# Secure mobile chat with message routing

Student Name: Daniel Randell
Student Number: C1120409
Supervisor: Dr. George Theodorakopoulos
Moderator: Mr. Michael W. Daley

Submitted: 5/5/2015

CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD

*Project 242*
*BSc Computer Science – Cardiff University*
*CM3203 – One Semester Individual Project*
*40 credits*

## *Abstract*

*This dissertation outlines the aims, designs and ultimately the implementation of the application CryptoChat that has been created for this project. The application has been developed with the aid of a variety of tools, such as Eclipse and the Android SDK using the Java & XML programming languages.*

*This application attempts to answer the questions of various security and privacy concerns that become threats to users when they communicate using public infrastructure and cellular networks. The project attempts to negate these issues from occurring by making use of direct communication with peers via WiFi Direct and using cryptographic measures, namely Diffie-Hellman to provide the secure communication exchanges. The application also details it use of the Spray and Wait network routing protocol and details all aspects of the project that have been successful and those that have not.*

## *Acknowledgements*

*With many thanks to my project supervisor Dr. George Theodorakopoulos for his support and guidance throughout my project. His comments and feedback helped keep me on track and in the right direction.*
*I would also like to thank my family and friends for their ongoing support during my studies at Cardiff University.*

## Contents

## *Figures*

# 1. Introduction

## 1.1 Preface

More people than ever before are now using “smart” devices for a variety of everyday tasks, the most prominent of these being communication applications that use Wi-Fi & Cellular networks to send messages. A simple look at the top applications in the Google App Store for example *[1],* show that the top two applications are in fact instant messaging applications. These applications are used practically on a daily basis by their users. This increase in digital communications data being sent over public networks makes communications data sent susceptible to attacks & methods of eavesdropping and interception which leaves it highly vulnerable and therefore making it a privacy and security concern for users.

The aim of this project is to therefore provide an alternative to the instant messaging applications available, the project aims to create a method of communication that is direct to each user’s device by making use of cryptography techniques and algorithms to provide security procedures so that data sent is better protected against the various methods of eavesdropping, interception and attacks that are in practice today. It will aim to do this with “1-to-1” communication and also with groups of users communicating. Secure communication exchanges will mainly be text exchanges however the project aims to implement secure picture messaging as-well. The project will also aim to implement the *Spray and Wait* network routing protocol so that message(s) sent to devices not available to communicate or out of range will still at some point in time receive the intended message(s).

Finally the project will aim to be light and efficient in how it handles such areas as *CPU usage*, *power consumption*, *end-to-end delay*, and *memory footprint*.

## 1.2 The Application

This report will focus on the core deliverable of this project, the application built. It will state and detail the methods used to create the application, the challenges faced during the development of the application and ultimately what was achieved during the time period spent on this project.

The application that was built is titled *CryptoChat* and was created for the Android operating system using the Android SDK in the Java & XML programming languages. All of the code (unless stated in this report or in the source code) was written by myself in the Eclipse IDE (Integrated Development Environment) on my desktop computer.

## 2. Background

The rise of smart "devices" namely smartphones in the developed and developing world has led to a huge change in how people communicate today. Coupled with the emergence of ever increasing cellular networks and public WiFi networks, more people than ever before are now communicating using a mobile device. It is however apparent that people are not using mobile devices to make phone calls, this research article carried out by the regulator Ofcom *[2]*, shows the widespread adoption of text messaging using services such as instant messaging and social media in the UK to communicate with others.

This increase in the amount of communications data now being sent across unsecured & public infrastructure networks means there is an increased risk in how that data can be intercepted, monitored and captured by unwanted third parties. We also have the dominance of data going to specific companies, namely Facebook and the Facebook owned "WhatsApp" which as this article states *[3]* "controls the future of messaging". Facebook owned products are used globally on a regularly basis by over 1 billion people worldwide *[3]*, to place that into perspective; ⅐ of the world's population intrusts their communications data to one company. It is through the use of these services that the world communicates, but it is also through the use of these services that government & intelligence agencies can "keep track" of its citizens and what they are communicating. The issue of privacy and security between how we communicate with each-other has never been more prominent than it is in today's digital world.

Who we communicate with, what we communicate to them and how frequently we communicate are extremely personal to us as individuals, not only are we expressing ourselves in some manner, information about ourselves is also sent along with what we communicate whether that be something as simple as a name or something more personal such as a location. This data can be used by unwanted parties in ways that most users do not even fully understand, it gives these people or agencies a glimpse into your life and for some individuals this is not something that they desire. This raises the question of how we can communicate with people that we want to talk too in a secure manner. That question is one this report and application will try to answer.

