Implementation Details

This is the segment of my report that describes how the software works, if you are looking at my repo and are confused by anything, there's a good chance it's elaborated on in here (except view logic, which is not covered because it was not relevant to my dissertation goals).

Chapter 5
Implementation

In this chapter the chosen technologies are mentioned and there will be discussions about how features of the software work and show any relevant code extracts.

Towards the end of this chapter some structural diagrams are shown, as well as a brief description of classes that have not already been touched on. Finally there are a number of object sequence diagrams which illustrate what happens during the use cases of the application.

5.1 Technologies

This sections covers the technologies that were chosen for the implementation, if they are different than what was originally decided on then the reason switch will be explained.

5.1.1 Language

C#. In the project proposal python was originally chosen, but after doing some research C# seemed favourable for the following reasons:
C# has a very good, well documented IDE with many plugins which could be useful.
C# has a well-documented Networking library as well as multiple cryptography libraries (all published by Microsoft themselves)
C# provides an API for encrypting data in memory to protect it from memory scanning.
Visual studio allows the creation of GUI's in a much easier fashion than coding it all manually, which saved me a lot of time in implementation.

5.1.2 DBMS

SQLite was kept as the DBMS of choice as most computers will have it and it's a very lightweight and useful tool. C# also has many user libraries to interact with it.

5.1.3 Cryptography

"Cryptography API: Next generation (CNG)", a library made by Microsoft, was not used because it is fairly new and very poorly documented, instead the Microsoft.Security.Cryptography library was used for the implementation.

As was revealed by preliminary testing, the use of RSA alone was acceptable for the small amounts of data transferred by text messages, therefor symmetric encryption for message exchange was not necessary.

In terms of key storage, a Master-key like solution was chosen, however instead of the key being stored anywhere on the machine, it is derived from a password that the user must remember.

5.1.4 Network Address Tunnelling (NAT)

NAT punching tests were carried out and the results were described under preliminary testing, in short it was not feasible. Instead the application relies on the user port-forwarding to gets packets through firewalls. A minor inconvenience for most users.

5.1.5 Architectural Pattern

The originally planned architecture was implemented, however there was a misunderstanding of the implementation of the MVC (model-view-controller) design pattern. In reality what ended up being implemented was MVA (model-view-adapter), where the model and the view are unaware of each other, and the adapter was the link between them.
This architecture was chosen because the complete lack of coupling between the model and the view meant that one could be radically changed without affecting the other, which did occur multiple times throughout development

5.2 Cryptography

This section covers the implementation of cryptographic functions in the application.

In the application nearly all cryptographic functions are used via the “CryptoUtility” class.
In this section the functions CryptoUtility class will be discussed, as well as relevant code snippets of how it helps different areas of the program, such as transmission and disk storage.

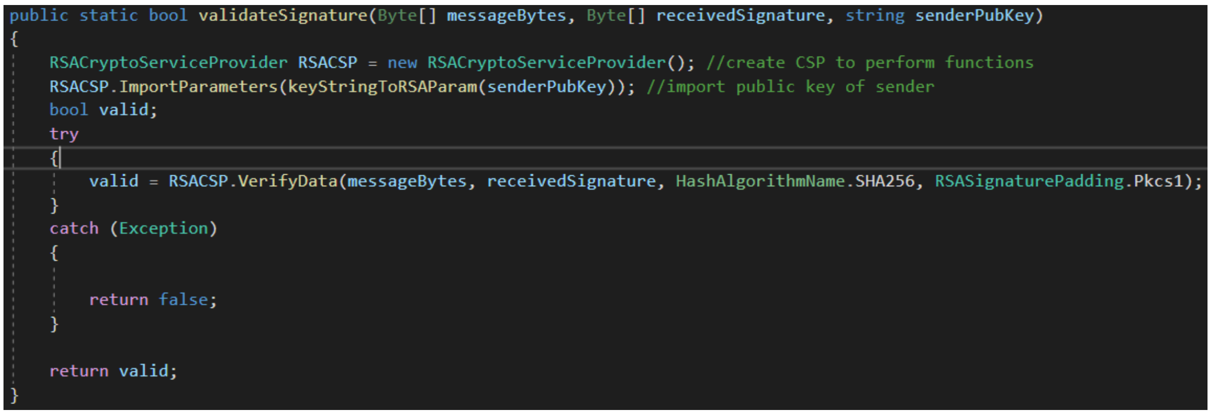
5.2.1 Transmission

Transmission uses RSA (4096 bits) to protect the sent information, what exactly the data is used for is covered under the Transmission segment, but here we will cover how the CryptoUtility class assists in transmission.

On start up the CryptoUtility class generates a random RSA key pair, these are the keys that will be used for any transmission until the application is restarted. The RSA implementation being used is the System.Security.Cryptography .RSACryptoServiceProvider API.

CryptoUtility provides the following utilities related to transmission:

- Get the RSA keys in string formats, for exchange during transmission
- Signs messages with the private key
- Encrypt with the received public key and decrypt with its own private key
- Validate signature of a received message (shown in Figure 1)



```
public static bool validateSignature(Byte[] messageBytes, Byte[] receivedSignature, string senderPubKey)
{
    RSACryptoServiceProvider RSACSP = new RSACryptoServiceProvider(); //create CSP to perform functions
    RSACSP.ImportParameters(keyStringToRSAParam(senderPubKey)); //import public key of sender
    bool valid;
    try
    {
        valid = RSACSP.VerifyData(messageBytes, receivedSignature, HashAlgorithmName.SHA256, RSASignaturePadding.Pkcs1);
    }
    catch (Exception)
    {

        return false;
    }

    return valid;
}
```

Figure 1: Code snippet: the method that validates signatures

5.2.2 Disk Storage

In order to have conversations between restarts of the application data must be stored on the disk. This data is protected using AES (128 bit), once again we will not cover how these are used yet, but what AES related functions are provided by cryptoUtility.

AES uses an Initialisation Vector (IV), the IV is a nonce (number used once) that prevents two identical inputs from producing the same output, when data is encrypted both the cipher text and the IV are needed, the IV on its own is useless and so can be stored in plaintext.

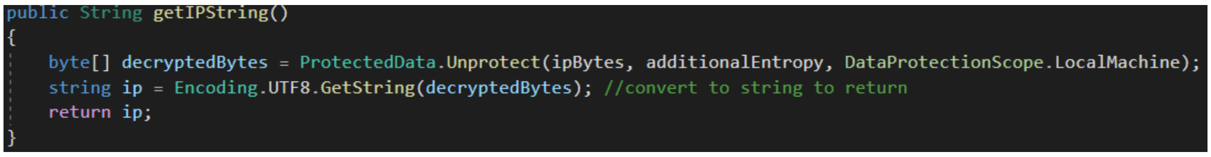
CryptoUtility provides the following utilities related to disk storage:

- Generate random AES keys
- Encrypt data with a specified key and random IV
- Encrypt a set of data all with the same IV (used for database tuples)
- Decrypt data with a specified key and IV

5.2.3 Memory Protection

Inevitably when data is needed it is decrypted and loaded into memory, this however causes a vulnerability, which is that memory imaging can be used to find the information in plaintext. To combat this data that will persist in memory is encrypted in memory and only temporarily exposed when being used. This was achieved using the System.Security.Cryptography .ProtectedData API.

Examples of data protected in memory include the IP addresses of contacts that have been loaded (see Figure 2) and cached AES keys that have been read from the database.



```
public String getIPString()
{
    byte[] decryptedBytes = ProtectedData.Unprotect(ipBytes, additionalEntropy, DataProtectionScope.LocalMachine);
    string ip = Encoding.UTF8.GetString(decryptedBytes); //convert to string to return
    return ip;
}
```

Figure 2: Code snippet: Example from Contact. Accessing information that has been protected in memory

5.2.4 Master Key

The AES keys themselves stored in the database need to be protected, as does the information regarding stored contacts. For these an unchanging key is needed, but a plaintext key stored anywhere or hardcoded is extremely vulnerable, so a key is instead derived from a password.

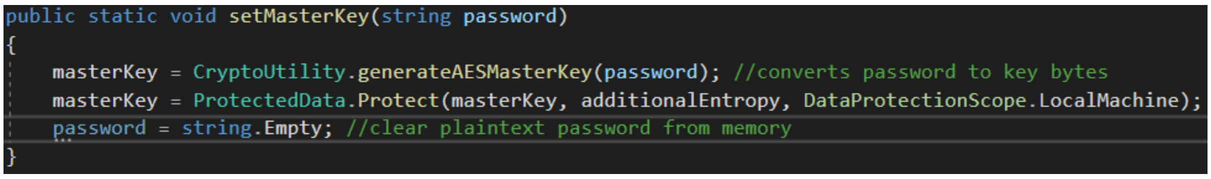
When the application is launched the user is asked for a password (if it's the first time then they choose one), this password is given to the Globals class which uses CryptoUtility to derive the master key from it.

CryptoUtility provides the following utility related to the master key:

- Derive AES key from a string

The key its self is protected in memory using the ProtectedData API, it is never permanently stored anywhere, only in the users memory.

Figure 3 shows the code that derives the master key from the users' password.



```
public static void setMasterKey(string password)
{
    masterKey = CryptoUtility.generateAESMasterKey(password); //converts password to key bytes
    masterKey = ProtectedData.Protect(masterKey, additionalEntropy, DataProtectionScope.LocalMachine);
    password = string.Empty; //clear plaintext password from memory
}
```

Figure 3: Code snippet: example from Globals. creating the master key. Note that the plaintext is eliminated when it is no longer needed

5.3 Networking

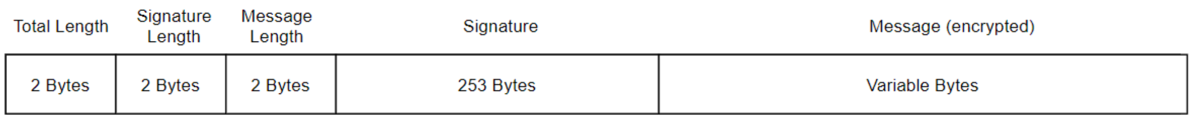
This section discusses the implementation of transmission and the related cryptography in the application, but does not cover how the messages reach the networking components through the architecture or what is done with them after they have been received.

