Boolean = False)

    'A problem occured which didnt throw an error except just had unecessary execution

    'If closeConnection() was called without the use of \_disposed then it was possible for the user to call any send function, due to error catching it would catch the stream disposed execption and then just end up close the connection again

    'This wasted execution time, to fix the problem we throw an ObjectDisposed exception which should notify the user that we can reset the object fine

    If \_disposed Then

        'Throw New ObjectDisposedException(Me.ToString)

        writeDebugLog("Object has already been disposed, reinitialise before accessing", GetCurrentMethod.Name, True)

        Return

    End If

    writeDebugLog("Adding new data for sending", GetCurrentMethod.Name)

    Dim parameters As New sendingParameters

    parameters.Data = data

    parameters.errorBytes = errorBytes

    linearMessageQueue.Enqueue(parameters)

End Sub

Private Sub MTsend()

    'keep the thread paused until the networkstream is free

    'Do

    '    Threading.Thread.Sleep(1000)

    'Loop Until canWrite And canRead

    Do

        If linearMessageQueue.Count > 0 And canWrite Then

            Dim params As sendingParameters = linearMessageQueue.Dequeue

            canRead = False

            HeartbeatTimer.Timer.Enabled = False

            HeartbeatTimer.Timer.Stop()

            Try

                stream.Write(params.Data, 0, params.Data.Length)

            Catch ex As Exception

                writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

                closeConnection(True)

                canRead = True

                Return

            End Try

            stream.Flush()

            If Not params.errorBytes Then

                Dim serverResponseBytes(recBuffSize - 1) As Byte

                Dim value As Integer

                Try

                    stream.Read(serverResponseBytes, 0, recBuffSize)

                Catch ex As Exception

                    writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

                    canRead = True

                    Return

                End Try

                Try

                    value = convertSetAmountToInt32(serverResponseBytes, 2)

                Catch ex As Exception

                    'Dont worry

                End Try

                Dim serverResponseEnum As sendingErrorEnum = CType(serverResponseBytes(3), sendingErrorEnum)

                If value = 61149 And serverResponseEnum = sendingErrorEnum.RESEND Then

                    writeDebugLog("Resending", GetCurrentMethod.Name, False)

                    Dim ret As Boolean = trySendAgain(params.Data)

                    If ret = False And \_disposed = False And killswitch = False Then

                        closeConnection(True)

                        canRead = True

                        Return

                    ElseIf ret = False And \_disposed = True Or killswitch = True Then

                        canRead = True

                        Return

                    End If

                ElseIf value = 61149 And serverResponseEnum <> sendingErrorEnum.RESEND Then

                    errorHappened(serverResponseBytes, "Recieved message from server: " & serverResponseEnum.ToString, errorEnum.VALUE\_CONDITIONS\_NOT\_MET, GetCurrentMethod.Name)

                ElseIf Not value = 43724 Then

                    writeDebugLog("Recieved a message from the server we wasnt expecting", GetCurrentMethod.Name, True)

                End If

                writeDebugLog("ACK", GetCurrentMethod.Name, False)

                HeartbeatTimer.Timer.Start()

                HeartbeatTimer.Timer.Enabled = True

            End If

            canRead = True

        End If

        Threading.Thread.Sleep(1000)

    Loop Until Ended

End Sub

Private Function trySendAgain(ByVal data() As Byte) As Boolean

    If \_disposed Then

        'Throw New ObjectDisposedException(Me.ToString)

        writeDebugLog("Object has already been disposed, reinitialise before accessing", GetCurrentMethod.Name, True)

        Return False

    End If

    'Debug.Print(data(recBuffSize - 1))

    If killswitch Then

        Return False

    End If

    writeDebugLog("Sending Again", GetCurrentMethod.Name)

    Try

        stream.Write(data, 0, data.Length)

    Catch ex As Exception

        writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

        Return False

    End Try

    stream.Flush()

    Dim serverResponseBytes(recBuffSize - 1) As Byte

    Dim value As Integer

    Try

        stream.Read(serverResponseBytes, 0, recBuffSize)

    Catch ex As Exception

        writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

        Return False

    End Try

    'Get the value returned as the enum

    value = convertSetAmountToInt32(serverResponseBytes, 2)

    Dim serverResponseEnum As sendingErrorEnum = CType(serverResponseBytes(3), sendingErrorEnum)

    If value = 61149 And serverResponseEnum = sendingErrorEnum.RESEND Then

        writeDebugLog("Resending", GetCurrentMethod.Name, False)

        If trySendAgain(data) = False Then

            Return False

        End If

    ElseIf value = 61149 And serverResponseEnum <> sendingErrorEnum.RESEND Then

        errorHappened(serverResponseBytes, "Recieved message from server: " & serverResponseEnum.ToString, errorEnum.VALUE\_CONDITIONS\_NOT\_MET, GetCurrentMethod.Name)

    ElseIf Not value = 43724 Then

        writeDebugLog("Recieved a message from the server we wasnt expecting", GetCurrentMethod.Name, True)

    End If

    Return True

End Function

The process consists of three subs, one of them acts a continuous thread and one of the supplies the data to the thread, the other is a recursive function that sends the data to the server until it is received correctly. The first function **sendData** adds the parameters to a linear queue and the line in the thread checks if something has been added:

        If linearMessageQueue.Count > 0 And canWrite Then

If the server sends back the error reply then and the value after it is equal to the resend value in the enum below, then we call the subroutine **trySendingAgain** this function then tries sending the data to the server again and if the server replies with the same error as before it will recursively call itself again to try and send the data. If the data is wrong in some way this function has a problem where it will never exit until the program crashes when the call stack is overflowed. The best method to combat this is to cancel sending if three retries have been performed and then notifier the user that sending the data failed.

Public Enum sendingErrorEnum

    RESEND = 0

    NOT\_ADMIN = 1

    PASSWORD\_MISMATCH = 2

    USER\_EXISTS = 3

    USER\_DOES\_NOT\_EXIST = 4

End Enum

There are also two replies that we can send to the server, an ACK and an error provided an enum is set:

Public Sub sendError()

    writeDebugLog("Sent Error", GetCurrentMethod.Name)

    Dim value As Integer = 61149

    Dim errorByte() As Byte = BitConverter.GetBytes(value)

    sendData(errorByte, True)

End Sub

Public Sub sendACK()

    writeDebugLog("Sent ACK", GetCurrentMethod.Name)

    Dim value As Integer = 43724

    Dim errorByte() As Byte = BitConverter.GetBytes(value)

    sendData(errorByte, True)

End Sub

In order to send a file or a IM to the server there are two procedures that make it easier to do the two. These procedures automatically set the data in the right order for the server including encrypting the data and call the **sendData** sub allowing the data to get transmitted to the server. The first subroutine is for sending a message:

Public Sub sendMessage(ByVal Username As String, ByVal Message As String, ByVal localUsername As String)

        'A problem occured which didnt throw an error except just had unecessary execution

        'If closeConnection() was called without the use of \_disposed then it was possible for the user to call any send function, due to error catching it would catch the stream disposed execption and then just end up close the connection again

        'This wasted execution time, to fix the problem we throw an ObjectDisposed exception which should notify the user that we can reset the object fine

        If \_disposed Then

            'Throw New ObjectDisposedException(Me.ToString)

            writeDebugLog("Object has already been disposed, reinitialise before accessing", GetCurrentMethod.Name, True)

            Return

        End If

        'The design of the messaging system should be [DEPRECIATED]:

        '[DEPRECIATED]| MessageType | Username Lenght | Username | Data |

        '[DEPRECIATED]|    Byte     |       Byte      |   Bytes  | Bytes|

        'Had to set the username to a max byte lenght of 20, due to the fact that when sending data split into 6500 byte chunks when the server changes the username theres a posibility we loose data

        '[DEPRECIATED]| MessageType | x | of y |   Username   | Data |

        '[DEPRECIATED]|    Byte     | B | Byte |Bytes (max 20)| Bytes|

        'Had to add a hash of the data in case of incorrectly sent messages

        '| Message Type | x | of y | username | data hash | data |

        '|     Byte     | B |   B  | 20 bytes | 32 bytes  | Bytes|

        If ASCII.GetByteCount(Username) > 20 Then

            writeDebugLog("Username longer than the maximum allowed of 20 bytes", GetCurrentMethod.Name, True)

            Return

        End If

        Dim UsernameBytes(19) As Byte

        Array.Copy(ASCII.GetBytes(Username), UsernameBytes, ASCII.GetByteCount(Username))

        Const usernameLength As Integer = 20

        Const messageType As Byte = 1

        Const offset As Integer = 3

        Const hashLen As Integer = 32

        Dim tmpMessageBytes() As Byte = Curve25519\_ECDH.Crypto.AES.Convert\_String\_ToByteArray(Message)

        Dim x As Integer

        Dim y As Integer

        Dim maxMessageByteLength As Integer = recBuffSize - offset - usernameLength - hashLen

        Dim sha256 As New SHA256Managed

        Dim lastNonZeroIndex As Integer = getLastNonZeroIndex(tmpMessageBytes)

        Dim nonZeroMessageBytes(lastNonZeroIndex) As Byte

        Dim CBC\_AES\_Return As ECIES.TEncryption

        Dim remotePoint As ECPoint = lookupTable.getClientsPK(Username)

        Array.Copy(tmpMessageBytes, nonZeroMessageBytes, lastNonZeroIndex + 1)

        CBC\_AES\_Return = ECIES\_AES.Encrypt\_AES\_256(nonZeroMessageBytes, localKeys, curve, remotePoint)

        If CBC\_AES\_Return.didError Then

            errorHappened(CBC\_AES\_Return, "Failed to encrypt data", errorEnum.ENCRYPTION\_FAILED, GetCurrentMethod.Name)

            Return

        End If

        Dim encryptedData(IVBytesLen + MACBytesLen + CBC\_AES\_Return.Data.Length - 1) As Byte

        CBC\_AES\_Return.Data.CopyTo(encryptedData, 0)

        CBC\_AES\_Return.IV.CopyTo(encryptedData, CBC\_AES\_Return.Data.Length)

        CBC\_AES\_Return.HMAC.CopyTo(encryptedData, CBC\_AES\_Return.Data.Length + IVBytesLen)

        If encryptedData.Length > maxMessageByteLength Then

            y = CInt(encryptedData.Length / maxMessageByteLength) + 1

        Else

            y = 1

        End If

        For x = 1 To y

            Dim messageBytes(maxMessageByteLength - 1) As Byte

            If x = y And y <> 1 Then

                ReDim messageBytes(maxMessageByteLength - (tmpMessageBytes.Length - lastNonZeroIndex - 1))

                Array.ConstrainedCopy(encryptedData, maxMessageByteLength \* (x - 1), messageBytes, 0, maxMessageByteLength - (tmpMessageBytes.Length - lastNonZeroIndex - 1) + 1)

            ElseIf x = y And y = 1 Then

                ReDim messageBytes(encryptedData.Length - 1)

                messageBytes = encryptedData

            Else

                messageBytes = encryptedData

            End If

            Dim hashBytes() As Byte = sha256.ComputeHash(messageBytes)

            If hashBytes.Length > hashLen Then

                writeDebugLog("Generated hash for the message was longer than the 32 bytes allowed for SHA256 algorithm", GetCurrentMethod.Name, True)

                Return

            End If

            Dim fullMessage(recBuffSize - 1) As Byte

            fullMessage(0) = messageType

            fullMessage(1) = CByte(x)

            fullMessage(2) = CByte(y)

            'fullMessage(3) = usernameLength

            UsernameBytes.CopyTo(fullMessage, offset)

            hashBytes.CopyTo(fullMessage, offset + usernameLength)

            messageBytes.CopyTo(fullMessage, offset + usernameLength + hashLen)

            'For i = 0 To usernameLength - 1

            '    fullMessage(i + offset) = UsernameBytes(i)

            'Next

            'i = 0

            'For i = 0 To messageBytes.Length - 1

            '    fullMessage(i + offset + usernameLength) = messageBytes(i + (maxMessageByteLength \* (x - 1)))

            'Next

            Try

                writeDebugLog("Sending IM to the server", GetCurrentMethod.Name)

                sendData(fullMessage)

            Catch ex As ObjectDisposedException

                writeDebugLog("Unable to send the file, stream was disposed before we finshed", GetCurrentMethod.Name, True)

                Return

            End Try

        Next

        Dim SQL As String

        'Add our sent message to the database in non encrypted form, set the recieve date to todays and set the sender UserID to the recipients

        SQL = "INSERT INTO TBL\_MESSAGES ([MessageData], [ReceivedDate], [IsFile], [Sender\_UserID]) VALUES ('" & Message & "', '" & Date.Today.ToShortDateString.ToString & "', " & False & ", '" & localUsername & "');"

        databaseConnection.sendSQLNonQuery(SQL)

    End Sub

It first starts off by checking to make sure that the function has not been disposed of which is set to true when closeConnection is called. We also check to make sure that the username is not longer than the maximum 20 bytes. We first start of by perfomring the encryption of the message, taking the data from that, the MAC, the IV and the ciphertext and copying it to an array which will then be split into 65k chunks and added to the end of each packet. We then calculate how many packets we need which is stored as the y value in the packet and we construct each packet starting off with the ID which for messages is equal to 1, followed by the x value and the total number of packets to expected as a y value, after that we insert the 20 bytes for the username and then copy the hash of the chunmk of encrypted data to the array. We construct the packets in a loop which runs from 1 to y and we can use that value as the inidicator to what packet we start on. Having created a packet, we then call **sendData** and write that packet to the queue, then continue on to build up the next packet calling **sendData** again at the end.