The application abstracts networking into two classes, the ClientComp and the ServerComp, the server is always listening for incoming connections, the client is instantiated when needed and initiates a connection to the server when a message is being sent.

Terminology: the term "paired application" will be used, if two applications are in the middle of exchanging a message, then the senders paired application would be the receiver, and the receivers paired application will be the sender.

5.3.1 Transmission Format

The exchanging of messages requires the sending of two discreet pieces of information, the message and the signature. The chosen solution was to create a format that allows both to be sent in the same transmission, which can be seen in Figure 4.



Total Length	Signature Length	Message Length	Signature	Message (encrypted)
2 Bytes	2 Bytes	2 Bytes	253 Bytes	Variable Bytes

Figure 4: Illustration of how the transmission bytes are organised

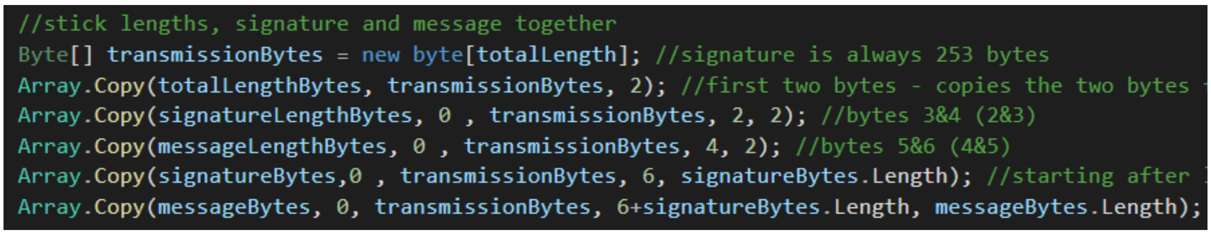
The format takes inspiration from a networking datagram, just like a datagram the message is a meaningless series of binary, but using headers set in fixed positions allows meaning to be extracted from it. For example, the first three bytes of a TCP packet state if it is a unicast or group address, this solution works on the same principles.

The transmission is formatted as an array of Bytes, when the array is being built (shown in Figure 5) an encrypted and signed version of the message is created, from that the signature, message and total lengths are calculated. The total length is how many bytes the entire transmission is, and is used during receiving.

An array is created that is *total length* elements in length, the first 6 elements are filled with the length headers. Each header is a two byte representation of a short (int16).

The next 253 bytes are filled with the message signature. 253 bytes are always generated with the chosen hash algorithm (SHA256), which means the headers (aside from total length) are not needed. While this is true, including the headers futureproofs the format in the event that the hashing algorithm or the signing process change in the future.

Finally the remaining bytes (of variable length, depending on key size) that represent the encrypted message will be put into the array.



```
//stick lengths, signature and message together
Byte[] transmissionBytes = new byte[totalLength]; //signature is always 253 bytes
Array.Copy(totalLengthBytes, transmissionBytes, 2); //first two bytes - copies the two bytes
Array.Copy(signatureLengthBytes, 0 , transmissionBytes, 2, 2); //bytes 3&4 (2&3)
Array.Copy(messageLengthBytes, 0 , transmissionBytes, 4, 2); //bytes 5&6 (4&5)
Array.Copy(signatureBytes,0 , transmissionBytes, 6, signatureBytes.Length); //starting after
Array.Copy(messageBytes, 0, transmissionBytes, 6+signatureBytes.Length, messageBytes.Length);
```

Figure 5: Code snippet: from ClientComp. assembling information into transmission format

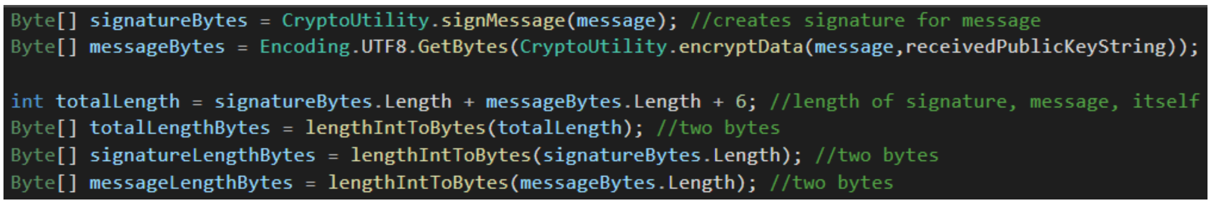
5.3.2 Client Component

The ClientComp is instantiated when the user selects a contact, it is instantiated with the IP Address of the selected user.

When the client sends a message, the content is given to the ClientComp and it performs the following process.

- Synchronously connect to the ServerComp at the contacts IPAddress.
- Sends the applications current public RSA key (acquired via the CryptoUtility class).
- Receives and stores (in memory) the public key of the paired application.
- Builds transmission bytes (see below) and synchronously send them to paired application.
- Wait up to two seconds to get confirmation of a successful receive from paired application.
- Use controller callback to inform it that message has been successfully sent.

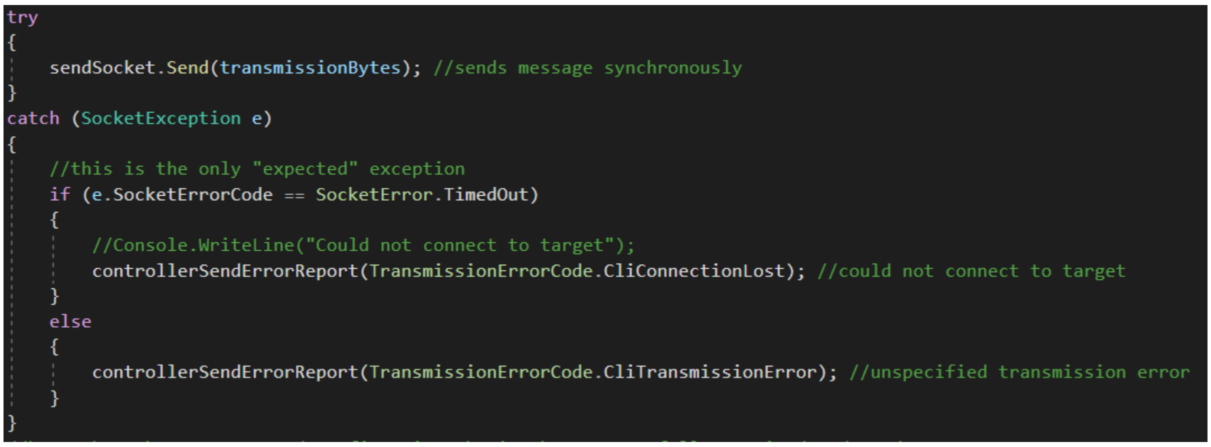
To build the transmission bytes the CryptoUtility class is used to get a message signature with the private RSA key, it then once again uses CryptoUtility to encrypt the message with the public RSA key received from the paired application. It then does the rest of the calculations needed to build the message format as described under “Transmission Format”. This process is shown in Figure 6.



```
Byte[] signatureBytes = CryptoUtility.signMessage(message); //creates signature for message
Byte[] messageBytes = Encoding.UTF8.GetBytes(CryptoUtility.encryptData(message,receivedPublicKeyString));

int totalLength = signatureBytes.Length + messageBytes.Length + 6; //length of signature, message, itself
Byte[] totalLengthBytes = lengthIntToBytes(totalLength); //two bytes
Byte[] signatureLengthBytes = lengthIntToBytes(signatureBytes.Length); //two bytes
Byte[] messageLengthBytes = lengthIntToBytes(messageBytes.Length); //two bytes
```

Figure 6: Code snippet: Leveraging CryptoUtility to encrypt data and then working out the headers

There are try-catch blocks for various exceptions that could occur during this process, when one is caught a callback is used to send the relevant error code to the controller (see Figure 7), which then informs the user.



```
try
{
    sendSocket.Send(transmissionBytes); //sends message synchronously
}
catch (SocketException e)
{
    //this is the only "expected" exception
    if (e.SocketErrorCode == SocketError.TimedOut)
    {
        //Console.WriteLine("Could not connect to target");
        controllerSendErrorReport(TransmissionErrorCode.CliConnectionLost); //could not connect to target
    }
    else
    {
        controllerSendErrorReport(TransmissionErrorCode.CliTransmissionError); //unspecified transmission error
    }
}
```

Figure 7: Code snippet: sending the data synchronously and reporting errors depending on the caught exception

5.3.3 Server Component

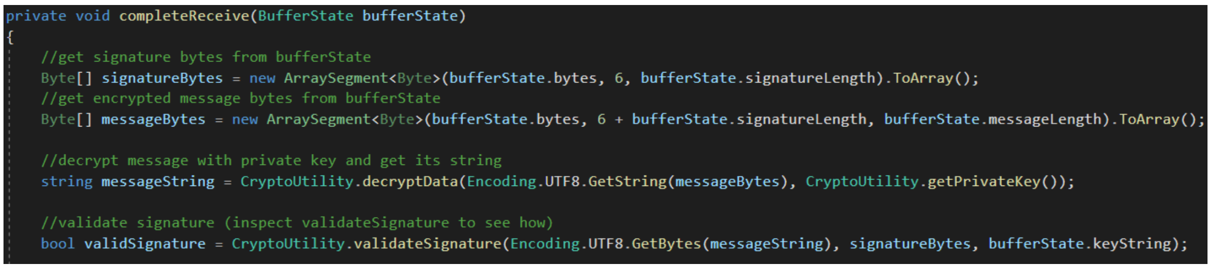
The server component is always running and listens for incoming connections on all network interfaces.

Unlike the client component it cannot do things synchronously, while it is acceptable to have the client block when the user *sends* a message, if it blocks while waiting to receive one the user would never be able to do anything, therefor the server makes heavy use of threading.