The function for sending the files is exactly the same but instead of taking the data in as a string in the parameters, we take the filename in and open that file in a filestream, we then read the file into memory and pass it to the encryption routine, taking the outputs as we did in **sendMessage**, the construction of the packets is the the same as in the procedure above:

Public Sub sendFile(ByVal Username As String, ByVal FilePath As String)

        'A problem occured which didnt throw an error except just had unecessary execution

        'If closeConnection() was called without the use of \_disposed then it was possible for the user to call any send function, due to error catching it would catch the stream disposed execption and then just end up close the connection again

        'This wasted execution time, to fix the problem we throw an ObjectDisposed exception which should notify the user that we can reset the object fine

        If \_disposed Then

            'Throw New ObjectDisposedException(Me.ToString)

            writeDebugLog("Object has already been disposed, reinitialise before accessing", GetCurrentMethod.Name, True)

            Return

        End If

        If ASCII.GetByteCount(Username) > 20 Then

            'Throw New Exception("Username longer than the maximum allowed of 20 bytes")

            errorHappened(Me, "Username longer than the maximum allowed of 20 bytes", errorEnum.USERNAME\_LONGER\_THAN\_MAX, GetCurrentMethod.Name)

            Return

        End If

        If Not File.Exists(FilePath) Then

            writeDebugLog("Unable to find the file specified at path: " & FilePath, GetCurrentMethod.Name, True)

            Return

        End If

        '| Message Type | x | of y | username | data hash | data |

        '|     Byte     | B |   B  | 20 bytes | 32 bytes  | Bytes|

        'Newest revision of the message data where we must include the hash for the data that got sent that time, this was due to possible transmission errors

        'the hash is SHA256 so should be 32 bytes of data when the server gets the data the hash can be checked against generated hash for the data, and same on client end

        'Dim i As Integer

        Dim UsernameBytes(19) As Byte

        Array.Copy(ASCII.GetBytes(Username), UsernameBytes, ASCII.GetByteCount(Username))

        Dim usernameLength As Byte = 20

        Const messageType As Byte = 2

        Dim x As Integer

        Dim y As Integer

        Const offset As Integer = 3

        Const hashLen As Integer = 32

        Const filenameLen As Integer = 100

        Dim maxMessageByteLength As Integer = recBuffSize - offset - usernameLength - hashLen

        Dim fs As FileStream

        Dim sha256 As New SHA256Managed

        Try

            fs = New FileStream(FilePath, FileMode.Open)

        Catch ex As Exception

            writeDebugLog(ex.Message, GetCurrentMethod.Name, True)

            Return

        End Try

        If (fs.Length + MACBytesLen + IVBytesLen) > maxMessageByteLength Then

            y = CInt((fs.Length + MACBytesLen + IVBytesLen) / maxMessageByteLength) + 1

        Else

            y = 1

        End If

        If y > 256 Then

            errorHappened(fs, "File length greater than allowed", errorEnum.VALUE\_LONGER\_THAN\_MAX, GetCurrentMethod.Name)

        End If

        Dim fileName() As Byte = ASCII.GetBytes(Path.GetFileName(FilePath))

        If fileName.Length > filenameLen Then

            writeDebugLog("Filename length was longer than allowed", GetCurrentMethod.Name, True)

            fs.Dispose()

            Return

        End If

        'Contains the filename and thats sent first, we then setup the filestream ready for storage on the other end

        Dim initalMessage(recBuffSize - 1) As Byte

        initalMessage(0) = messageType

        initalMessage(1) = 0

        initalMessage(2) = CByte(y)

        UsernameBytes.CopyTo(initalMessage, offset)

        fileName.CopyTo(initalMessage, offset + usernameLength)

        sendData(initalMessage)

        Dim fileBytes(fs.Length - 1) As Byte

        fs.Read(fileBytes, 0, fs.Length)

        fs.Close()

        Dim lastNonZeroIndex As Integer = getLastNonZeroIndex(fileBytes)

        Dim fileBytesNoZeros(lastNonZeroIndex) As Byte

        Array.Copy(fileBytes, fileBytesNoZeros, lastNonZeroIndex + 1)

        Dim CBC\_AES\_Return As ECIES.TEncryption

        Dim remotePoint As ECPoint = lookupTable.getClientsPK(Username)

        CBC\_AES\_Return = ECIES\_AES.Encrypt\_AES\_256(fileBytesNoZeros, localKeys, curve, remotePoint)

        If CBC\_AES\_Return.didError Then

            errorHappened(CBC\_AES\_Return, "Failed to encrypt data", errorEnum.ENCRYPTION\_FAILED, GetCurrentMethod.Name)

            Return

        End If

        Dim encryptedData(IVBytesLen + MACBytesLen + CBC\_AES\_Return.Data.Length - 1) As Byte

        CBC\_AES\_Return.Data.CopyTo(encryptedData, 0)

        CBC\_AES\_Return.IV.CopyTo(encryptedData, CBC\_AES\_Return.Data.Length)

        CBC\_AES\_Return.HMAC.CopyTo(encryptedData, CBC\_AES\_Return.Data.Length + IVBytesLen)

        For x = 1 To y

            Dim data(maxMessageByteLength - 1) As Byte

            If x = y AndAlso encryptedData.Length - (maxMessageByteLength \* (y - 1)) < maxMessageByteLength Then

                ReDim data(encryptedData.Length - (maxMessageByteLength \* (y - 1)) - 1)

                Array.ConstrainedCopy(encryptedData, maxMessageByteLength \* (x - 1), data, 0, encryptedData.Length - (maxMessageByteLength \* (y - 1)))

            Else

                Array.ConstrainedCopy(encryptedData, maxMessageByteLength \* (x - 1), data, 0, maxMessageByteLength)

            End If

            Dim fullMessage(recBuffSize - 1) As Byte

            Dim hashBytes() As Byte = sha256.ComputeHash(data)

            If hashBytes.Length > hashLen Then

                writeDebugLog("Generated hash for the message was longer than the 32 bytes allowed for SHA256 algorithm", GetCurrentMethod.Name, True)

                Return

            End If

            fullMessage(0) = messageType

            fullMessage(1) = CByte(x)

            fullMessage(2) = CByte(y)

            UsernameBytes.CopyTo(fullMessage, offset)

            hashBytes.CopyTo(fullMessage, offset + usernameLength)

            data.CopyTo(fullMessage, offset + usernameLength + hashLen)

            Try

                writeDebugLog("Sending file to the server", GetCurrentMethod.Name)

                sendData(fullMessage)

            Catch ex As ObjectDisposedException

                writeDebugLog("Unable to send the file, stream was disposed before we finshed", GetCurrentMethod.Name, True)

                Return

            End Try

        Next

        writeDebugLog("Finished sending file to the server", GetCurrentMethod.Name)

    End Sub

###### Interfacing with the Server’s Database

From pages 120 through 127 we talk about the ways an administrator can log into a client session and update values in the server’s database without direct access to the server, the code below performs these actions with the server and thusly provide the interface for the client to the server’s database:

Public Sub sendNewUser(ByVal username As String, ByVal passwordHash() As Byte, ByVal isAdmin As Boolean)

    If passwordHash.Length > 32 Then

        errorHappened(passwordHash, "The length of the hash was longer than allowed", errorEnum.VALUE\_LONGER\_THAN\_MAX, GetCurrentMethod.Name)

        Return

    End If

    Dim usernameBytes() As Byte = ASCII.GetBytes(username)

    If usernameBytes.Length > 20 Then

        errorHappened(usernameBytes, "Username was longer than allowed", errorEnum.USERNAME\_LONGER\_THAN\_MAX, GetCurrentMethod.Name)

        Return

    End If

    Dim fullMessage(recBuffSize - 1) As Byte

    Dim messageID As Byte = 202

    fullMessage(0) = messageID

    usernameBytes.CopyTo(fullMessage, 1)

    passwordHash.CopyTo(fullMessage, 21)

    fullMessage(53) = CByte(isAdmin)

    sendData(fullMessage)

End Sub

Public Sub updatePasswordHash(ByVal username As String, ByVal oldPasswordHash() As Byte, ByVal newPasswordHash() As Byte)

    If oldPasswordHash.Length > 32 Or newPasswordHash.Length > 32 Then

        errorHappened(oldPasswordHash, "The length of the hash was longer than allowed", errorEnum.VALUE\_LONGER\_THAN\_MAX, GetCurrentMethod.Name)

        Return

    End If

    Dim usernameBytes() As Byte = ASCII.GetBytes(username)

    If usernameBytes.Length > 20 Then

        errorHappened(usernameBytes, "Username was longer than allowed", errorEnum.USERNAME\_LONGER\_THAN\_MAX, GetCurrentMethod.Name)

        Return

    End If

    Dim fullMessage(recBuffSize - 1) As Byte

    Dim messageID As Byte = 203

    fullMessage(0) = messageID

    usernameBytes.CopyTo(fullMessage, 1)

    oldPasswordHash.CopyTo(fullMessage, 21)

    newPasswordHash.CopyTo(fullMessage, 53)

    sendData(fullMessage)

End Sub

Public Sub updateUsersRights(ByVal username As String, ByVal isAdmin As Boolean)

    Dim usernameBytes() As Byte = ASCII.GetBytes(username)

    If usernameBytes.Length > 20 Then

        errorHappened(usernameBytes, "Username was longer than allowed", errorEnum.USERNAME\_LONGER\_THAN\_MAX, GetCurrentMethod.Name)

        Return

    End If

    Dim fullMessage(recBuffSize - 1) As Byte

    Dim messageID As Byte = 204

    fullMessage(0) = messageID

    usernameBytes.CopyTo(fullMessage, 1)

    fullMessage(22) = CByte(isAdmin)

    sendData(fullMessage)

End Sub

Public Sub removeUser(ByVal username As String)

    Dim usernameBytes() As Byte = ASCII.GetBytes(username)

    If usernameBytes.Length > 20 Then

        errorHappened(usernameBytes, "Username was longer than allowed", errorEnum.USERNAME\_LONGER\_THAN\_MAX, GetCurrentMethod.Name)

        Return

    End If

    Dim fullMessage(recBuffSize - 1) As Byte

    Dim messageID As Byte = 205

    fullMessage(0) = messageID

    usernameBytes.CopyTo(fullMessage, 1)

    sendData(fullMessage)

End Sub

Public Sub snycWithServer()

    Dim message(1) As Byte

    message(0) = 206

    sendData(message)

End Sub

## Error Handler

In order to micromanage errors and to inform the user what is going on or keeping a record of whats happening in functions I created a small error handler as a dll that can be imported in an application. It started of by creating a log file and saving data to that but that was no longer possible in the end as importing it into a dll then importing that dll into the main program and also importing the error handler dll into the program too caused errors with access to that one log file. Instead now it just writes out to the debug window in visual studio or a message box if **errorHappend** is called:

Imports System.Diagnostics

Imports System.Reflection.MethodInfo

Imports System.IO

Imports System.Windows.Forms

Public Class Events\_Handler

    Public Enum errorEnum

        NULL\_REFERECE\_EXCEPTION = 1

        DIVIDE\_BY\_ZERO = 2

        POINT\_INFINITY = 3

        POINT\_NOT\_ON\_CURVE = 4

        VALUE\_NOT\_ZERO = 5

        VALUE\_CONDITIONS\_NOT\_MET = 6

        ECPOINT\_ARITHMETIC\_ERROR = 7

        ENCRYPTION\_FAILED

        PARAM\_BIT\_LENGTH\_NOT\_EQUAL = 10

        PARAM\_B\_EQUALS\_ZERO = 11

        DOMAIN\_PARAMETERS\_EQUALS\_ZERO = 12

        COFACTOR\_NOT\_EQUAL\_TO\_ONE = 13

        USERNAME\_LONGER\_THAN\_MAX

        VALUE\_LONGER\_THAN\_MAX

        FAILED\_SERVER\_AUTH

        CLIENT\_ALREADY\_CONNECTED

        STREAM\_EXCEPTION

        SOCKET\_EXCEPTION

    End Enum

    'Unable to use events when inheriting the class, RaiseEvent shows the error "Derived classes cannont raise base class events"

    'Protected Event Exception(ByVal Sender As System.Object, ByVal Message As System.String)

    Public Shared Sub errorHappened(ByVal Sender As System.Object, ByVal Message As System.String, ByVal Ex As errorEnum, ByVal currentSub As String)

        Dim outString As String

        outString = Sender.ToString & Space(5) & Ex.ToString & vbNewLine & "Error occured in: " & currentSub & vbNewLine & "Message generated by error: " & Message

        MessageBox.Show(outString, "Exception, Recovering", MessageBoxButtons.OK, MessageBoxIcon.Error)

        writeDebugLog(outString, currentSub, True)

    End Sub

    'Protected Event writeLog(ByVal Sender As System.Object, ByVal Message As System.String)

    Public Shared Sub writeDebugLog(ByVal Message As System.String, ByVal currentSub As String, Optional ByVal isException As Boolean = False)

        Dim outString As String

        If isException Then

            outString = "EXCEPTION: " & currentSub & Space(5) & Message

        Else

            outString = currentSub & ": " & Message

        End If

        'MessageBox.Show(outString, "Exception, Recovering", MessageBoxButtons.OK, MessageBoxIcon.Error)

        Debug.Print(outString)

        'Dim stmWriter As StreamWriter = Open\_File()

        'stmWriter.WriteLine(outString)

        'stmWriter.Dispose()

        'stmWriter = Nothing

    End Sub

    'Private Shared Sub closeFile(ByVal Sender As Object, ByVal e As System.EventArgs)

    '    stmWriter.Write("\\\\\\\\\\\\\\\\\\\\\\\ TAIL OF LOG \\\\\\\\\\\\\\\\\\\\\\\")

    '    stmWriter.Close()

    'End Sub

    'Public Shared Function Open\_File() As StreamWriter

    '    Dim debugLogFileName As String = Directory.GetCurrentDirectory() + "\Log.txt"

    '    If Not File.Exists(debugLogFileName) Or File.GetCreationTime(debugLogFileName) <> Date.Today Then

    '        Dim tmp As FileStream = File.Create(debugLogFileName)

    '        tmp.Close()

    '    End If

    '    Dim stmWriter As New StreamWriter(debugLogFileName, True)

    '    'Dim DT As New DateTime

    '    'DT = DateTime.Now

    '    'stmWriter.Write(DT.Date.ToShortDateString)

    '    'stmWriter.Write(Space(1) & DT.TimeOfDay.ToString & vbNewLine)

    '    'stmWriter.WriteLine("\\\\\\\\\\\\\\\\\\\\\\\ HEAD OF LOG \\\\\\\\\\\\\\\\\\\\\\\")

    '    Return stmWriter

    'End Function

End Class

As you can see above some of the commented-out code is the orrigional code that created the file bu that was no longer going to work with the intent that I had for it.