When the server is instantiated an asynchronous socket function is started, which calls back to acceptTPCRequest() when a TCP connection request is received. acceptTCPRequest() creates a new socket to handle the connection in a new thread and the original socket is told to do the same asynchronous function again.

When the server receives a connection request, it performs the following process.

- Receives and stores (in BufferState- see below) the public RSA key from paired application.
- Sends its own public RSA key (acquired via the CryptoUtility class) back to paired application.
- Begins the asynchronous Receive method (see below).
- Decrypts message, validates signature and builds Message object with senders IP address, part of which is shown in Figure 8.
- Use callback to return the message to the controller.



```
private void completeReceive(BufferState bufferState)
{
    //get signature bytes from bufferState
    Byte[] signatureBytes = new ArraySegment<Byte>(bufferState.bytes, 6, bufferState.signatureLength).ToArray();
    //get encrypted message bytes from bufferState
    Byte[] messageBytes = new ArraySegment<Byte>(bufferState.bytes, 6 + bufferState.signatureLength, bufferState.messageLength).ToArray();

    //decrypt message with private key and get its string
    string messageString = CryptoUtility.decryptData(Encoding.UTF8.GetString(messageBytes), CryptoUtility.getPrivateKey());

    //validate signature (inspect validateSignature to see how)
    bool validSignature = CryptoUtility.validateSignature(Encoding.UTF8.GetBytes(messageString), signatureBytes, bufferState.keyString);
```

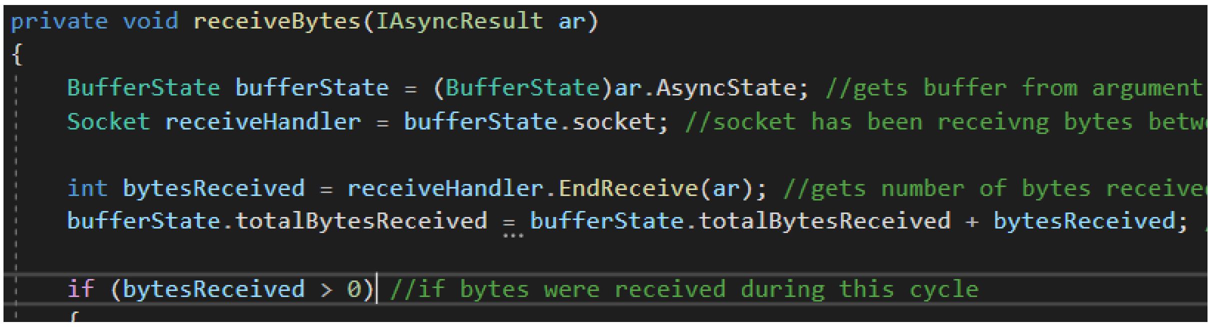
Figure 8: Code Snippet: Extraction of signature and encrypted message from the transmitted bytes, followed by message decryption and signature validation

5.3.3.1 Asynchronous receive method

The method is called receiveBytes and it takes an asynchronous state object as a parameter. The state object is used to retain/restore states between asynchronous method calls.

receiveBytes() does the following process:

- Firstly it gets the bytes that have so-far been received by the socket (see Figure 9).
- If enough bytes have been received and the values have not been set yet, then the headers are read, converted to integers and stored.
- If the number of bytes received is less than the header claims there are, then the socket is told to receive again and call this method when it has data.
- If the number received is the same as the total length header, then the bytes are sent to be decrypted and validated.
- Finally if no bytes or more than the header says there should be are received, then an error is reported to the controller.

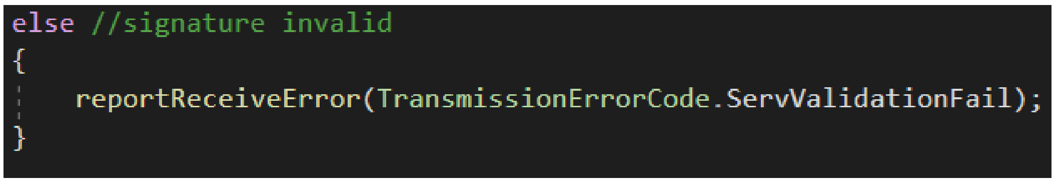


```
private void receiveBytes(IAsyncResult ar)
{
    BufferState bufferState = (BufferState)ar.AsyncState; //gets buffer from argument
    Socket receiveHandler = bufferState.socket; //socket has been receivng bytes betw

    int bytesReceived = receiveHandler.EndReceive(ar); //gets number of bytes receive
    bufferState.totalBytesReceived = bufferState.totalBytesReceived + bytesReceived;

    if (bytesReceived > 0) //if bytes were received during this cycle
    {
```

Figure 9: Code snippet: start of receiveBytes, showing how the state is resumed between iterations.

The server is designed to handle multiple incoming messages at the same time, for that reason all intermediate states are stored in an instance of the BufferState class, it is also used to maintain state between asynchronous functions.

Like the client component, there are try-catch blocks for various exceptions that could occur during this process, when one is caught a callback is used to send the relevant error code to the controller, which then informs the user. One such example would be the signature failing to validate, which can be seen in Figure 10.



```
else //signature invalid
{
    reportReceiveError(TransmissionErrorCode.ServValidationFail);
}
```

Figure 10: Code snippet: from completeReceive. Reporting invalid signature

5.4 Local Storage

This section covers how local storage was implemented, including the database schema and the classes that interface with it.

5.4.1 Database Schema

Figure 11 shows the final database schema.

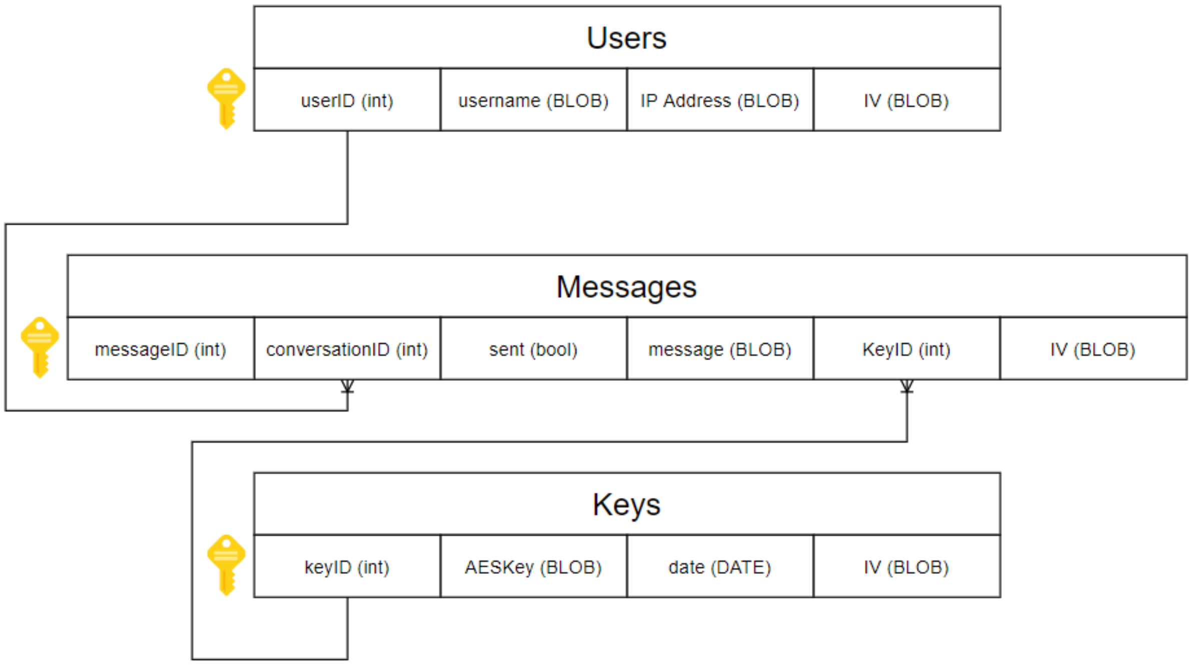


Figure 11: Illustration of the finalised database schema

Username, IP Address, message and AESKey are all encrypted, the output of encryption is an array of bytes, rather than converting the byte arrays to characters they are simply stored as BLOBs.

IV stands for initialisation vector, this is the nonce (number used once) that prevents two identical inputs from creating the same cipher text. The IV alone is useless and can be stored in plaintext, which in this case is also a BLOB.

In the Users table the username and IPAddress are encrypted with the same IV, this means that an extra IV does not have to be stored, which, if the master key was compromised, would add no additional security.

The reason Keys has a date is described in chapter 3. A key could either be associated with a user or a particular day (whereas a key per message would be impractical). A key per user would be simpler, but if it was ever compromised potentially thousands (over years) of

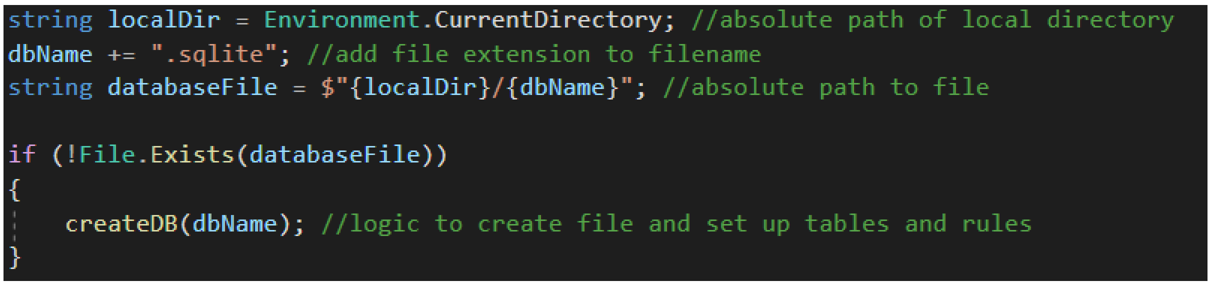
messages could be exposed.
Instead each day a new key (if any messages are sent/received) is created, and for that day all messages will use that key. That way if a key is compromised, only a maximum of several hundred messages (if the user is very active) could be exposed.

5.4.2 Database Interfacing

There are several classes that wrap database interaction, the generalised DatabaseInterface, and three more specialised classes that use DatabaseInterface.

The Database interface class is a singleton which wraps a System.Data.SQLite.SQLiteConnection class.

When the class is instantiated it is able to check that the database already exists, and creates one if it does not, this is shown in Figure 12.



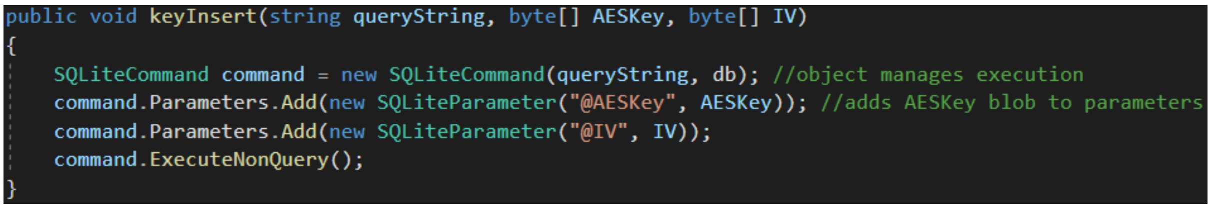
```
string localDir = Environment.CurrentDirectory; //absolute path of local directory
dbName += ".sqlite"; //add file extension to filename
string databaseFile = $"{localDir}/{dbName}"; //absolute path to file

if (!File.Exists(databaseFile))
{
    createDB(dbName); //logic to create file and set up tables and rules
}
```

Figure 12: Code snippet: DatabaseInterface checking if file exists, creating it if it does not

DatabaseInterface provides the following methods for interacting with the SQLite database:

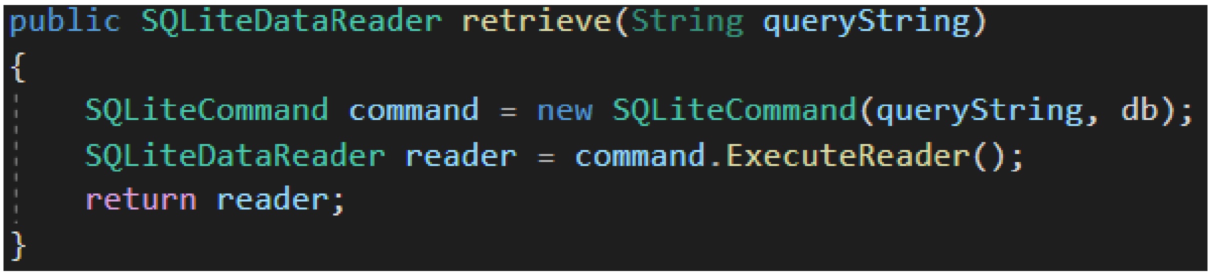
- Run any query (string) that has no return
- Run any query (string) that does have a return
- Run queries that create entries for Users, Messages and Keys

Users, Messages and Keys have their own methods because, using this SQLite library, at least, blobs cannot be passed in through the query string. Figure 13 shows one of the methods that inserts blobs into queries, and Figure 14 shows the method that is used to retrieve any information from the database.



```
public void keyInsert(string queryString, byte[] AESKey, byte[] IV)
{
    SQLiteCommand command = new SQLiteCommand(queryString, db); //object manages execution
    command.Parameters.Add(new SQLiteParameter("@AESKey", AESKey)); //adds AESKey blob to parameters
    command.Parameters.Add(new SQLiteParameter("@IV", IV));
    command.ExecuteNonQuery();
}
```

Figure 13: Code snippet: Example of how blobs are inserted into queries



```
public SQLiteDataReader retrieve(String queryString)
{
    SQLiteCommand command = new SQLiteCommand(queryString, db);
    SQLiteDataReader reader = command.ExecuteReader();
    return reader;
}
```

Figure 14: Code snippet: The method used to retrieve any information from the database

5.4.3 Contact Management

The ContactManager is the class that manages contacts and wraps contact-related database interactions.

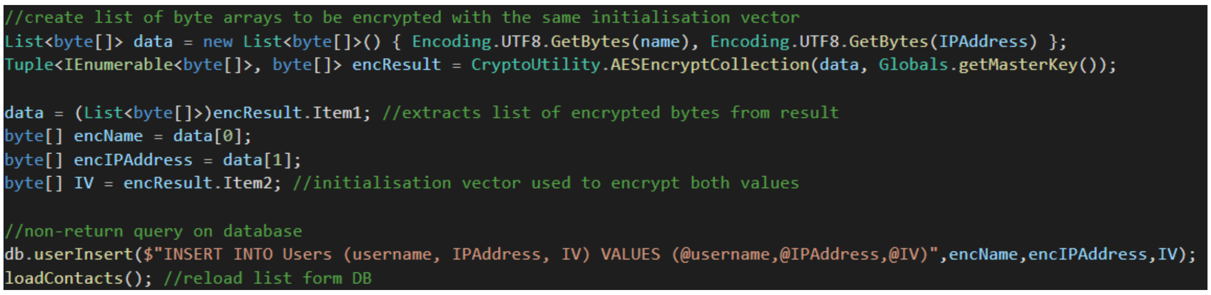
ContactManager maintains and manages access to a list of Contact objects which represent each contact the user has in the database.

Contact Manager provides the following methods for managing contacts:

- Get individual contact by ID
- Get the full list of contacts
- Identify the contact who sent a message via their IP address
- Add a new contact to the database

5.4.3.1 Creating/reading contact entries

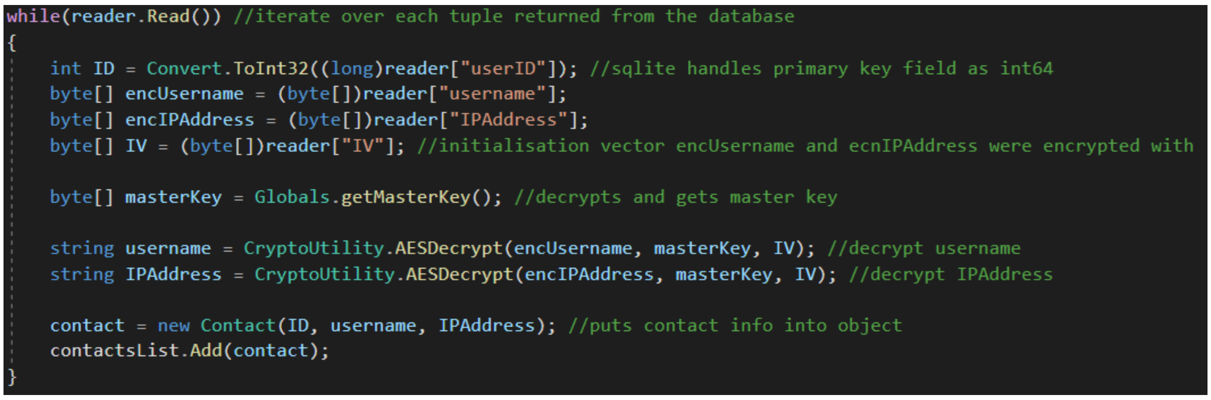
When an entry is created the CryptoUtility class is used to encrypt both the contacts name and IP address with the master (AES) key and the same IV. The encrypted data and the IV are then inserted into the table, as seen in Figure 15.



```
//create list of byte arrays to be encrypted with the same initialisation vector
List<byte[]> data = new List<byte[]>() { Encoding.UTF8.GetBytes(name), Encoding.UTF8.GetBytes(IPAddress) };
Tuple<IEnumerable<byte[]>, byte[]> encResult = CryptoUtility.AESEncryptCollection(data, Globals.getMasterKey());

data = (List<byte[]>)encResult.Item1; //extracts list of encrypted bytes from result
byte[] encName = data[0];
byte[] encIPAddress = data[1];
byte[] IV = encResult.Item2; //initialisation vector used to encrypt both values

//non-return query on database
db.userInsert($"INSERT INTO Users (username, IPAddress, IV) VALUES (@username,@IPAddress,@IV)",encName,encIPAddress,IV);
loadContacts(); //reload list form DB
```

Figure 15: Code snippet: Encrypting contact information, extracting the ciphertext, then inserting it into the database

Reading is the same process done in reverse, the data is read and the IV is used to decrypt the name and IP address with the master key. The information is then used to create Contact objects, which then encrypt the IP address in memory. Figure 16 shows the extraction of contact information from a single tuple.



```
while(reader.Read()) //iterate over each tuple returned from the database
{
    int ID = Convert.ToInt32((long)reader["userID"]); //sqlite handles primary key field as int64
    byte[] encUsername = (byte[])reader["username"];
    byte[] encIPAddress = (byte[])reader["IPAddress"];
    byte[] IV = (byte[])reader["IV"]; //initialisation vector encUsername and ecnIPAddress were encrypted with

    byte[] masterKey = Globals.getMasterKey(); //decrypts and gets master key

    string username = CryptoUtility.AESDecrypt(encUsername, masterKey, IV); //decrypt username
    string IPAddress = CryptoUtility.AESDecrypt(encIPAddress, masterKey, IV); //decrypt IPAddress

    contact = new Contact(ID, username, IPAddress); //puts contact info into object
    contactsList.Add(contact);
}
```

Figure 16: Code snippet: Iterating over query results, extracting contact information, decrypting it, then instantiating Contact objects

5.4.4 Key Management

The KeyManager is the class that manages access to AES keys stored in the database.

The KeyManager does not maintain a list of keys, but it does Cache any keys that have been requested before. This prevents unnecessary database reads but does not compromise security, as they are encrypted in memory.