## Unit Testing

### Encryption

In order to test encryption, I started off by testing the public classes and with AES I tested the private subroutines that perform the changes on the data in order to assure that the data was being processed correctly:

#### EC Point Tests

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *A user creates a new instance of an ECPoint a provides only the x an y values* | *Test succeeded* |  |
| *A user creates a new instance of ECPoint and provides the x, y and p values* | *Test succeeded* |  |
| *A user creates a new instance of ECPoint providing another point as the base* | *Test succeeded* |  |
| *Checking if the point is on a curve by providing domin parameters* | *The test succeeds when the user provides a valid point and valid domain parameter.* | *There is no checking for if the given domain parameters are null or if the point coordinates are null, but this could never happen in ECC ecnryption* |
| *Checking if a point was set to erroneous* | *The test succeeds fine* |  |
| *Checking the point is at inifinity* | *The test succeeded* |  |
| *Checking if the point is null* | *The test succeeded* |  |
| *Setting a point equal to a null value* | *The test succeeded* |  |

#### EC Point Operations Tests

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Check that addition is performed properly* | *Test failed, the returned value was correct when adding on the prime* | *I used the values 10,10,47 and 20,20,47 and found that the returned value and the correct value calculated with a calculator online had a difference of the prime value when so I added on the prime when the value was negative* |
| *Check that point addition functions correctly after the update* | *The test succeeded and the correct point was returned* |  |
| *Check that the point doubling function works* | *Test failed with the same problem as in the first test of point addition* | *After having learnt about the first test of aditions problem it was easy to spot the error and so by copying the code I used in addition to make it correct and changing some of the values point doubling tests succeeded* |
| *Check tha point doubling now works after the update* | *Test succeeded and the point was doubled correctly* |  |
| *Perform scalar multiplication of a point* | *The test initially succeeded but failed when the domain parameter a was set to 0 the check for if a = 0 was removed and the test succeeeded* |  |
| *Test the extended elucidean algorithm* | *The test succeeded straight away and returned the x and y values plus the GCD of the provided input* |  |

#### Testing Domain Parameters class

|  |  |  |
| --- | --- | --- |
| *Test scenario* | *Test Outcome* | *Notes* |
| *Testing the validate parameters function* | *The test failed for the bitlengths of both the base point’s x and y values and the bitlength of the prime integer* | *Removing the checks for the bitlength of the base and the bitlength of the prime field stoped the error and the test succeeded.* |
| *Testing the validate parameters function* | *The test failed this time when trying to use the Secp-256k1 curve where the parameters a and b are not 256* | *When using the curve, the parameter a is 0 and the parameter B is 7 and therefore the parameters are not 256 bits in length so the import fails.* |
| *Providing null parameters* | *The test fails as the provided parameters are not checked to see if they are null* | *Adding a test for null parameters ensures that the domain parameters are even more likely to be correct.* |

#### Testing Importing of a Curve

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Import of a correct curve from the xml file* | *Test succeeded* |  |
| *Import of a curve not present in the xml file* | *Test succeeded, error was raised telling the user that the curve could not be found in the file* |  |
| *Import of a curve where there aren’t the correct number of nodes in the file* | *The test succeeded and the program throws an error that informs the user that the curve entry is corrupted* |  |
| *Import of a curve from the xml file where the curves attributes are present but the curve is not correct* | *The test fails because the import succeeds as the domain parameters were not validated* | *By adding a validation routine for the imported curves its more likely to catch domain parameters that are not correct and therefore supress errors on further into the procedure* |
| *Import of a curve from the xml file where the curves attributes are present but the curve is not correct* | *Test succeeds and the user is notified that the imported curve was incorrect* |  |
| *Importing a curve from the xml file where the curve had the correct number of child nodes but the nodes had no text* | *The test succeeds as the import crashes as there was no text to read in* | *The test succeeds because there was a crash, that crash can be stopped by placing a try catch around the instanciation of the weierstrass\_Curve object, but the test succeeds because the import failed to continue* |
| *Trying to import from a file that doesn’t exist* | *Test succeeds, the program crashes and the user is notified* |  |

As with the tests above, due to the nature of how instanciating an object works in vb.net the only way of notifying a calling function that something has failed is to let the program crash and therefore catch that crash with a try catch, then the execution can be stopped before the problem breaks any other areas of the program.

#### Testing the Fp Class

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Trying to create an instance of Fp with a correct prime* | *Test succeeds* |  |
| *Trying to create an instance of Fp with an incorrect prime, or a prime that does not exist* | *Test succeeds and an error is thrown stating that the provided prime number was not prime* |  |

#### Testing the AES Routines

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Checking that Strings are correctly converted to byte array* | *Test succeeded* |  |
| *Checking if the byte array converts back to string* | *Test succeeded and the byte array from the first test returned the string passed in the first test* |  |
| *Converting a BigInteger to byte array* | *The test succeeds and the BigInteger is converted correctly for the endianness of the computer* |  |
| *Converting the byte array back to BigInteger* | *The test succeeds and the byte array is converted correctly* |  |
| *Converting from a state to a byte array* | *Test converts the fine* |  |
| *Converting from a byte array to a State* | *Converts fine* | *Added a quick test to make sure that the byte array was not longer than 16 bytes so that last values do not get cut off* |
| *Convert a byte array to a state3* | *Converts fine* |  |
| *Populating an array from a start position* | *Populates the array correctly* |  |
| *Copy a state3 to a state2* | *Copies the arrays correctly* |  |
| *Perfomr the exponentiation of 2 to a user specified value* | *Works correctly* |  |
| *Cyclic rotation of a byte array around wrapping the last value round to the start* | *Works correctly* |  |
| *Calculating the inverse of the number inside Rijndaels finite field* | *Test failed* | *The program couldn’t properly generate the SBox values for the lookup so instead we defined an array of pre-calculated values* |
| *Lookup of the SBox array for the value we need* | *Test succeeded* |  |
| *Calculating the inverse of the galios field multiplication* | *Test failed* | *We were unable to generate the log and antilog tables correctly when performing the operation so as we did with SBox we defined an array with the pre-calculated values* |
| *Lookup of the value in the anit-log and log tables* | *Test succeeded* |  |
| *Calculating the galios field multiplication of two numbers* | *Test succeeded* | *When replacing the same routine that generated the antilog log values in gmul\_inverse I replaced the same values in this subroutine and it worked straight away* |
| *Peforming Rijndael’s key stretching on a give private key* | *Test succeeded* | *The test succeeded once the functions above were corrected to work with the pre-defined array values* |
| *Testing the XOR of two states* | *Test succeeded* |  |
| *Testing the Substitution of two bytes* | *Test succeeded* | *Once the SBox had been fixed up above here Sub\_Bytes worked fine* |
| *Testing the invertible linear transformation Mix\_Columns* | *Test succeeded* |  |
| *Testing shift rows* | *Test succeeded* |  |
| *Testing AES encryption* | *Test succeeded* |  |
| *Testing CBC AES encryption routines* | *Test failed* | *Initially when generating the IV I had a problem where the IV used for the encryption and not the MAC generation would crash because the value was larger than the 256 maximum allowed for a byte so I added a temporary integer which I could then divide by 255 and get the remainder with MOD and therefore the number would be small enough to fit in a byte* |
| *Testing CBC AES encryption routines* | *Test succeeded* |  |
| *Testing CBC MAC generation* | *Test failed* | *The test succeded fine but the returned byte array was too large for the single cipher, meaning an excess of zeros were added to the end, therefore I redimensioned the array to make it 16 bytes long* |
| *Testing CBC MAC generation* | *Test Succeeded* |  |
| *Tesing Remove Zeros* | *Test failed* | *I forgot to add the -=1 for the counter so it got stuck in an infinite loop* |
| *Testing Remove Zeros* | *Test succeeded* |  |
| *Testing inverse of SBox* | *Test Succeeded* |  |
| *Testing inverse of Mix Columns* | *Test Succeeded* |  |
| *Testing the inverse of Shift Rows* | *Test succeeded* |  |
| *Testing AES Decryption* | *Test succeeded* |  |
| *Testing AES CBC Decryption* | *Test succeeded* |  |

#### Testing ECDH

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Testing generation of keys* | *Test succeeded* |  |
| *Testing the generation of keys where the domain parameters are invalid* | *Test succeeded* | *Due to the use of validate parameters routine it caught the incorrect domain parameters* |
| *Testing regeneration of public key* | *Test succeeded* |  |
| *Testing regeneration of public key with incorrect domain parrameters* | *Test succeeded* |  |
| *Testing regenerationg of public key with a null private key* | *Test failed* | *There was no check for a null private key, added check for a null private key* |
| *Testing generate shared secret* | *Test succeeded* |  |
| *Testing generate shared secret where the public key was a point on another curve* | *Test succeeded* | *The problem was caught by checking if the point was on the curve provided by the domain parameters* |
| *Testing generating a shared secret where the given curve parameters are not correct* | *Test Succeeded* | *The problem was caught by validation of the domain parameters.* |

#### Testing ECIES

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Testing the key deriviation function* | *Test succeeded* |  |
| *Testing the insertion of a byte* | *Test succeeded* |  |
| *Testing is array zeroed* | *Test succeeded* |  |
| *Testing compare arrays* | *Test succeeded* |  |
| *Testing ECIES encryption* | *Test failed* | *I used insert\_byte to insert the length of the ciphertext at the end of the cipher text, this would crash if the ciphertext’s length was longer than 255 which is the max allowed for a byte, instead I created a function that inserted the length at the end of the array* |
| *Testing ECIES encryption with the new insertion method* | *Test succeeded* |  |
| *Test ECIES encryption with a point that is not on the curve* | *Test succeeded* | *The function writes out an error to the debug log and the return sets the didError value in the structure* |
| *Test ECIES encryption with an invalid curve* | *Test succeeded* | *The function writes out an error to the debug log and the return sets the didError value in the structure* |
| *Test ECIES encryption with a point that is not on the curve* | *Test succeeded* | *The function writes out an error to the debug log and the return sets the didError value in the structure* |
| *Test ECIES with null points or points that exist at infinity* | *Test succeeded* | *The function writes out an error to the debug log and the return sets the didError value in the structure* |
| *Testing ECIES decryption* | *Test failed* | *This test failed in the same way as it did with encryption so I used the new procedure to replace the now defunct insert byte* |
| *Testing ECIES decryption with the new length insertion method* | *Test succeeded* |  |
| *Testing ECIES decryption with an incorrect MAC* | *Test succeeded* | *The function writes out an error to the debug log and returns nothing in the decrypted byte array* |
| *Testing ECIES decryption with a public key that is erroneous* | *Test succeeded* | *The function writes out an error to the debug log and returns nothing in the decrypted byte array* |
| *Test ECIES decryption with a public key that is not on the curve* | *Test succeeded* | *The function writes out an error to the debug log and returns nothing in the decrypted byte array* |
| *Test ECIES decryption with a local public key that is not a point on the give curve* | *Test succeeded* | *The function writes out an error to the debug log and returns nothing in the decrypted byte array* |
| *Test ECIED decryption with points that are infinity* | *Test succeeded* | *The function writes out an error to the debug log and returns nothing in the decrypted byte array* |

### Database Tools

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Testing the instanciating of the class with an existing client database* | *Test succeeded* |  |
| *Testing the instanciating of the class with an existing server database* | *Test succeeded* |  |
| *Testing the instanciating of the class without an existing client database* | *Test failed* | *We were unbale to run SQL queries with the file created by File.Create methods instead we had to use the inbuilt ADOX Catalog in order to create a new database with the Jet engine* |
| *Testing the instanciating of the class without an existing client database* | *Test Succeeded* |  |
| *Testing the instanciating of the class without an existing server database* | *Test succeeded* | *After having the new routine to create a database file the test worked fine* |
| *Testing SQL non-queries* | *Test succeeded* |  |
| *Testing SQL queries* | *Test succeeded* |  |

### Server Unit Testing

#### Testing the client lookup table class

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Testing adding a new client* | *Test succeeded* |  |
| *Testing removing a client* | *Test succeeded* |  |
| *Testing adding a client that already exitsts* | *Test succeeded* |  |
| *Test removing a client that doesn’t exist* | *Test succeeded* |  |
| *Test checking a client exists* | *Test succeeded* |  |
| *Testing getting details of a client* | *Test succeeded* |  |
| *Testing getting the details of a client who doesn’t exist* | *Test succeeded* |  |
| *Test getting the public key of a client who exists* | *Test succeeded* |  |
| *Test getting the public key of a client who is not connected* | *Test succeeded* |  |
| *Test getting the username of client who exists* | *Test succeeded* |  |
| *Test getting the username of a client who doesn’t exist* | *Test succeeded* |  |
| *Test getting the client connection reference of a client who exists* | *Test succeeded* |  |
| *Test getting the clients connection reference of a client who doesn’t exist* | *Test succeeded* |  |
| *Test close all conection when there are no connections* | *Test succeeded* |  |
| *Test close all connections when connections exist* | *Test succeeded* |  |
| *Test get the list of connected clients when none exist* | *Test succeeded* |  |
| *Test get the list of connected clients when there are clients connected* | *Test succeeded* |  |
| *Test resolve a username to IP when the username does not exist* | *Test succeeded* |  |
| *Test resolve a username to IP when the username exists* | *Test succeeded* |  |

#### Testing the client heartbeat

|  |  |  |
| --- | --- | --- |
| Test Scenario | *Test Outcome* | *Notes* |
| *Check that the timer fire is caught when 2 minutes is expired* | *Test succeeded* |  |
| *Check that the timer fire is caught when a multiple of 2 minutes expires* | *Test succeeded* |  |

#### Testing New TCP Listener

|  |  |  |
| --- | --- | --- |
| *Test scenario* | *Test outcome* | *Notes* |
| *Test that the thread is started when startClientPoll is called* | *Test succeeded* |  |
| *Test that the thread is stopped when stopListener is called* | *Test succeeded* |  |
| *Test that the event is raised when a new connection occurs* | *Test succeeded* |  |

#### Testing the UI Updater

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Test that the UI is updated when writeInstance is called and the instance has been set* | *Test succeeded* |  |
| *Test that the UI fails to update when the instance is not set and writeInstance is called* | *Test succeeded* | *The subroutine returns an error suggesting that the instance has not been set* |
| *Set the instance of the main form* | *Test succeeded* |  |
| *Test adding the client to the data grid view when the forms instance has been set* | *Test succeeded* |  |
| *Test adding a client to the data grid view when the forms instance has not been set* | *Test succeeded* | *The subroutine returns an error suggesting that the instance has not been set* |
| *Test removing a client from the data grid view when that client does not exist* | *Test succeeded* |  |
| *Test removing a client from the data grid view when that client does exist but the instance of the form has not been set* | *Test succeeded* | *The subroutine returns an error suggesting that the instance has not been set* |
| *Test removing a client from the data grid view when the client exists and the form instance has been set* | *Test succeeded* |  |
| *Test get the count of users in the data grid view* | *Test succeeded* |  |
| *Test get the count of users in the data grid view when the instance has not been set* | *Test succeeded* | *The subroutine returns an error suggesting that the instance has not been set* |