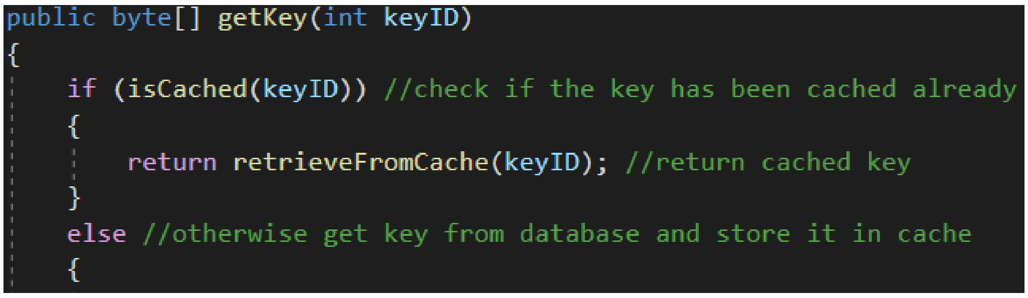
KeyManager provides the following methods for managing keys:

- Retrieve a key by its ID (which is cached to avoid unnecessary database reads)
- Get the key being used for the current day

5.4.4.1 Creating/reading key entries

The KeyManager does not provide an external method for creating and storing keys, it is only done internally when the day's key is requested and does not yet exist.

Reading keys is done when the message manager needs them to decrypt messages being read from the database. The same key may be needed many times in a session (many messages could be sent on the same day, therefor having the same key), so when a key is read it is cached to prevent unnecessary database reads. Figure 17 shows how keys are retrieved from the cache, prevent unnecessary database interactions.



```
public byte[] getKey(int keyID)
{
    if (isCached(keyID)) //check if the key has been cached already
    {
        return retrieveFromCache(keyID); //return cached key
    }
    else //otherwise get key from database and store it in cache
    {
```

Figure 17: Code snippet: How caching the keys is used to avoid unnecessary work

5.4.5 Message Management

The MessageManager is the class that manages all message-related database interactions.

MessageManager provides the following methods for managing messages:

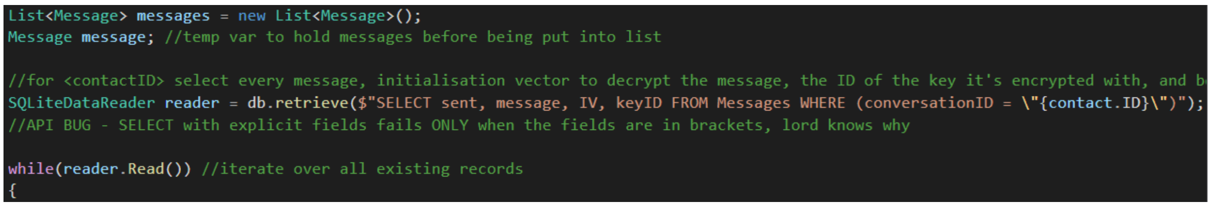
- Get all Messages associated with a contact
- Commit a message to the database

5.4.5.1 Creating entries

When a message is stored the day's key is acquired in plaintext, the encryption is done, getting the ciphertext and IV, and the entry is made in the database. After the encryption is completed the plaintext key is discarded.

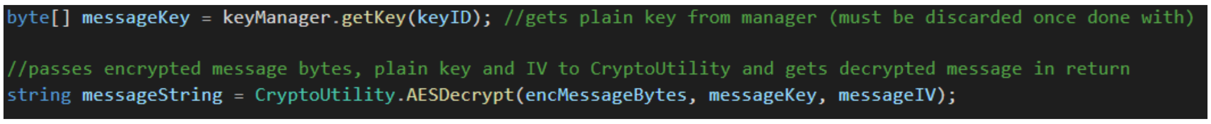
5.4.5.2 Getting all Messages associated with a contact

All entries associated with the contact's ID are fetched (Figure 18), for each one the information is extracted. The ID of the key used for each message is used to acquire the key from the KeyManager (Figure 19), after the first instance of a key being fetched it is cached. Once all the information is in plaintext it is used to instantiate a Message object which is put into a list to be returned.



```
List<Message> messages = new List<Message>();
Message message; //temp var to hold messages before being put into list

//for <contactID> select every message, initialisation vector to decrypt the message, the ID of the key it's encrypted with, and b
SQLiteDataReader reader = db.retrieve($"SELECT sent, message, IV, keyID FROM Messages WHERE (conversationID = \"{contact.ID}\")");
//API BUG - SELECT with explicit fields fails ONLY when the fields are in brackets, lord knows why

while(reader.Read()) //iterate over all existing records
{
```

Figure 18: Code snippet: The query that gets all Messages associated with a contact, which then goes onto extract the information and make Message objects



```
byte[] messageKey = keyManager.getKey(keyID); //gets plain key from manager (must be discarded once done with)

//passes encrypted message bytes, plain key and IV to CryptoUtility and gets decrypted message in return
string messageString = CryptoUtility.AESDecrypt(encMessageBytes, messageKey, messageIV);
```

Figure 19: Code Snippet: The process of extracting and decrypting the information nearly identical to that of contacts and keys, except the key is requested from the KeyManager, instead of using the master key from Globals

5.5 Structural Diagrams

This section shows and explains diagrams which communicate the structure of the system and how parts of the system interact with each other at runtime.

Some information about how the system works is explained where relevant, if it was not already touched on.

The Diagram in Figure 20 shows a high level overview of how two users can communicate with each other over the internet and their local networks using this application.

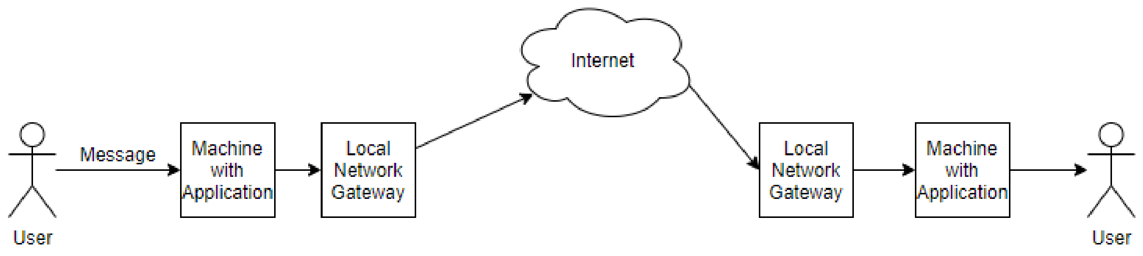


Figure 20: High level overview of how users can communicate using this software

5.5.1 MVA

The application is an implementation of the Model-View-Adapter (MVA) architecture, an abstract representation of MVA can be seen in Figure 21.

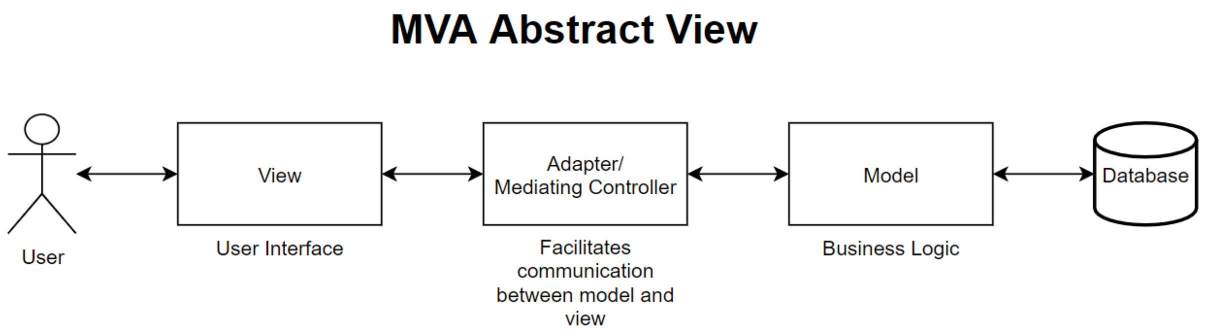

MVA Abstract View
User
View
User Interface
Adapter/ Mediating Controller
Facilitates communication between model and view
Model
Business Logic
Database

Figure 21: Abstract diagram showing the fundamentals of MVA architecture

As the diagram shows, the user interacts with the application via the View, this means the view must have means of taking input or displaying output.
The Model contains the business logic, that is, what the application is actually being used for, this includes the code for transmitting messages, encrypting information, storing information, and other background jobs such as managing contacts.

The Adapter/Mediating Controller coordinates and facilitates communication between both the model and the view, it mostly allows the model to react to the view, or vice versa.
For example when the user presses the send message button the view collects the input and informs the controller, the controller then uses the business logic in the model to have the message sent. When the model receives feedback that the message has been received it informs the controller, which then tells the view to display the message.

MVA is an excellent choice for implementing a GUI because the abstraction makes it much easier to plan and explain what the software is doing. It is also convenient because classes in the view are coupled only with the controller, and not at all with the model, meaning that even if major refactors are made to the model (which did occur during development of this application) the view may not have to be changed at all (which it did not).

5.5.2 Class diagrams

Now that the overall abstract architectural choices have been illustrated and explained, we will discuss the concrete class diagrams.
As the system is quite complex, the class diagrams are split up into classes that form the Model, View and Adapter respectively. There is also a separate diagram for special case classes.

5.5.2.1 Special Case

This category contains classes that do not strictly belong to the model, view or Adapter, and are instead used by all three; they will be illustrated here once and referenced by name in the other diagrams.

Figure 22 Shows the Globals class, as the name would suggest it wraps global variables, which are the length of the AES key, the name of the database that will be read from/written to and protects and allows access to the master key.

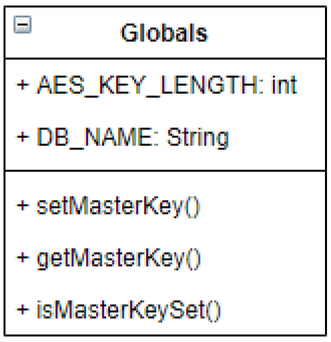


Figure 22: Class diagram: the Globals class

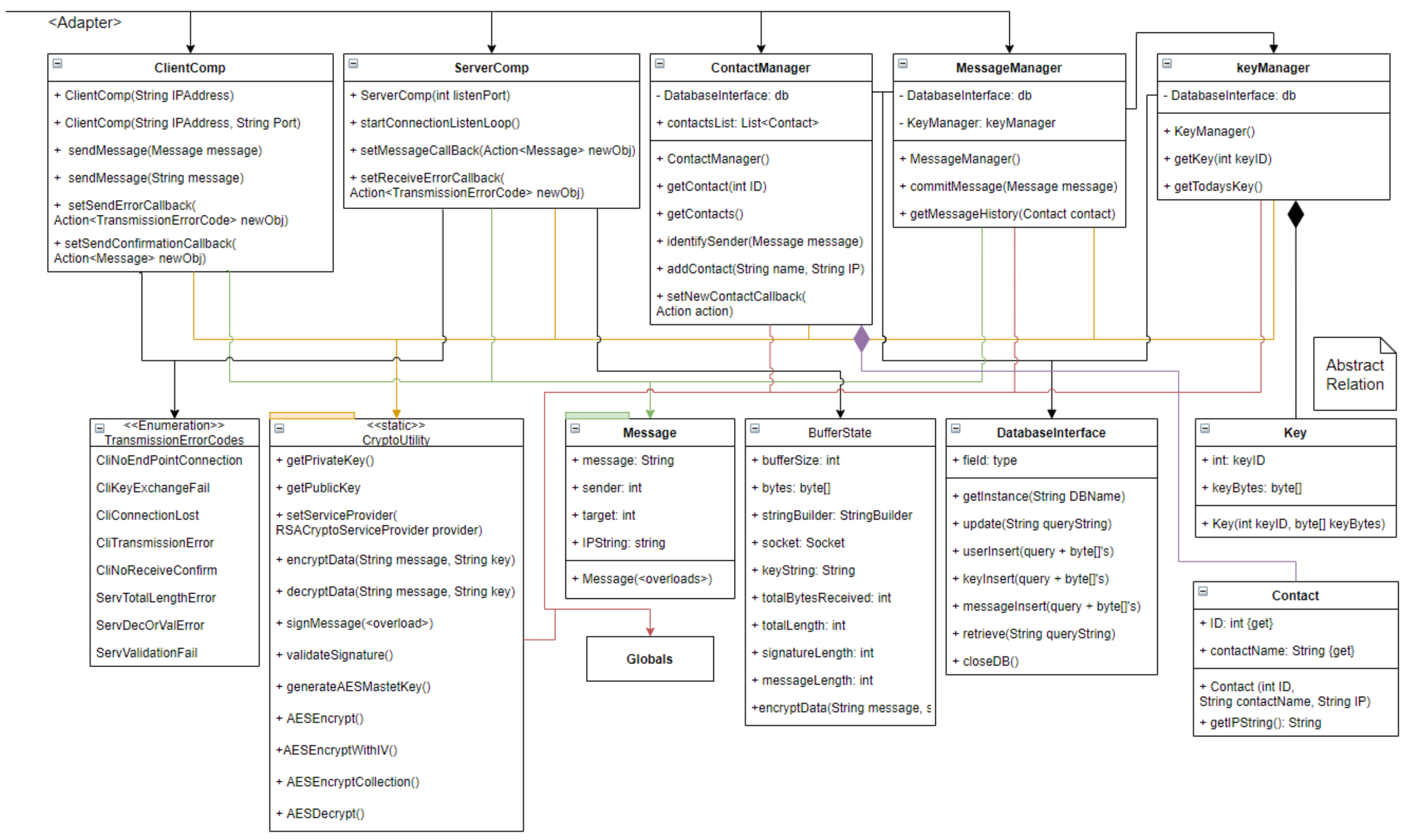
5.5.2.2 Model

The diagram in Figure 23 illustrates the classes that make up the model and their relationships.

The model could be further abstracted into classes that are leveraged in transmission and classes that are leverages in local storage.

Transmission: Client/Server components, BufferState, TransmissionErrorCodes, CryptoUtility, Message, ContactManager, Contact.

Local Storage: ContactManager, MessageManager, keyManager, DatabaseInterface, CryptoUtility, Key, Message, Contact.



<Adapter>
ClientComp
+ ClientComp(String IPAddress)
+ ClientComp(String IPAddress, String Port)
+ sendMessage(Message message)
+ sendMessage(String message)
+ setSendErrorCallback(Action<TransmissionErrorCode> newObj)
+ setSendConfirmationCallback(Action<Message> newObj)
ServerComp
+ ServerComp(int listenPort)
+ startConnectionListenLoop()
+ setMessageCallBack(Action<Message> newObj)
+ setReceiveErrorCallback(Action<TransmissionErrorCode> newObj)
ContactManager
- DatabaseInterface: db
+ contactsList: List<Contact>
+ ContactManager()
+ getContact(int ID)
+ getContacts()
+ identifySender(Message message)
+ addContact(String name, String IP)
+ setNewContactCallback(Action action)
MessageManager
- DatabaseInterface: db
- KeyManager: keyManager
+ MessageManager()
+ commitMessage(Message message)
+ getMessageHistory(Contact contact)
keyManager
- DatabaseInterface: db
+ KeyManager()
+ getKey(int keyID)
+ getTodaysKey()
Abstract Relation
<<Enumeration>> TransmissionErrorCodes
CliNoEndPointConnection
CliKeyExchangeFail
CliConnectionLost
CliTransmissionError
CliNoReceiveConfirm
ServTotalLengthError
ServDecOrValError
ServValidationFail
<<static>> CryptoUtility
+ getPrivateKey()
+ getPublicKey
+ setServiceProvider(RSACryptoServiceProvider provider)
+ encryptData(String message, String key)
+ decryptData(String message, String key)
+ signMessage(<overload>)
+ validateSignature()
+ generateAESMastetKey()
+ AESEncrypt()
+AESEncryptWithIV()
+ AESEncryptCollection()
+ AESDecrypt()
Message
+ message: String
+ sender: int
+ target: int
+ IPString: string
+ Message(<overloads>)
Globals
BufferState
+ bufferSize: int
+ bytes: byte[]
+ stringBuilder: StringBuilder
+ socket: Socket
+ keyString: String
+ totalBytesReceived: int
+ totalLength: int
+ signatureLength: int
+ messageLength: int
+encryptData(String message, s
DatabaseInterface
+ field: type
+ getInstance(String DBName)
+ update(String queryString)
+ userInsert(query + byte[]'s)
+ keyInsert(query + byte[]'s)
+ messageInsert(query + byte[]'s)
+ retrieve(String queryString)
+ closeDB()
Key
+ int: keyID
+ keyBytes: byte[]
+ Key(int keyID, byte[] keyBytes)
Contact
+ ID: int {get}
+ contactName: String {get}
+ Contact (int ID, String contactName, String IP)
+ getIPString(): String

Figure 23: Class diagram: Illustration of classes that make up the model

5.5.2.3 View

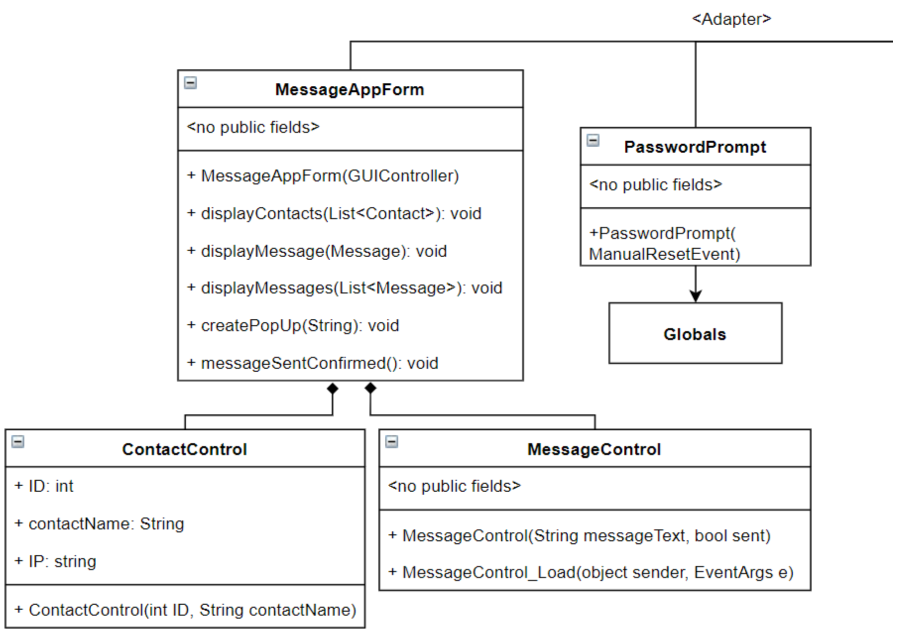
Figure 24 illustrates the classes that make up the view, along with their relationships.

On startup the main thread is paused and PasswordPrompt is instantiated in its own thread, once it has the password it sets the master key via Globals, unpauses the main thread and terminates itself.

MessageAppForm is the main view class and is owned by the controller, it leverages ContactControl and UserControl to fulfil its purpose.

MessageAppForm has the logic needed to accept user inputs and display information, it is able to pass inputs down to the controller or react to commands from the controller, such as displaying a message.

Contact/MessageControl are user controls. In the C# .NET framework a programmer can define their own GUI element called a control, the ContactControls are the panels on the left side of the window listing the users' contacts, and the MessageControls are the messages in the chat window.