#### Testing the Client Send and Receive Function

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Test creating a new instance where the client connection has been closed* | *Test failed* | *The subroutine never checks if the TcpClient has been disposed of or if the stream is no longer open* |
| *Test creating a new instance where the client connection has been closed* | *Test succeeded* | *Added a new check to check if client.Connected and if that returns false then we throw an exception, stopping the instance from being created* |
| *Test calling sendData when the data field is null* | *Test failed* | *There was no check for null data therefore when the copying starts an error is thrown* |
| *Test calling sendData when the data field is null* | *Test succeeded* | *Added checking to make sure that the provided parameter was not null, but when running its not possible that this parameter could be null* |
| *Test calling the send function when there is no client connection* | *Test succeeded* | *The function catches the error when trying to write to the stream and calls close client connection which removes the client from memory and removes references to it from the lookup table* |
| *Testing sendError* | *Test succeeded* |  |
| *Testing sendACK* | *Test succeeded* |  |
| *Testing NoHearbeat* | *Test succeeded* |  |
| *Testing closeClientConnection* | *Test succeeded* |  |

#### Testing SocketMT

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Test starting TcpListener* | *Test succeeded* |  |
| *Test starting Tcp Listener when the port is equal to 0* | *Test failed* | *There was no check to ensure that the port was not equal to 0 or the maximum allowed which is 65535* |
| *Test starting Tcp Listener with a null IP address* | *Test failed* | *Added a check to ensure that the provided LocalIp to start the server on was not null* |
| *Test clientConnecting* | *Test succeeded* |  |
| *Test clientConnecting but closing the stream half way through* | *Test failed* | *There was no error catching around the stream functions and it caused the program to crash. Added try catch around send and receive functions* |
| *Test sending unsent data and providing a closed network stream* | *Test failed* | *There was no test to see if the stream was closed and no error catching around the write functions* |
| *Test removing a client when there was no client connected* | *Test succeeded* | *A message was written to the server’s log indicating it tried to remove a client that did not exist* |
| *Test removing a client* | *Test succeeded* |  |
| *Test processing client data* | *Test failed* | *I had an reoccurring issue where I had forgotten to add -1 for array dimensioning and where it returns an index I had forgotten to add +1 to array.copy and array.constrainedCopy functions* |
| *Test processing client data* | *Test failed* | *There was a problem with a few of the SQL statements where I had omitted an apostrophe to text so the databaseTools would crash the program* |
| *Test processing client data* | *Test failed* | *There was also another problem where if the first packet of the file was not received correctly and it was thrown away and thusly the filestream was not created when we can to write the data out to the filestream it would crash, so adding checks around the filestream.write function stopped this* |
| *Test processing client data* | *Test succeeded* |  |
| *Test processing client data when the provided data was zero* | *Test failed* | *Ther was no checks to check the header of the data so when it was all set to zero then the converting functions would fail and database commands would fail as there was no provided data and the field could not be null* |
| *Test calling stop to the function when the function was not started* | *Test succeeded* | *There are checks to check if any of the variable are equal to nothing* |
| *Test converting IP to decimal notated* | *Test succeeded* |  |
| *Test converting IP to decimal notated when the IP was nothing* | *Test succeeded* | *There was an if statement catching any null values* |
| *Test conveting IP to decimal notated when the IP was a MAC address* | *Test succeeded* | *It was caught when the split function doesn’t split it into four different strings* |

### Unit Testing the Client

#### Encryption Registry Tools

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Test setting a registry value* | *Test succeeded* |  |
| *Test setting a registry value with either a null username or a null value* | *Test failed* | *There is no check for null values but it shouldn’t be possible for this situation to occur when the program is running* |
| *Test retrieving a value for an existing username* | *Test succeeded* |  |
| *Test retrieving a value for a non-existing username* | *Test succeeded* | *The program automatically returns null if nothing exists* |

#### Testing the UI Updater

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Test setting the form instance* | *Test succeeded* |  |
| *Test adding the user to the list* | *Test succeeded* |  |
| *Test removing a user from the list who does not exist* | *Test succeeded* | *As it uses a for loop system to find the user, if the username can not be found then the procedure just exists* |
| *Test removing a user from the list* | *Test succeeded* |  |
| *Test changing the list icon* | *Test succeeded* |  |
| *Test changing the list icon to one that doesn’t exist* | *Test failed* | *There was no routine to check if that image existed at that index in the image list* |
| *Test check and add a user* | *Test succeeded* |  |

#### Testing Hearbeat Timer

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Testing that the timer fires after 1:30 minutes* | *Test succeeded* |  |
| *Test that the timer fires after 3:00 minutes* | *Test succeeded* | *This was to prove that the timer was reset correctly after the first fire, if this test succeeded then it should be fine for any other multiples of 1 minute 30* |
| *Test writing to a stream that has been closed when the timer fires* | *Test succeeded* | *There is a try catch to help prevent any errors being thrown with the stream, this then raises an IOException event that SendRecieve can deal with.* |

#### Testing the user lookup table

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Test add a client* | *Test Succeeded* |  |
| *Test adding a client who already exists* | *Test succeeded* | *There is a check to ensure that the username is not in the table already* |
| *Test removing a client* | *Test succeeded* |  |
| *Test removing a client who doesn’t exist* | *Test succeeded* | *There is also a check to make sure that the username exists* |
| *Test getting a client’s public key* | *Test succeeded* |  |
| *Test getting a client’s public key when the client doesn’t exist* | *Test succeeded* | *It retuns a null value* |
| *Test updating a client’s public key* | *Test succeeded* |  |
| *Test updating a client’s public key when the client does not exist* | *Test succeeded* | *Returns a value saying the client could not be found* |

#### Testing the Send and Receiving function

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Testing the new instance of SendRecieve* | *Test succeeded* |  |
| *Test a new instance of SendRecieve when the provided TcpClient reference is not connected* | *Test failed* | *There was no check to ensure that the client was connected before proceeding* |
| *Test closing the connection half way through connection with the server* | *Test failed* | *Much like the server there are no try catches to catch any sending or receiving errors to prevent the client from crashing* |
| *Test passing a username of length zero* | *Test failed* | *The conversion from string to bytes fails as there is no check to ensure that the username is not null, instead the check is performed when submitting the username* |
| *Test receiving unsent files* | *Test succeeded* |  |
| *Test receiving unsent files when the stream is closed* | *Test failed* | *There are no try catch statements surrounding the read statements so an error is thrown and the program crashes* |
| *Test sending data to the server* | *Test succeeded* |  |
| *Test sending data to the server when the data is null* | *Test failed* | *There is no check to ensure that the data is not null when being called* |
| *Test sending an Error reply* | *Test succeeded* |  |
| *Test sending an ACK reply* | *Test succeeded* |  |
| *Test closing the connection* | *Test succeeded* |  |
| *Test closing the connection when the connection is already closed* | *Test succeeded* |  |
| *Test sending a message* | *Test failed* | *When the encrypted data was origionally gathered the MAC byte array was defined for 32 bytes so when the hash was calculated for the data, the server had a problem regenerating the same hash as the zero bytes at the end of the MAC array were removed by the server* |
| *Test sending a message* | *Test succeeded* |  |
| *Test sending a file* | *Test succeeded* |  |
| *Test sending a new user* | *Test succeeded* |  |
| *Test updating a user’s password* | *Test succeeded* |  |
| *Test updating a user’s rights* | *Test succeeded* |  |
| *Test removing a user* | *Test succeeded* |  |
| *Test syncing the form with the database* | *Test succeeded* |  |

### Testing the UIs

The tests for the UIs will simply be to break them, the UIs are very simple so there should be no test to indicate they work as usual, if there are errors found the test will fail and the notes section will detail the problem found.

#### Testing the Server UI

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Testing the starting the server with no selected address* | *Test succeeded* | *An error was written out to the log* |
| *Testing starting the server with a port that is equal to 0* | *Test succeeded* | *The error was written to the log* |
| *Testing starting the server with a port that is not numeric* | *Test succeeded* | *The log was updated with the error* |
| *Testing starting the server with a port that is larger than the max allowed* | *Test succeeded* | *The log was updated with the error* |
| *Closing the form when the server is not running* | *Test succeeded* |  |
| *Trying to stop the server when its not running* | *Test succeeded* | *Its not possible due to the stop button being greyed out while the server is stopped* |
| *Trying to start the server when the server is already running* | *Test succeeded* | *Its not possible due to the start button being greyed out while the server is stopped* |

#### Testing the Client UI

|  |  |  |
| --- | --- | --- |
| *Test Scenario* | *Test Outcome* | *Notes* |
| *Test trying to log in by providing no credentials* | *Test succeeded* | *A message box tells the user that the username or password field is empty* |
| *Test providing an incorrect username or password* | *Test succeeded* | *An error message was generated saying that the server replied with an incorrect password* |
| *Test loggin in as a user who is already logged in* | *Test succeeded* | *A message is returned from the server saying the user is already logged in* |
| *Test trying to send a message while not logged in* | *Test succeeded* | *The user was notified that either they were not connected or they had no user selected to chat to* |
| *Selecting a user when not logged in* | *Test failed* | *The program crashed because an instance of database tools was not set so now there needs to be a check ensuring that the local user has been set* |
| *Selecting a user when not logged in* | *Test failed* | *Tried checking if srFunction was nothing and then tried checking if srFunction.databaseConnection was nothing and it crashed because the instance of srFunction was not set* |
| *Selecting a user when not logged in* | *Test succeeded* |  |
| *Test selecting no users* | *Test succeeded* | *The user is told to select a user by the program* |
| *Opening User settings when the user is not logged in* | *Test failed* | *The sub new tries calling srFunction.syncWithServer() and it crashes because the function has not been instanciated* |
| *Opening User settings when the user is not logged in* | *Test succeeded* | *The form view does not open until the user logs in* |
| *Closing the form while a user is not logged in* | *Test succeeded* |  |
| *Test Connecting to the server when it is not running* | *Test succeeded* | *An error message is presented saying the target machine is actively refusing connections* |
| *Test send a file to no users* | *Test succeeded* | *The user is told to pick another user from the list view* |
| *Test checking file history of no users* | *Test succeeded* | *The user is told to select a user from the list view* |
| *Test trying to add a new user as a non-admin* | *Test succeeded* | *The user is told that they do not have the required privilege* |
| *Test trying to change the password of another user as a non-admin* | *Test succeeded* | *The user is told that they do not have the required privilege* |
| *Test trying to change the privielage as a non-admin* | *Test succeeded* | *The user is told that they do not have the required privilege* |
| *Test trying to remove a user as a non-admin* | *Test succeeded* | *The user is told that they do not have the required privilege* |
| *Test providing no details when adding a new user* | *Test succeeded* | *The prompt just closes and no other pormpts open* |
| *Test setting the privilege number of a new user as something greater than 2* | *Test failed* | *The user was added and the server shows the user as an administrator* |
| *Test setting the privilege number of a new user as something greater than 2* | *Test succeeded* |  |
| *Test adding a user with the same name as one already* | *Test succeeded* | *The server replies with an error that says the user already exists* |
| *Test resetting a password without providing a password* | *Test succeeded* | *The server is not updated with the password* |

# 

#### UI Screenshots:

Below shows the running server and client UIs, featuring an explanation of what is happening in the pictures as a proof to show the working solution.

##### Starting the server:

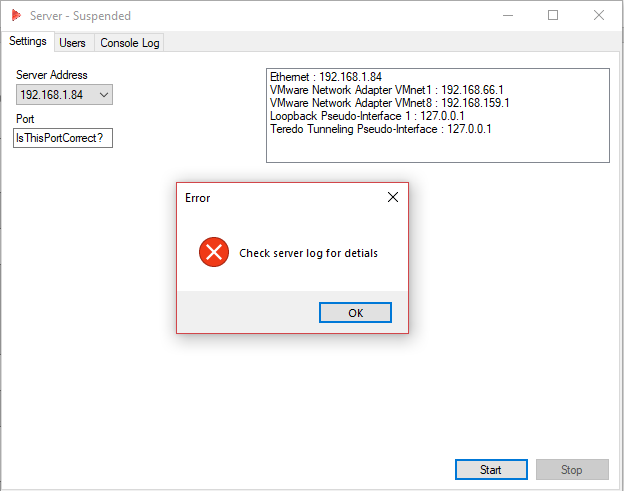


Figure 36:Server Error MsgBox

Above shows trying to enter an invalid port number for the server to function on, an error is logged to the console and the user is shown an exception message box that informs the user to check the log for error details.

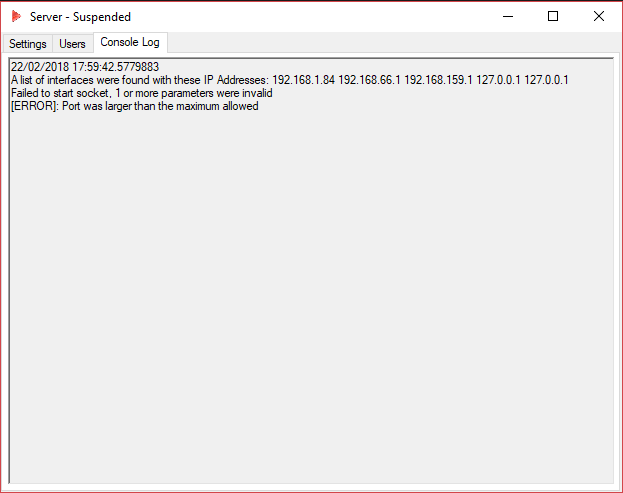


Figure 37:Server Errors

Here shows a screen capture of the server log when the server was first attempted to start with the port value = 0 and the second error where the port was equal to 70000 which is over the maximum.

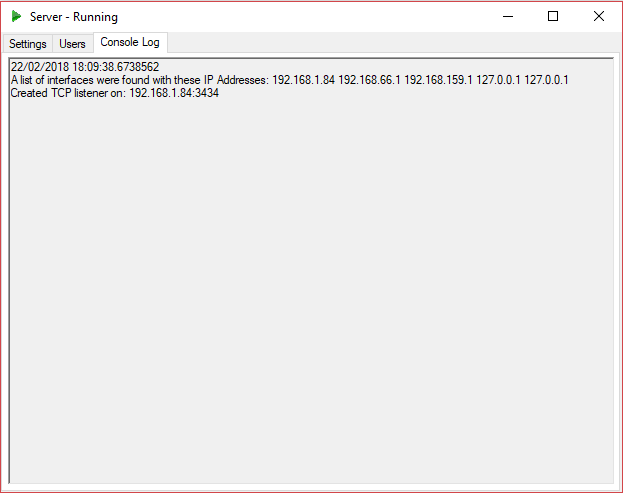


Figure 38: Server Starting

Starting the server normally on port 3434 with my internal IP address.

##### Connecting a Client

In order to connect a client when a new database is created we log in with the default values, username as Administrator and password as Admin1.

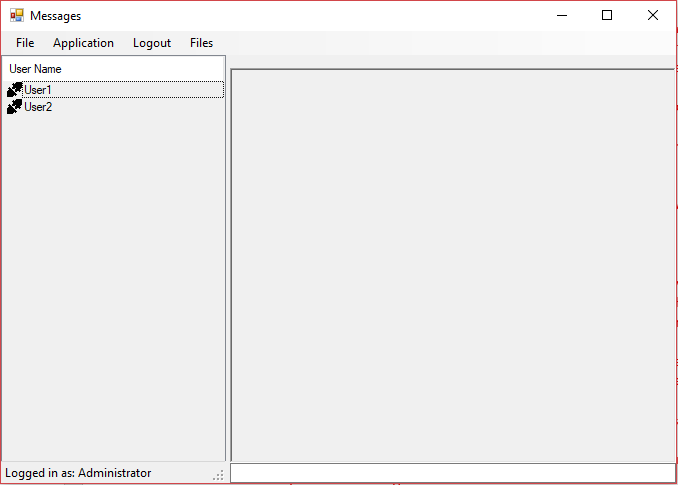


Figure 39: Successful Login

Above shows the results of a successful login, there are two known users that currently exists in the server database User1 and User2, but with a fresh server database then no users will be listed above until the user is created.

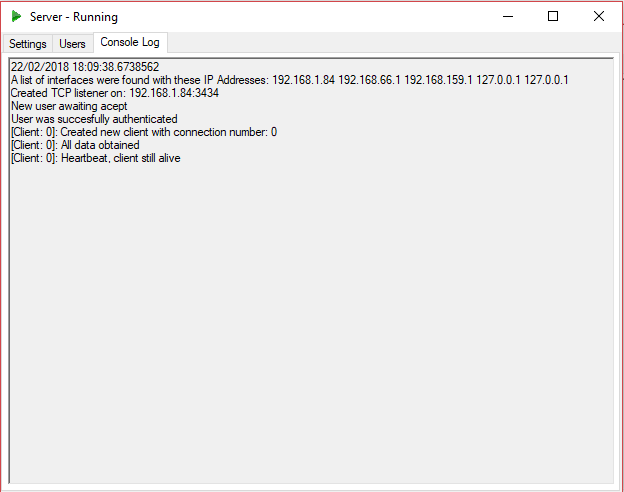


Figure 40: Server Log when Client Connects

Above shows the log from the server of a successful client connection and aswell the time inbetween taking screenshots must have passed the 1 minute 30 mark and therefore the client has sent a hearbeat to the server indicating it is still alive.

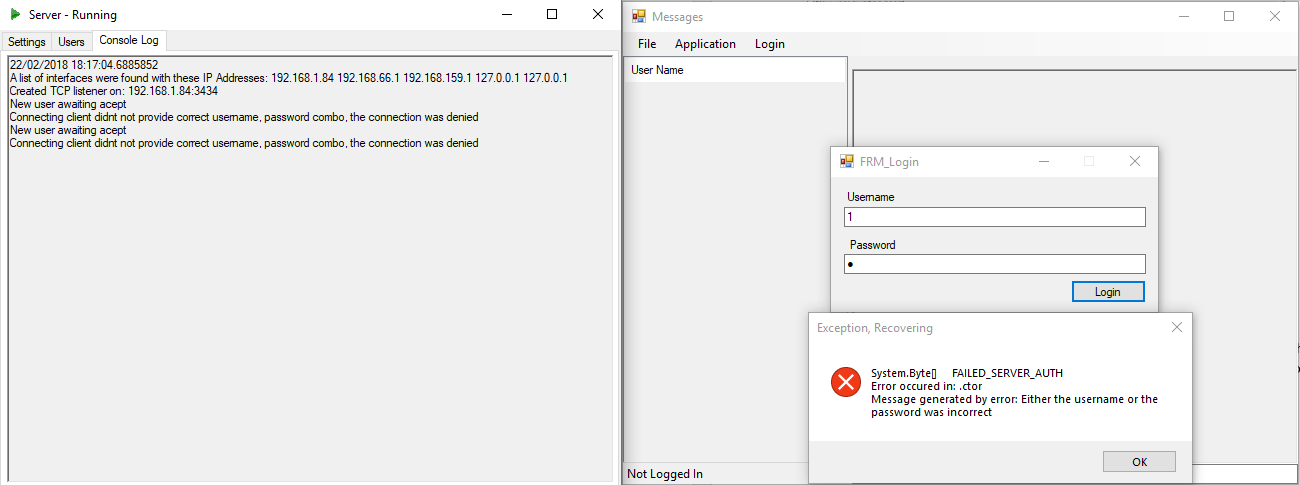


Figure 41: Failed password authentication

Above shows side by side the failed authentication for a user, a message is generated by the client in the form of a message box telling the client that there was an error with the authentication and in the server log you can see two failed login attempts from the user to the right.

##### User Management

Here is shown the process of an administrator being able to manage all of the users in the server’s database, initially with a new server database there would be only one entry, the default administrator, but for the purpose of the proof I added two more regular users, User1 and User2:

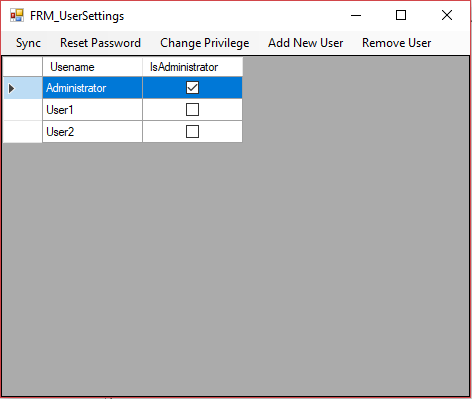


Figure 42: List of users in the server

Here is the server database to prove the users exist:

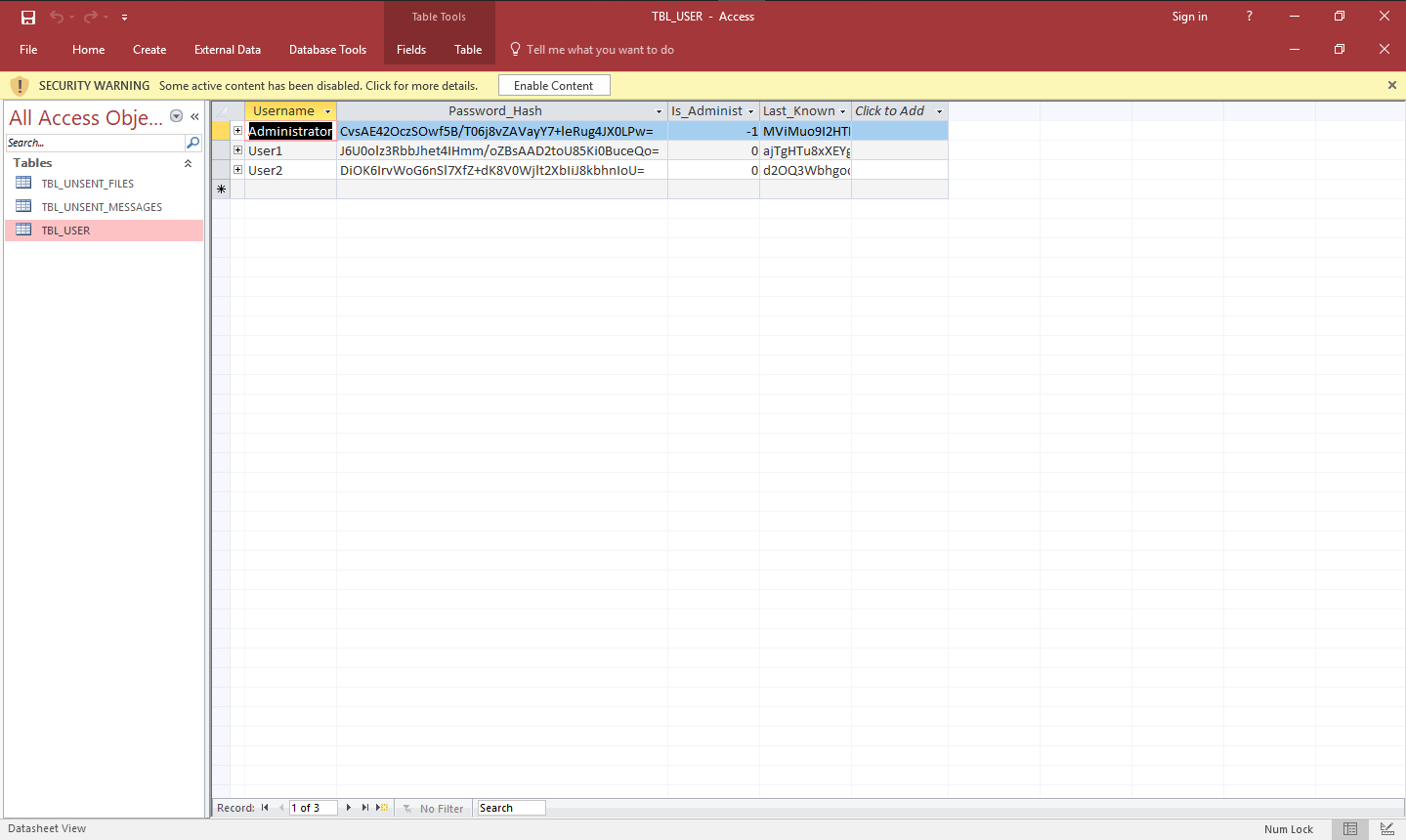


Figure 43: The server database

If we were to remove User2 then we simply click the remove user button and sync the form back with the server whilst connected as an administrator and the user should no longer appear:

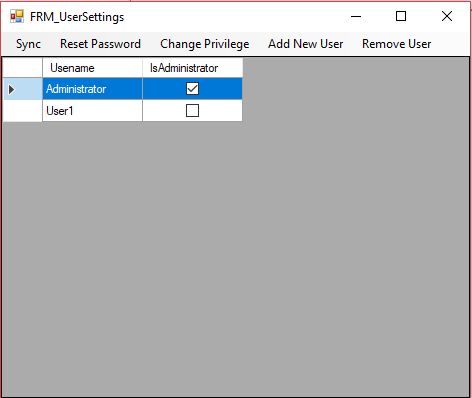


Figure 44: Removing a user

And if we wanted to add a new user called Admin2 with the password Admin2 and they were an administrator then we click Add New User, specifying those details and the ner user should be seen in the UI below:

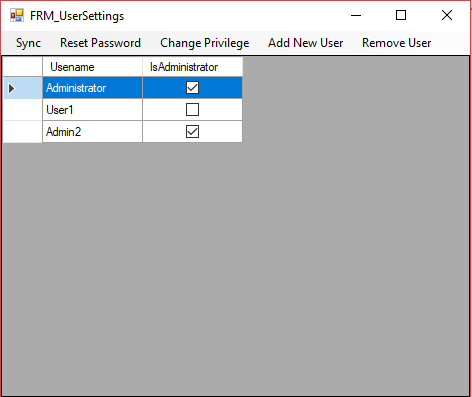


Figure 45: Adding a new user

Above you can see the same data that appears in the database in the server proved by the image below:

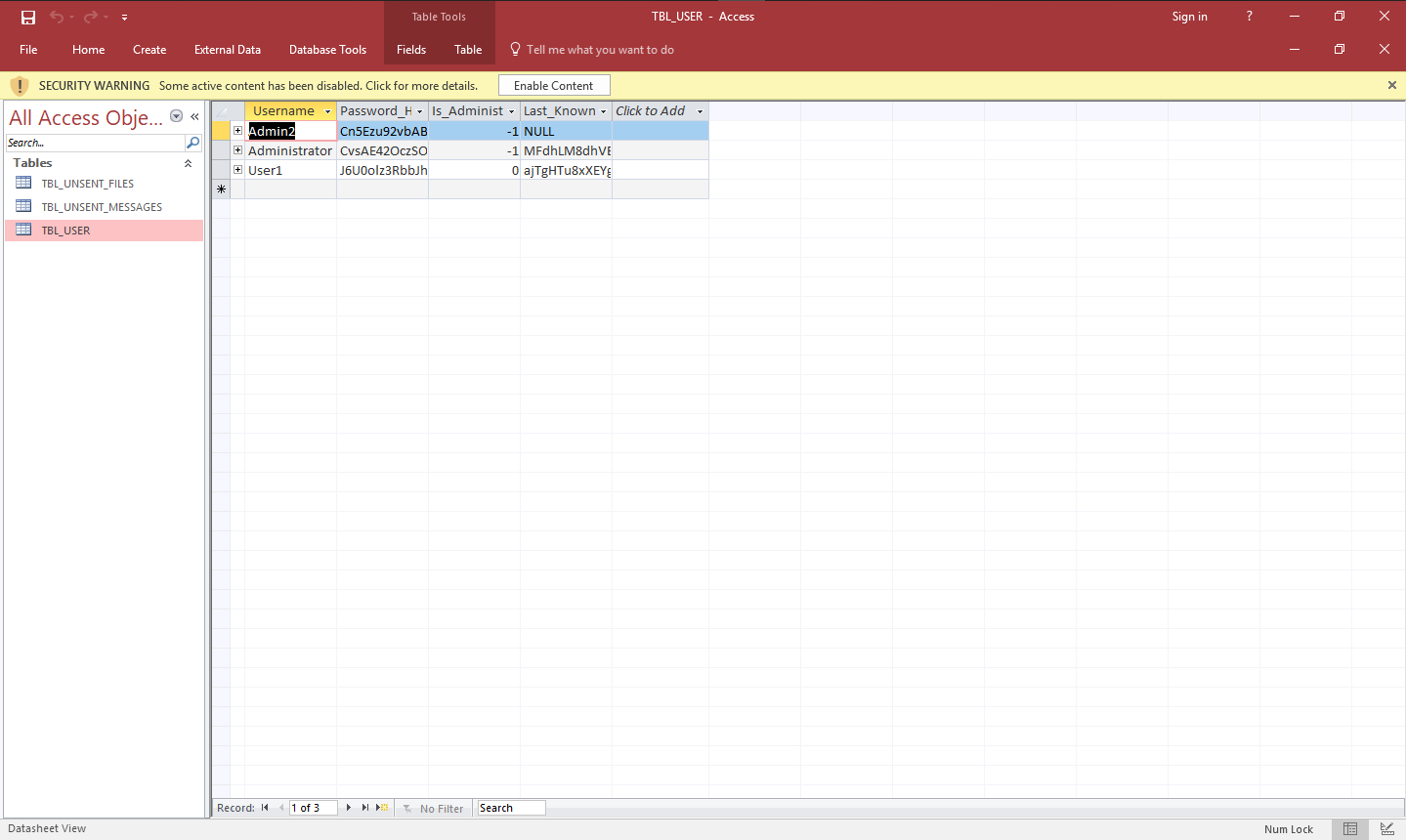


Figure 46: Corresponding Database

Here you can see the users Administrator, Admin2 and User1. Admin2 does not currently have a last known public key as the user has not logged on yet.