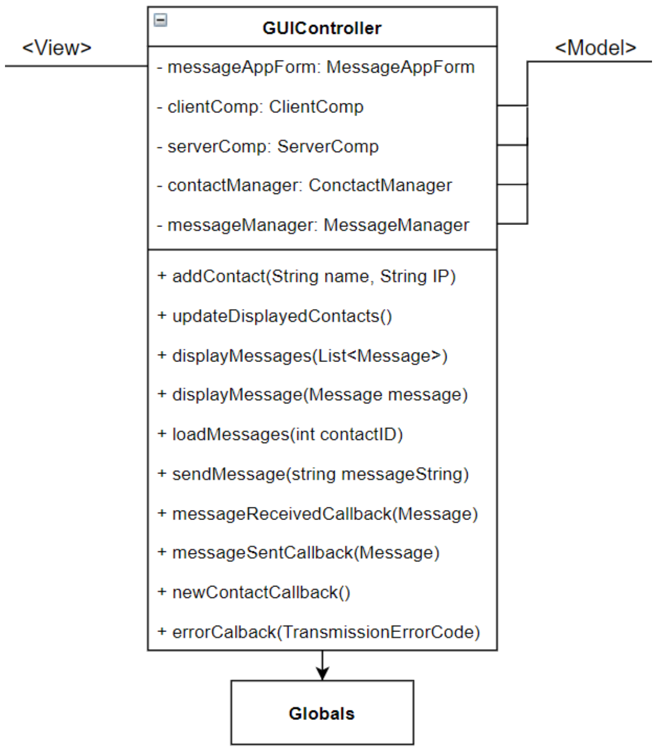
<Adapter>
MessageAppForm
<no public fields>
+ MessageAppForm(GUIController)
+ displayContacts(List<Contact>): void
+ displayMessage(Message): void
+ displayMessages(List<Message>): void
+ createPopUp(String): void
+ messageSentConfirmed(): void
PasswordPrompt
<no public fields>
+PasswordPrompt(ManualResetEvent)
Globals
ContactControl
+ ID: int
+ contactName: String
+ IP: string
+ ContactControl(int ID, String contactName)
MessageControl
<no public fields>
+ MessageControl(String messageText, bool sent)
+ MessageControl_Load(object sender, EventArgs e)

Figure 24: Class diagram showing the classes involved in the View and their relationships

5.5.2.4 Adapter

There is only one class under this category, which is the GUIController illustrated in Figure 25.

GUIController is the mediator between the model and the view, as discussed before it facilitates all communication and leverages the other classes in order to satisfy the user's needs.



<View>
<Model>
GUIController
- messageAppForm: MessageAppForm
- clientComp: ClientComp
- serverComp: ServerComp
- contactManager: ConctactManager
- messageManager: MessageManager
+ addContact(String name, String IP)
+ updateDisplayedContacts()
+ displayMessages(List<Message>)
+ displayMessage(Message message)
+ loadMessages(int contactID)
+ sendMessage(string messageString)
+ messageReceivedCallback(Message)
+ messageSentCallback(Message)
+ newContactCallback()
+ errorCalback(TransmissionErrorCode)
Globals

Figure 25: Class diagram: Illustration of the GUIController, the mediating controller of the software

5.5.3 Object Sequence Diagrams

This section shows the object sequence diagrams (OSDs) that illustrate what the program is doing in response to the various use cases. The use cases covered are the user sending a message to a contact, the user reading past messages exchanged with a contact, and finally the user adding a new contact.

5.5.3.1 Sending Messages

The OSD in Figure 26 shows the process that takes place when the user sends a message, the details of what ClientComp does can be seen in Figure 27 and the details of MessageManager can be seen in Figure 28.

In Summary the message details are passed from the view to the adapter (GUIController) to the model. The model handles the transmission of the message and reports back to the adapter, the adapter tells the model to store the message in the database and then passes the message back to the view to be displayed.

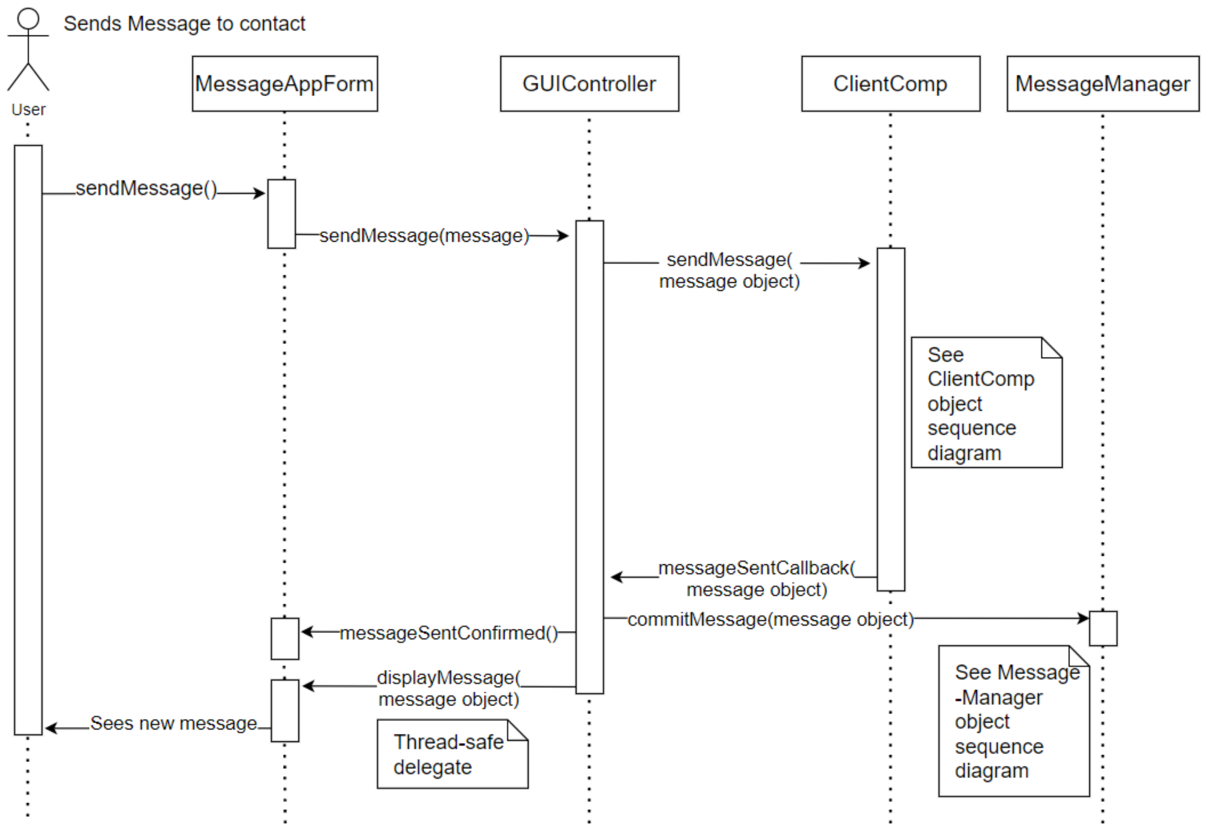

Sends Message to contact
User
MessageAppForm
GUIController
ClientComp
MessageManager
sendMessage()
sendMessage(message)
sendMessage(message object)
See ClientComp object sequence diagram
messageSentCallback(message object)
commitMessage(message object)
messageSentConfirmed()
displayMessage(message object)
See Message -Manager object sequence diagram
Sees new message
Thread-safe delegate

Figure 26: Object sequence diagram: what happens when the user sends a message

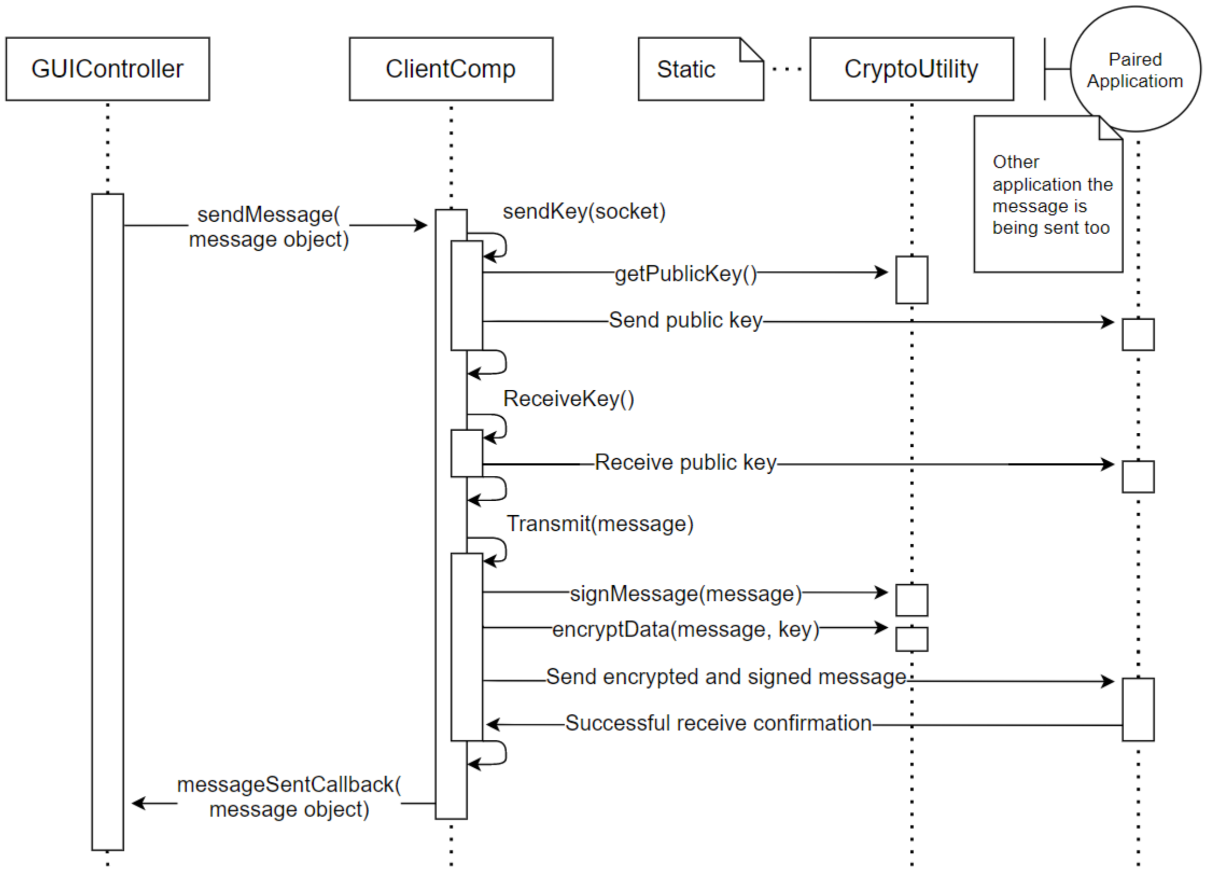

GUIController
ClientComp
Static
CryptoUtility
Paired Applicatiom
Other application the message is being sent too
sendMessage(message object)
sendKey(socket)
getPublicKey()
Send public key
ReceiveKey()
Receive public key
Transmit(message)
signMessage(message)
encryptData(message, key)
Send encrypted and signed message
Successful receive confirmation
messageSentCallback(message object)