Lastly, we can change the role of Admin2 to a normal user in order to revoke their priveleges:

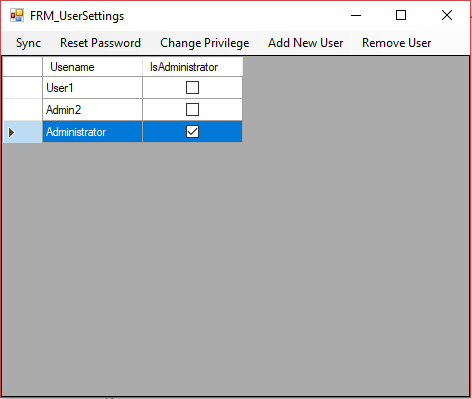


Figure 47: Admin2 priveleges changed

Say now for example Admin2 who is no longer an administrator logs in and tries to alter the clients in the database then an example of the message below would appear and no changes would be made to the database:

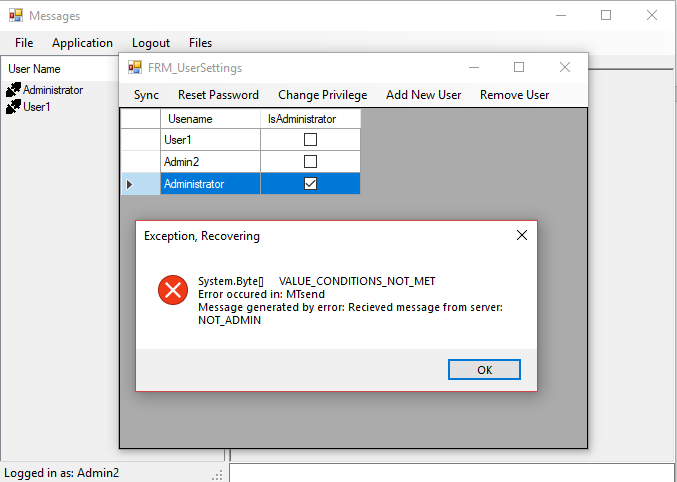


Figure 48: User is not an admin

##### Sending Messages

Messages can be sent when a user is either offline or online, as seen in the photo above down the left had side of the bottom window you can see the connection status of the other users, both being offline. When a user sends a message to them, for reference the image below, a message is displayed at the top of the messages box:

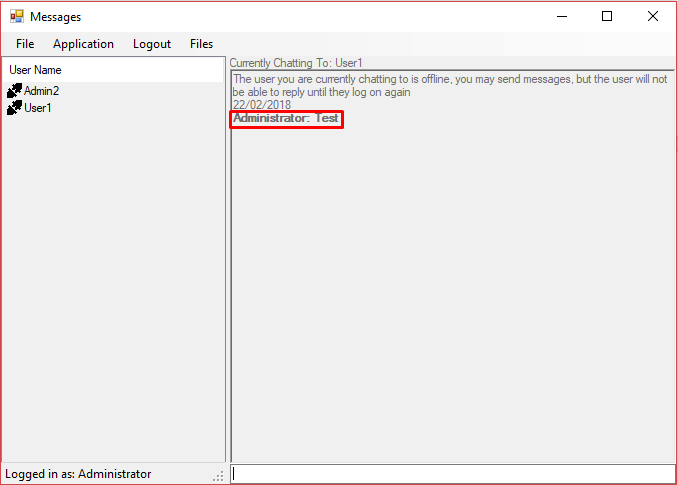


Figure 49:Offline message sending

As outline in the red box is the message to the user User1 that we sent while the user was offline, now logging into the users account there should be a notification for that message:

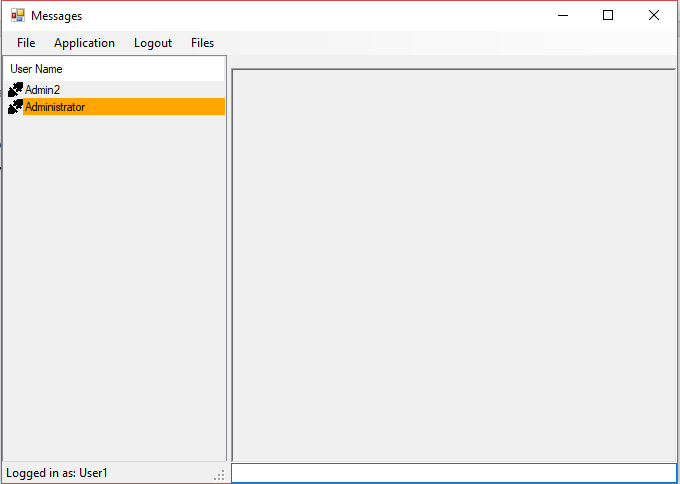


Figure 50: Notification

And upon clicking the username the message should be displayed on the right-hand side:

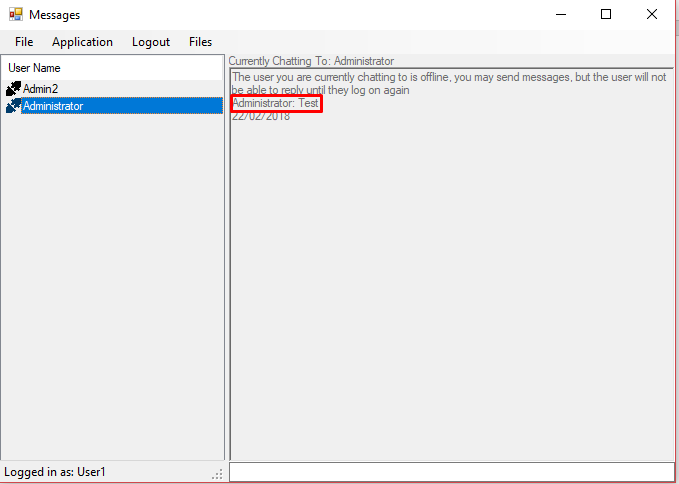


Figure 51: Message from the user

Taking a look at the server database in between the sending and receiving of the message we can see the base 64 encoded ciphertext of the message:

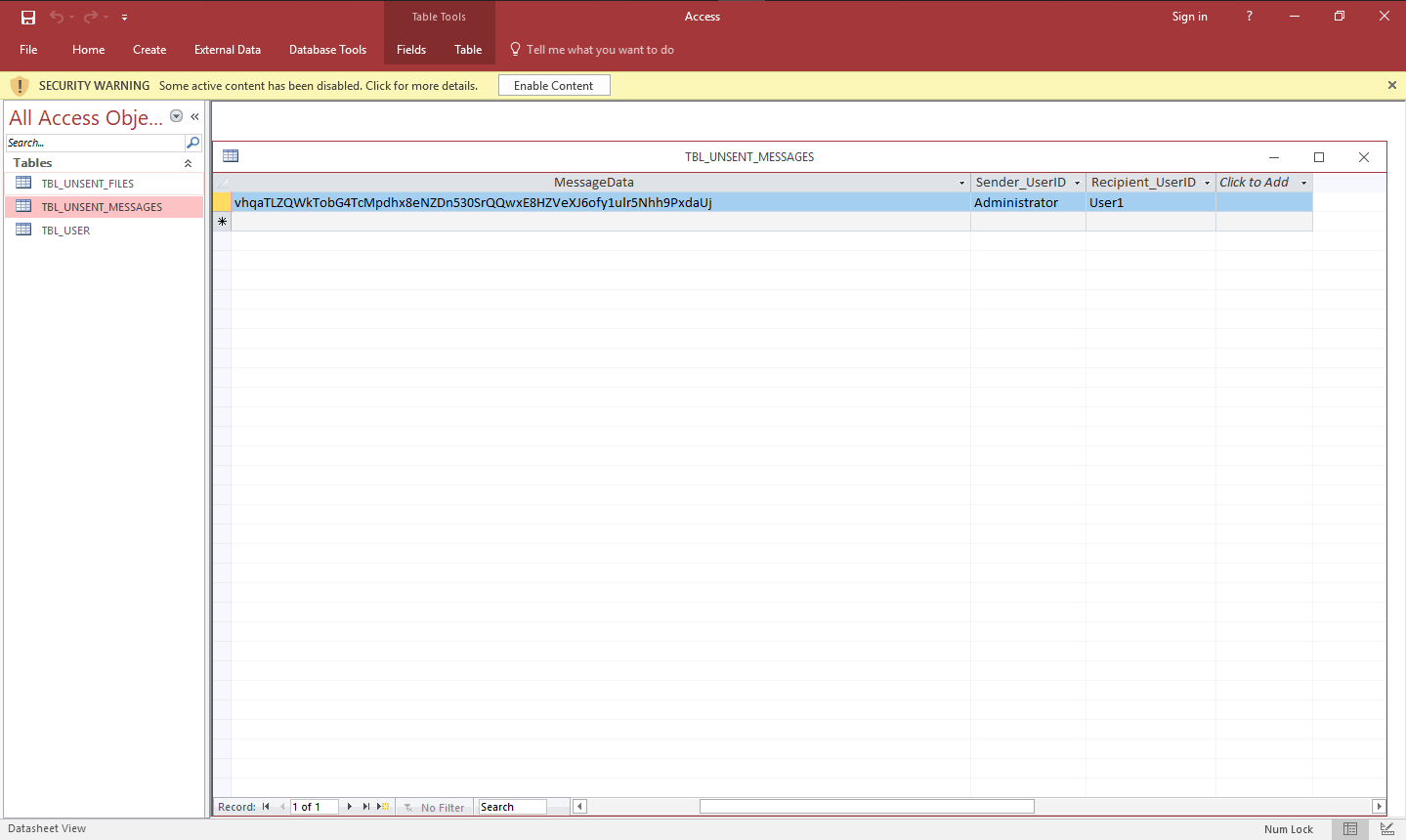


Figure 52: Encrypted message in server database

When sending a message online a user should first be connected to the server and it will be indicated by the icon below and the lack of the offline message at the top of the messages window:

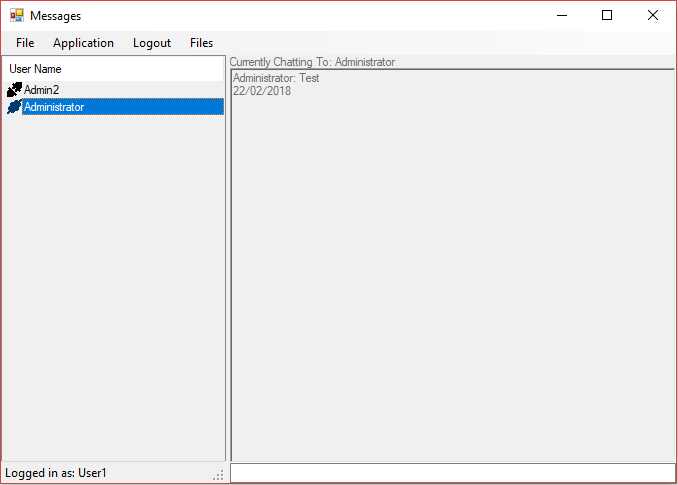


Figure 53: Connected user

By then sending a message to that user we will firstly see that the message sends fine and that on the other client the user should have a notification telling them that a new message has been received:

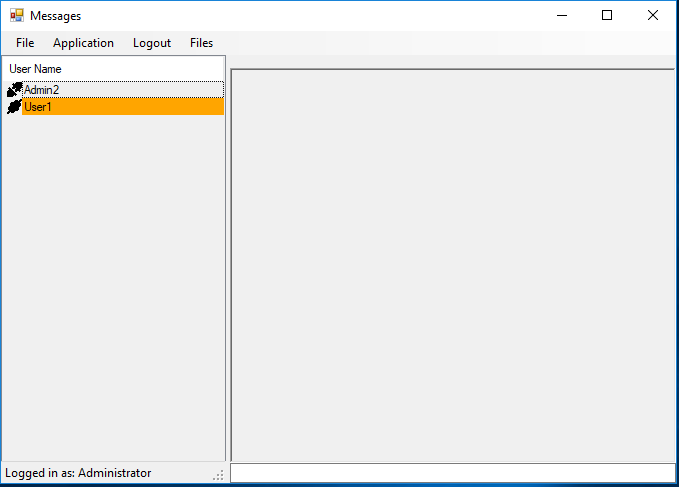
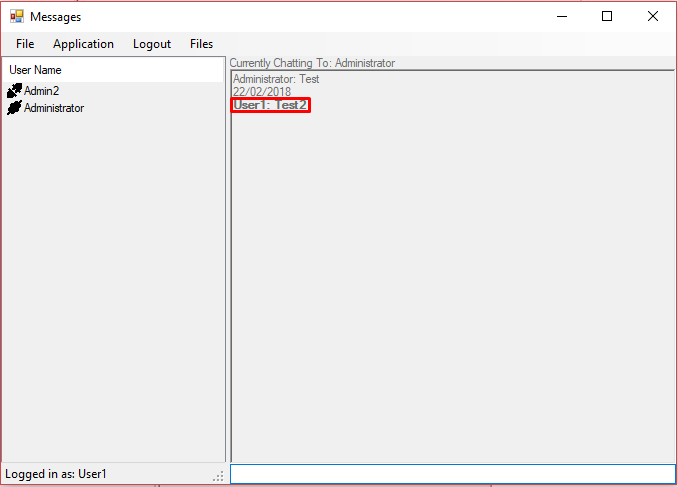


Figure 54: The two users messaging eachother

Upon clicking that user like before the notification is dismissed and the message can be seen decrypted:

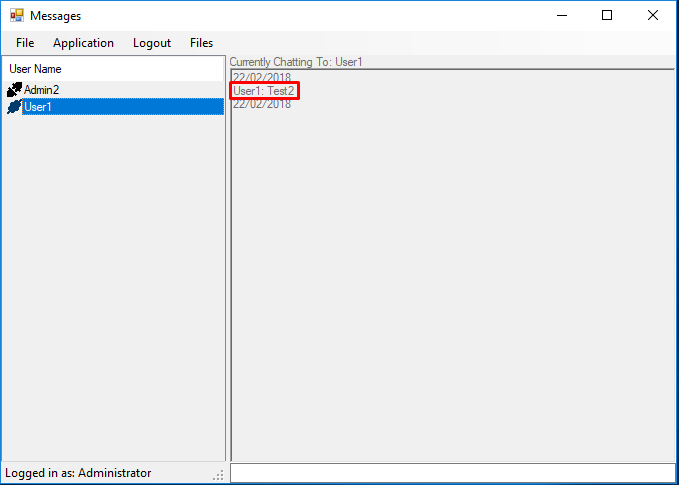


Figure 55: Plaintext message from User1

##### Sending Files

Just like sending messages files can also be sent offline from user to user and online too, in order to send a file and access the files sent offline the user uses the Files tab along the top of the main form. From the picture below, you can deduce that there has been no file sending history between them so, lets send them a text file.