Figure 27: Object sequence diagram: How the ClientComp sends a message

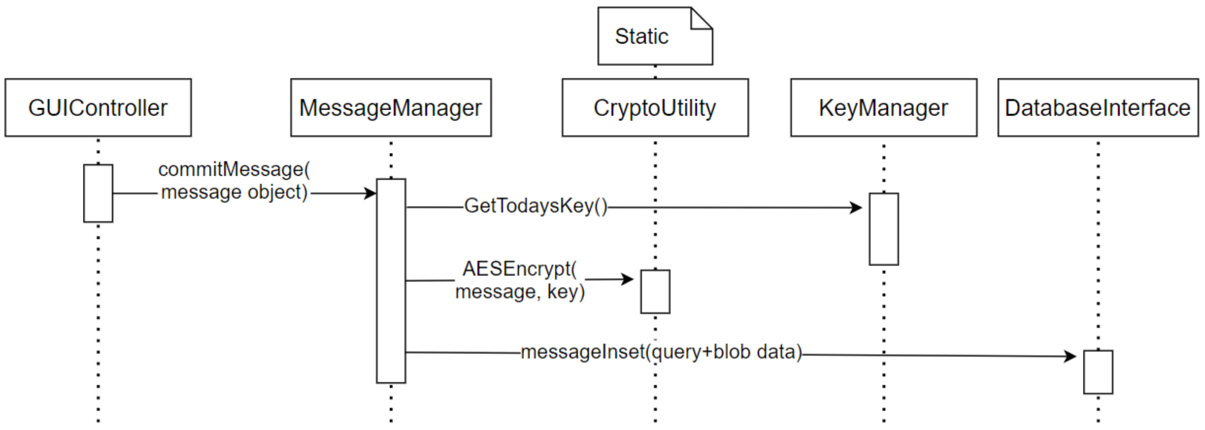


Figure 28: Object sequence diagram: How the MessageManager securely stores a message in the database

5.5.3.2 Reading Messages

The OSD in Figure 29 shows what happens when the user clicks on a contact in order to view their message history. Figure 30 shows what exactly the MessageManager is doing to get the messages from the database.

In Summary the ID of the clicked contact is passed from the view to the adapter (GUIController) to the model. The contact manager in the model returns the appropriate contact to the adapter, which uses the MessageManager to get their message history. A list containing the messages is passed up from the model to the view to be displayed to the user.

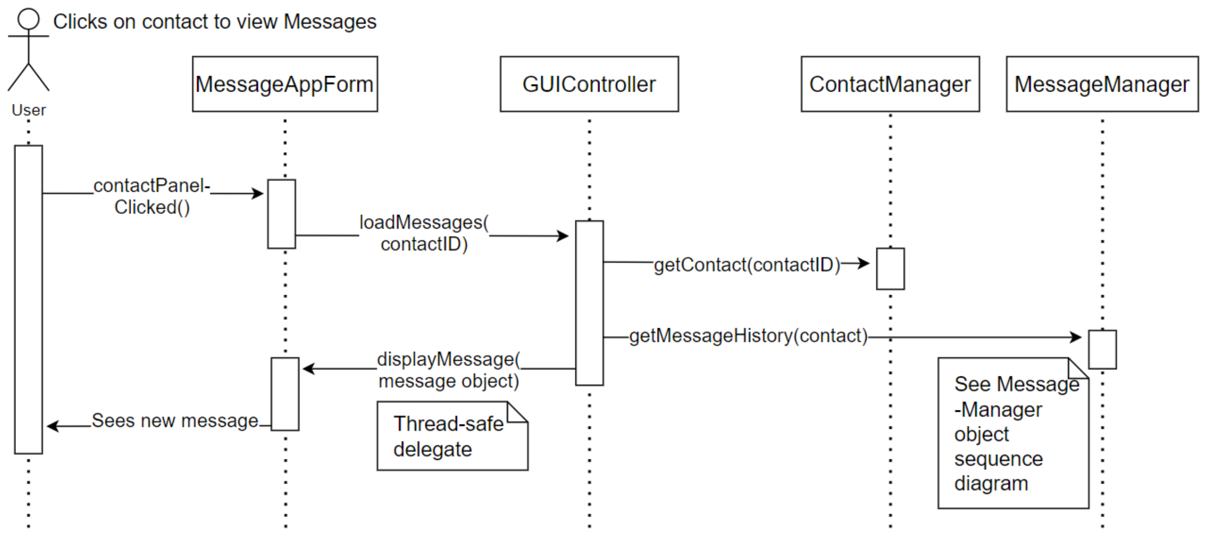


Figure 29: Object sequence diagram: What happens when the user tries to read messages exchanged with a contact

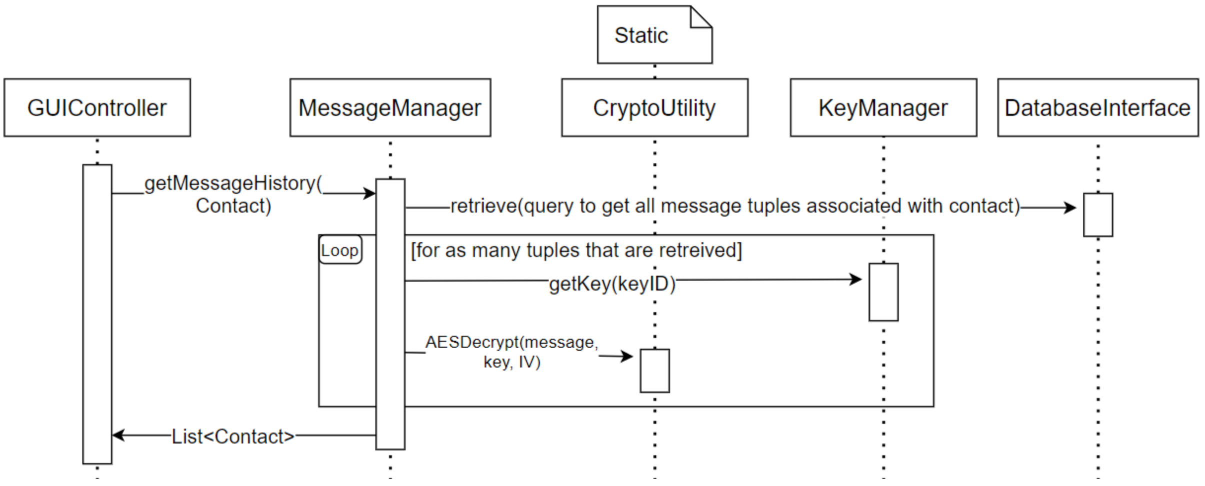

Static
GUIController
MessageManager
CryptoUtility
KeyManager
DatabaseInterface
getMessageHistory(Contact)
retrieve(query to get all message tuples associated with contact)
Loop
[for as many tuples that are retreived]
getKey(keyID)
AESDecrypt(message, key, IV)
List<Contact>

Figure 30: Object sequence diagram: How the message manager loads past conversations

5.5.3.3 Adding Contacts

The OSD in Figure 31 Illustrates what happens when the user adds a contact.

In summary the contact details are passed from the view to the adapter (GUIController) to the model. The model encrypts the data and stores it in the database. The contacts are then passed back up to the view to be displayed.

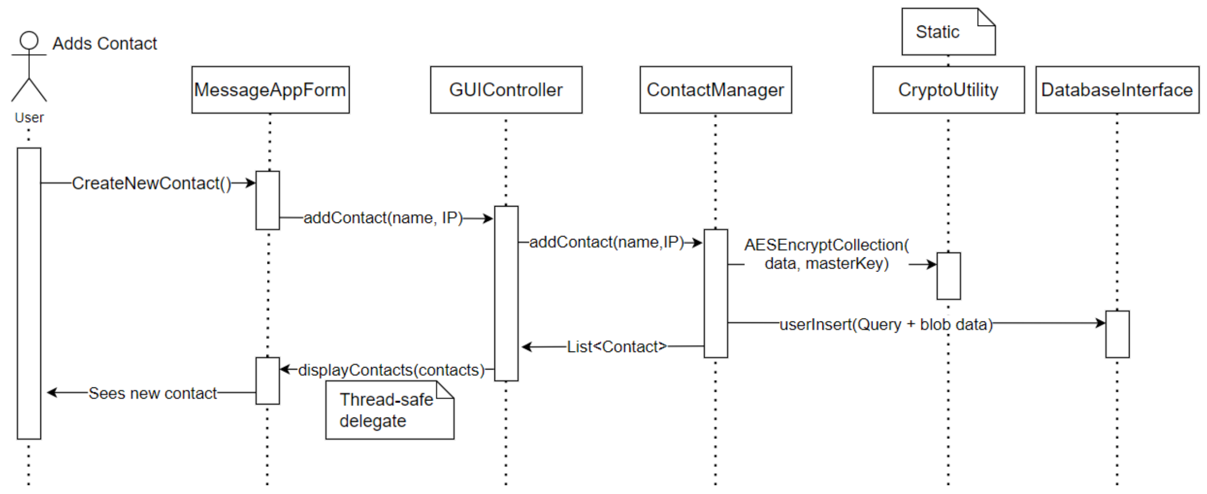


Figure 31: Object sequence diagram: What happens when the user adds a contact