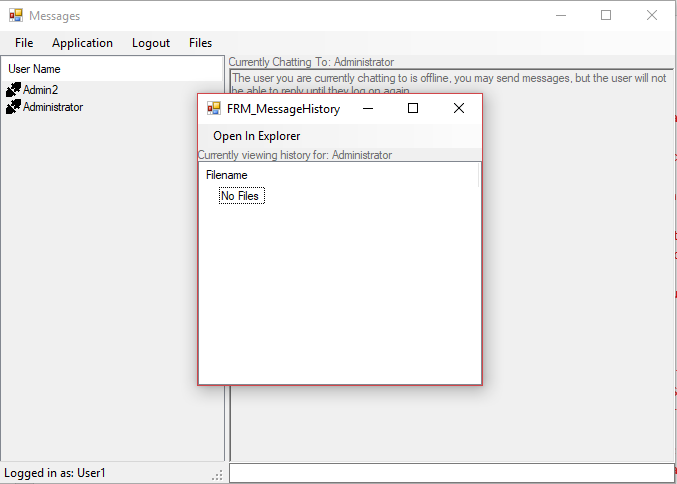


Figure 56: Files Tab

The text file we are sending contains a small list of components for a system that User1 wants the administrator to build:

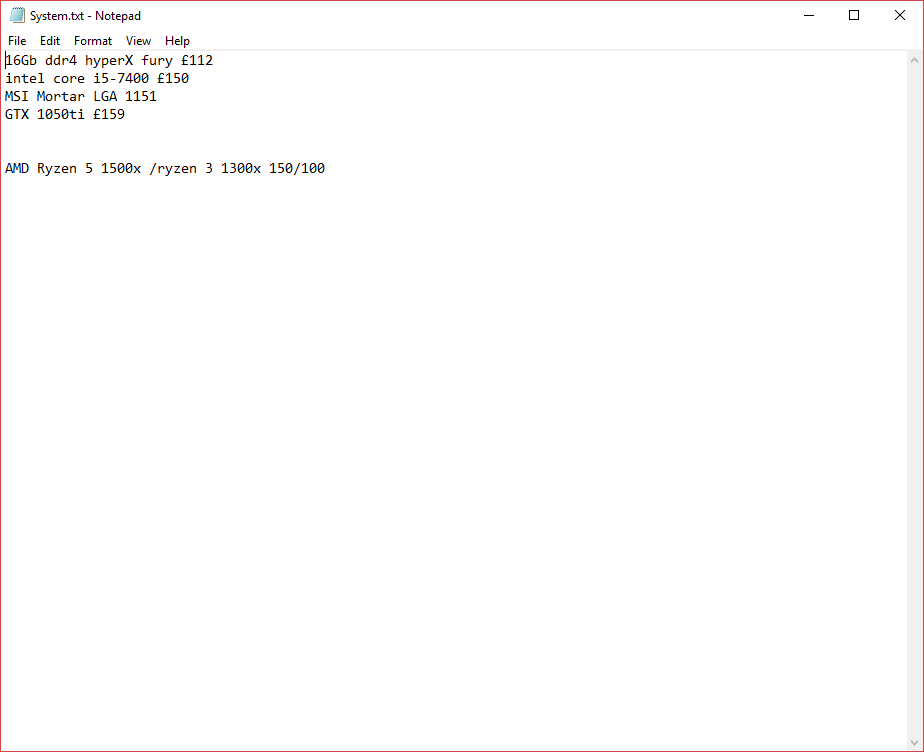


Figure 57:The text file to be sent

In order to send this file the user clicks on Files->Send and a file dialog box appears where they select the file to be sent to the user, now logging as the administrator there should be a light blue notification showing that they have received a file from the user:

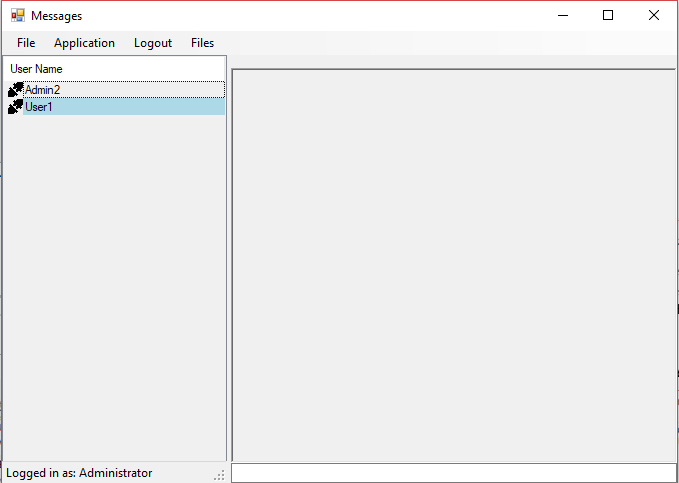


Figure 58: New file from User1

Thusly checking the file sending log we can see that the file has been received by the client and we can open that location in explorer and see the sent file in its original form:

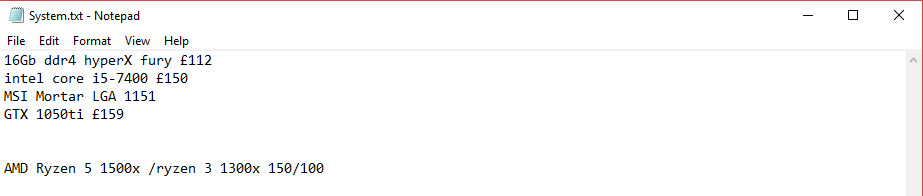
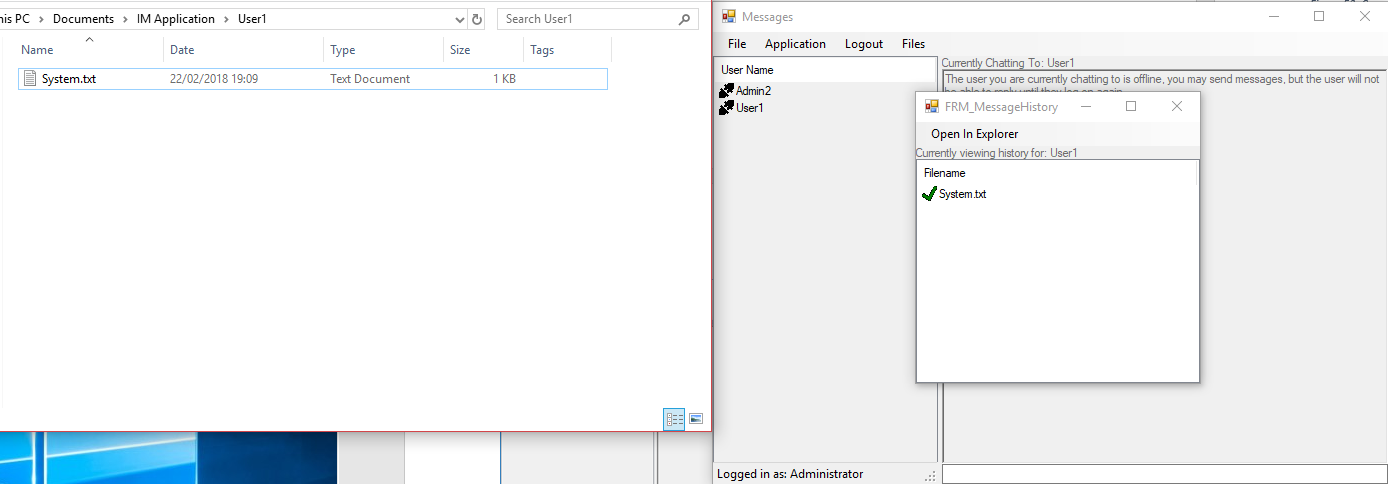


Figure 59: Recieved file from offline mode

And the encrypted version that was stored in the server’s temporary directory looks like:

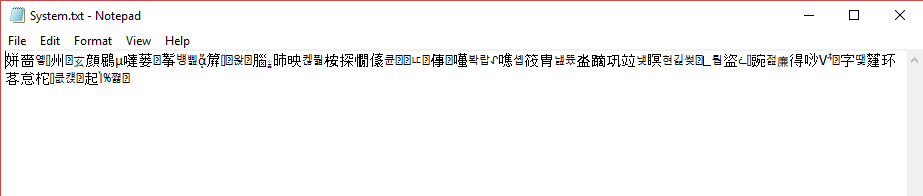


Figure 60: Encrypted File

Again, like in messages in order to send a file online the user will have to be logged on to the server to receive the message real time, once the user is connected and the file is sent the recipient will see some text highlighted in red when viewing the messages tab for that user:

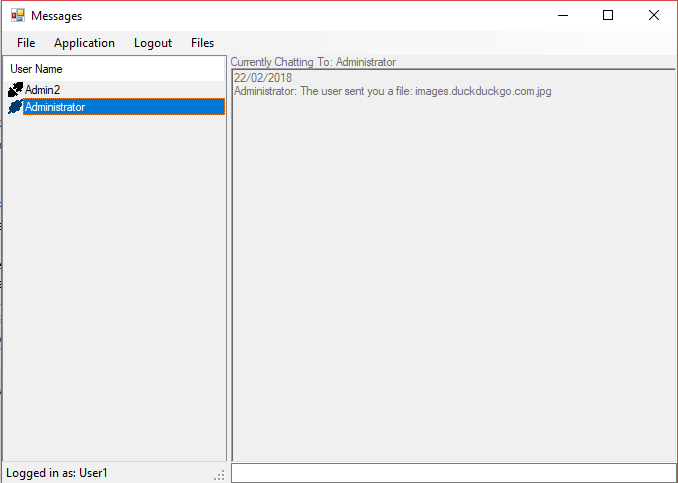


Figure 61: File notification

And by viewing the file history window we can see that the file being sent has been received by the client:

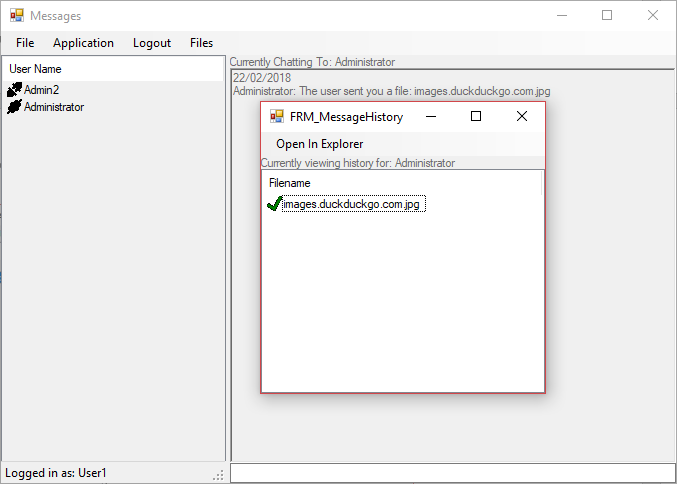


Figure 62: Recieved File

# Evaluation

I will be talking about how I met or didn’t meet the objectives below:

* TCP/IP
  + Get the receiving IP and MAC- This objective was not met as it seemed infeasible to get the server to send packets across the network identifying itself, the administrators should know the server’s IP address and therefore set each client to connect
  + Setup the stack- This was not required as it wasn’t feasible to write our own TCP client instead we used the one built into visual basic
  + Open a port to communicate- Created my own multithreaded enhancement of Tcp Listener for the server, had multi threaded send and receive functions for both the server and the client
  + Split the message into packets of X size- This was implemented but was actually the incorrect procedure to take with sending data, instead the data can be sent all together with the meta data at the start of the packet
* Database (MySQL, Postgres)
  + Install database software on computer if needed- This is not done by either the server or the client, it is assumed that the server and client already have MS Access installed and the Microsoft framework provides access to the JET engine for Access databases
  + Create a new database- This was required by both the client and the server; the server used a singular database whereas the client required multiple databases in one different for each user.
  + Password protect/encrypt the database- Encrypting the database was not possible with the encryption routines I had created, it would have meant writing new routine for encryption with a public and private key, and a way to safely store the private key in order to decrypt it.
  + Generic queries to lookup items in the database- These queries are used throughout the both the server and client project, the server uses most of the basic SQL queries available to the platform, INSERT, UPDATE, DELETE, CREATE, SELECT
  + Possibly authenticate users with a password database- This was done server side, the password hash and username are sent to the server and the server replies with errors or normal user or administrator
* Secure Messaging
  + Authenticate with a server, with a list of known users, local window user accounts or a Microsoft Active Directory server- We implemented authentication with the server where a list of know users is stored in the database, it would have been even more beneficial to have Microsoft Active Directory syncing but as a normal home user I have no access to a micrososft AD server.
  + Setup secure communication channels via signed RSA encrypted messages or via signed ECDH encrypted messages.
    - If using ECDH the domain parameters must be agreed on beforehand and computed public keys should be made available by both parties [1]- The domain parameters are hardcoded into the program but there it is possible for a future update to change that design. When logging in the user sends their public key to the server and the server makes that key available to all of the user who connect.
    - With RSA the public keys should also be transferred between the two clients
  + Send the messages to a server or local clients inside the network.
    - Open up a port and each client that connects uses something similar to a NAT table or ARP table so each user has a known MAC address or IP for their session, also enabling user identification from client to client.- Something very similar is implemented in both the client and the user side of the server, the server posseses a hash table of all of the currently connected users and the client has a hash table of all of the users in the server, making it possible to send offline messages.
* Encryption
  + Use of an applicable encryption scheme for how we are transmitting data- We used AES with the Cipher Block Chaining mode of operation to encrypt a message in a secure way using the public key of the recipient and the private key of the sender to produce a secret key.
  + Possibly use XOR with a key but this means sending the key over the same network so not really a secure encryption type
  + Use of RSA is a better idea yet the security of RSA means a greater bitlength for better security which might pose problems if in the future the program needs to be ported to mobile.
  + The best idea, although very complicated, is to use Elliptic Curve encryption. This form of encryption can use the same curve to not just generate keys to encrypt a message but also digitally sign any data- I decided in the end to use this encryption scheme to be able to have the most secure messaging service and making it easier for weaker CPUs to calculate
    - To use this form, both parties must agree on domain parameters. [1]- As said earlier the domain parameters are hard coded into the program but it could easily be adapted to send the name of the curve to each client upon connection
    - An algorithm needs to be implemented for point addition, point doubling, and scalar multiplication of an EC point. [1]- This is vital in order for elliptic curve encryption to function so the algorithims exist inside of the program shown under the documented design section, encryption.
    - Data structures and OOP should be used to create objects for these data structures- Both the server and the client use object orientated programming in order to make it possible for the whole project to work as a whole.
* Online and offline users
  + If we are using a dedicated message server, the server can send authenticated users, otherwise each logged on client sends a message similar to an ARP request but to save a DDOS affect it only requests when a message is typing or being sent- We are using the client server topology so therefore the server tells the clients when a user is connected or not therefore the ARP design was not needed. Due to a running server it is also possible for users to send messages offline and the server will store the encrypted message, the server never has access to the private keys of the users so there is no possible way for the server to decrypt the messages.
  + If a dedicated message server offline messages can be sent to and stored on the server until a user next logs on- The messages are received by the client when the connect to the server and their details have been verified by the server.
* Sending and receiving files
  + Some form of FTP sending- There is a possibility for file sending both online and offline, which is also encrypted the same as messages.
  + Sockets can also be used to send data from client to server
    - Send a request to open a socket session- There is a multithreaded poll that listenes for client connection requests, that request is forwarded to another class that then accepts the request and ensures that the sent details are correct.
    - Setup the client as a server- This was not needed as the server is a standalone application and this only applies to peer to peer instant messaging.
    - Get the sender to make a connection request to the port on the server- the socket on the server accepts the requests and the meta data for the session is transferred between the client and the server
    - The server grants it to one specific user, so some authentication is needed here- No two users with the same credentials are allowed to be logged on at the same time, due to the design with the lookup table and the use of the username in the database as a public key.
    - Send the file- In order to send a file to the server the user must already have a connection to the server.
    - Close the connection, save the data, kick the client off- the clients are kicked off and when the connection is closed by the server, so by either stopping the server, or the heartbeat of the client is not reqcieved.

### The User’s Thoughts and Concerns

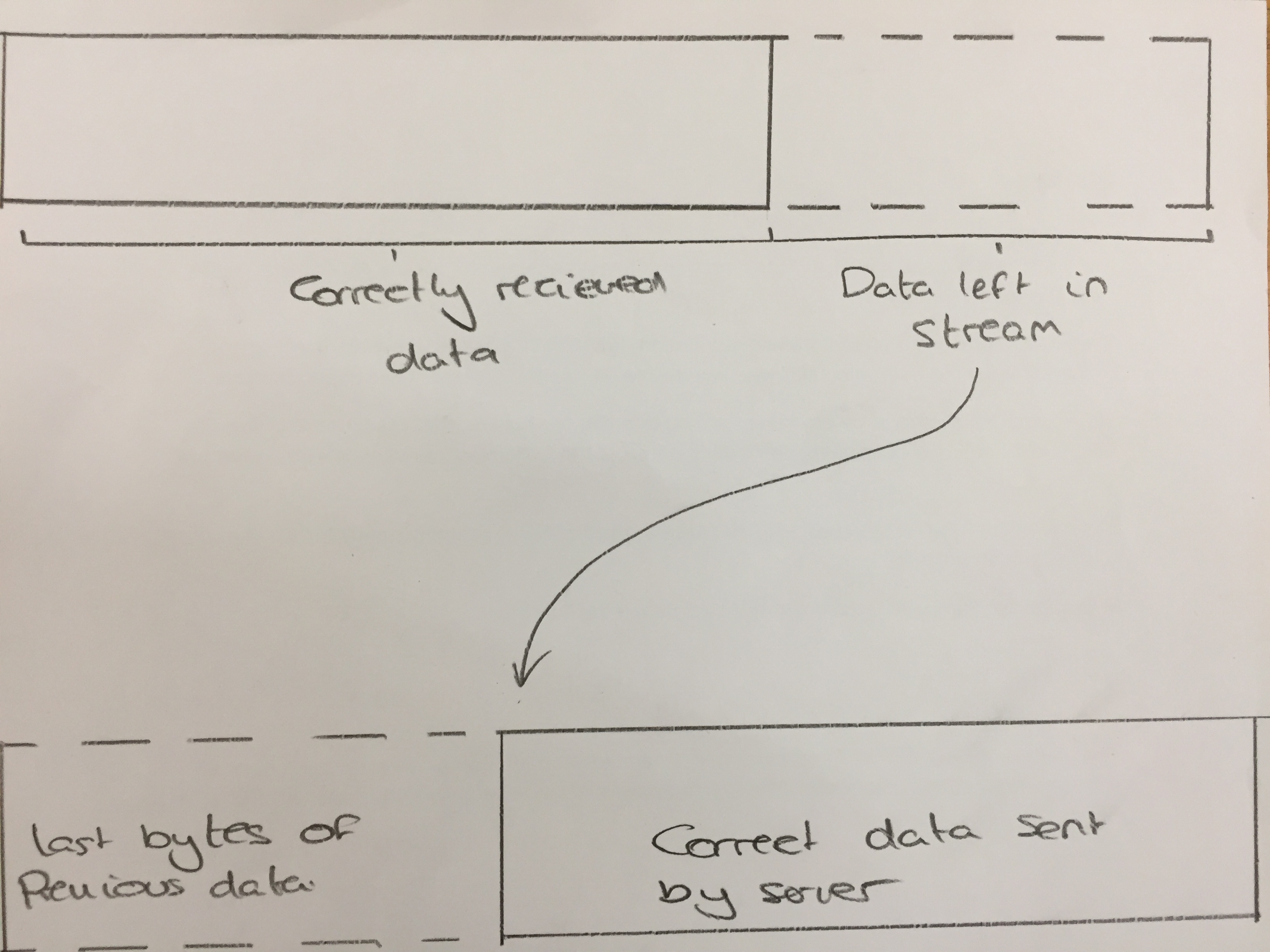
Upon seeing the application functioning, they acknowledged that in its present state the server and client feature a lot of bugs, yet the user was able to look past it. Their reviews of the UI were great, commenting that the UI had a clean design and that it was simple to user and with a barely any instruction was able to understand the basic features of the server. The user was happy with the sending and recieveing of messages and files and it was proved to them that they are encrypted by showing them the files in the server’s directory when the recipient was taken offline. The user was also supprised to see the notifications that was implemented in the connected user view and the simplicity of the connected icons for a connected and disconnected client. One comment was stated by the client that the files history tab wasn’t the best, and they would have preferred the tab to be on the same screen as the messages, where they could monitor it without having to enter another form each time. They were also impressed by the ability for a client to manage the server’s users from a client connection and liked the fact that the security checks were performed server side, and stateded that it was hard for an administrator to gain access to the server that the IM application’s server would be running on therefore being able to login as an administrator benefitted them greatly.

### Problems, Drawbacks and Design Changes

In terms of the whole project overall there are a lot of problems with both the server and the client on the TCP and databases side of code. Both could be strengthened a lot by firstly a rewrite of the server and the client message and file sending, as encryption works well it would not need to be touched, but the level of error checking with encryption needs to be present in the TCP and database side of the application. By rewriting the client and the server, the way files and messages are sent could be corrected to work by firstly sending the meta-data (the length of data about to be received, the public key of the sender, the username of the receiver), followed by the data where it all could be received in one go and thusly remove the need of opening filestreams and keeping them opened until the last packet was received. There are also some inherent issues with the offline message sending, I omitted from sending the user’s public key in the data as I thought it was not necessary, but in an work scenario where a user can log on multiple times, messages sent offline would not have the corresponding public key for the session where the user was logged on to send that message, therefore by adding the public key to the data, it can be extracted and the secret key regenerated easily and correctly. There is also a problem with the databases too, when a client opens the message history of a previous conversation all of the messages sent in that time appear to be from the recipient even though some are from the local user. It is not possible at the moment to change that design, another record would be needed in the client’s database, including the currently connecting client or allowing the field Sender\_UserID to be null when the local client sends the message.

Also, its not possible for a client at the moment to be able to decrypt messages or retrieve messages that were sent on previous occasions on another machine, the databases for the clients are not backed up to the server and the server should never be sent the plaintext or the private key to ensure complete anonymity. Therefore, the only choices would be to send an encrypted database to the server and require the client to carry around the private key with them on a usb flash drive or to send an encrypted form of the private key certificate to the server, which are both valid ideas but are also not a good solution to the problem.

Overall the main part of the project, the encryption, works flawlessly and to an extent so does databases, but due to problems with networking and TCP it causes crashes to occur in the databases side of things, but encryption seems to be stable. Files and messages can be sent offline and online, both versions encrypted and there is no possible way for a malicious user with access to just the server to decrypt these items. The server can be managed from a client machine with the client program and a client is able to see the message history and the file history. The server houses all of the authentication routines for the clients, and provides the clients with all of the details they need to send messages to one another. There are occasional problems with a connecting client, and I think the problem is due to TCP not reading in all of the data from the network stream, so occasionally a username is incorrectly received or the Boolean value indicating connection is sometimes not received by the stream when we read it in therefore a client appears offline to a user yet they are still able to communicate with the user via the server. In order to fix this its proposed to rewrite the connection routines to add the data length at the start of each section of data sent, therefore the client knows they have correctly read in all of the data from the network stream. A side effect of not reading in all of the sent data cause the receive unsent data to read in the unread data, corrupting the data that was sent by the server and therefore the server is unable to correctly decrypt it. A diagram shows the error happening below:



As you can see the top shows the total data of the first item sent by the server, as not all of the data was read in, some of the data is left in the stream upon arrival, meaning that when the data is read in by the server for the next item a ripple effect is created and that data is read in to the start of the array, offsetting the data that should be at the start, this then leaves data in the stream from the second sent item and this continues to happen until the client poll data is started and the last section of data is read in and thrown away as it is incorrectly configured. In order to fix this each sent packet should have the packet length at the start of the data as a 4-byte int32 or an 8-byte int64 therefore ensuring that all of the data was collected, preventing any of theses errors.

# Bibliography

|  |  |
| --- | --- |
| [1] | “Elliptic Curve Diffie Hellman,” 18 January 2012. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic\_curve\_Diffie%E2%80%93Hellman. |
| [2] | A. Corbellini, “Elliptic Curve Cryptography: a gentle introduction,” 17 May 2015. [Online]. Available: http://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/. |
| [3] | Wikipedia, “Elliptic Curve Point Multiplication,” 11 May 2012. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic\_curve\_point\_multiplication. |
| [4] | D. R. L. Brown, SEC 1: Elliptic Curve Cryptography, Certicom Corp, 2009. |
| [5] | Wikipedia, “Elliptic-curve cryptography,” 4 March 2005. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic-curve\_cryptography. |
| [6] | A. Corbellini, “Elliptic Curve Cryptography: ECDH and ECDSA,” 30 May 2015. [Online]. Available: http://andrea.corbellini.name/2015/05/30/elliptic-curve-cryptography-ecdh-and-ecdsa/. |
| [7] | Wikipedia, “Message authentication code,” 19 May 2006. [Online]. Available: https://en.wikipedia.org/wiki/Message\_authentication\_code. |
| [8] | A. Kirby, “How Elliptic Curve Cryptography encryption works,” 23 September 2015. [Online]. Available: https://www.nominet.uk/researchblog/how-elliptic-curve-cryptography-encryption-works/. |
| [9] | Wikipedia, “Block cipher mode of operation,” 14 November 2004. [Online]. Available: https://en.wikipedia.org/wiki/Block\_cipher\_mode\_of\_operation. |
| [10] | The McGraw-Hill Companies, Inc, “Advanced Encryption Standard,” [Online]. Available: http://www.cs.siue.edu/~tgamage/S17/CS490/L/WK05.pdf. [Accessed 09 2017]. |
| [11] | Wikipedia, “Rijndael S-Box,” 17 February 2008. [Online]. Available: https://en.wikipedia.org/wiki/Rijndael\_S-box. |
| [12] | Wikipedia, “Rijndael MixColumns,” 5 November 2012. [Online]. Available: https://en.wikipedia.org/wiki/Rijndael\_MixColumns#InverseMixColumns. |
| [13] | C. S. Exchange, “How to solve MixColumns,” 19 April 2012. [Online]. Available: https://crypto.stackexchange.com/questions/2402/how-to-solve-mixcolumns. |
| [14] | Wikipedia, “Rijndael key schedule,” 7 May 2009. [Online]. Available: https://en.wikipedia.org/wiki/Rijndael\_key\_schedule. |
| [15] | Wikipedia, “Advanced Encryption Standard,” 27 May 2004. [Online]. Available: https://en.wikipedia.org/wiki/Advanced\_Encryption\_Standard. |
| [16] | Wikipedia, “ECDSA,” 13 April 2008. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic\_Curve\_Digital\_Signature\_Algorithm. |
| [17] | Wikipedia, “HMAC,” 27 April 2006. [Online]. Available: https://en.wikipedia.org/wiki/Hash-based\_message\_authentication\_code. |
| [18] | Wikipedia, “CBC-MAC,” 26 September 2011. [Online]. Available: https://en.wikipedia.org/wiki/CBC-MAC. |
| [19] | Net-informations, “VB.NET MultiThreaded Server Socket,” [Online]. Available: http://vb.net-informations.com/communications/vb.net\_multithreaded\_server\_socket\_programming.htm. |