### 2.1 The Problem

When data is sent out over a public domain to a cellular network or to connect to some form of public network (such as the Internet) it becomes impossible to truly safeguard that data. The data being in the public domain can as stated above be monitored, intercepted and if required, captured by a third party *listener*. This use of public networks means we do not have definitive control over our communications data and importantly we do not know who is making use of these networks and listening in to what is being sent over them, therefore it raises various privacy and namely for this project, security concerns in using these networks.

To solve this problem *direct communication* with a party (an individual's smartphone) is the approach this project employs, using the technology standard developed by the WiFi Alliance called *"WiFi Direct"* to avoid the use of public infrastructure networks and talking to a party/peer directly. With the complete control over the flow of communications data exchanged between parties/peers, is how this project and the application to be developed attempts to negate the security & privacy concerns mentioned previously.

## 2.2 Stakeholders

Potential stakeholders for this project would be small businesses that require sensitive discussions to be held across a small office space using mobile devices running the Android operating system. It would provide an alternative, simple and ultimately free solution in providing a secure, mobile LAN communication service which in normal circumstances could cost thousands of pounds.

Limitations in the communications range severely limits the appeal of this application to the *average consumer* however in a local space and with the network routing protocol that this application implements it does provide an alternative means of communicating that some consumers may be interested in or currently looking for.

## 2.3 Existing Solutions

Various solutions exist that provide communications using the Android operating system & WiFi Direct such as *[4]*, there also exists solutions that provide group chat functionality *[5]*. The Android SDK itself provides a demo application demonstrating this technology in use *[6]*.

Throughout my research I could not however find any existing application that provided secure communication using WiFi Direct, nor could I find any existing WiFi direct solution that implemented the *Spray and Wait* network routing protocol.

## 2.4 Code Resources Used

The demo previously mentioned *[6]* provided by the Android SDK is the only core external code that was not written by myself for this application. The demo code was merged within my existing application after I had created a basic template UI.

The application implements the google provided *PRNG (Pseudo-random number generator) Fixes code [7]* as a security precaution for Android devices running Android 4.1-4.3. No third party security library was used (as stated would be in my Initial Plan).

# *3. Specification & Design*

## *3.1 Overview*

The core aim of this application was to ensure its functionality in regards to the *security* and *privacy* that it provided to users communicating with each other. The Android operating system was chosen to support this aim due it is wide popularity and available resources. The Android development landscape contains a vast array of guides, tutorials and code snippets online that I knew would make the Android operating system the best fit for this project.

For the implementation of the project, I plan on solely using the Android SDK in collaboration with the Java & XML programming languages. After further research I came to the conclusion that for the scope of this project and for the flexibility that I wanted, that having complete control over the code implemented and running on the application would be the best course of action therefore no external programming libraries were considered to be used. The application is designed to be used with any compatible WiFi Direct device running the Android Operating System 4.1+; WiFi Direct was introduced into the Android operating system in *Ice Cream Sandwich (version 4.0) [8]* however the application requires some functions only found in Android *Jelly Bean (version 4.1)*. It will be created & tested on mobile devices running different versions of the Android operating system to ensure functionality works across different operating system versions. The application will be optimised for a mobile device, but should also function on a larger device such as a Tablet.

## *3.2 Changes from Initial Plan*

After the completion and subsequent feedback that I received from my Initial Plan, I realised that some of the aims for the project were a bit ambitious considering the limited time-frame available for the project. Thus the *group messaging* functionality & *picture messaging* were made *"sub-core" aims; aims that if time permitted, I would work towards implementing.*

After some thought I also decided that it should be a main aim of this application to be *simplistic and user-friendly*, the application needed to have these properties to appeal more to potential consumers and to make it viable as an alternative communications solution to the '*mainstream'* applications available in the '*app stores*'.

## *3.3 Designs Principles*

The design plan for this project will follow several principles, as stated previously it must be *simplistic and user friendly.* The design will also follow the *minimalistic* principle, it will not provide or make use of many fancy effects or include many separate screens. It will be kept to a clear, linear navigational structure with minimal customisation options available.

The design emulates and takes inspiration from principles found in the latest Android L (*Lollipop) material design* philosophy *[9]*, it does not however follow all of them for example, the use of bright, vibrant colour is something *material design* emphasises, it is however not the colour scheme used in this project and drop shadows (an effect very prominent in *material design*) is kept to a minimum in this project in an attempt to be light on system resources specifically *battery & CPU consumption*.

## *3.4 Design Feature Functionality*

The overall approach of design to feature functionality in this application was to keep it ultimately *minimal and simplistic.* This falls in line with the aesthetic design principles mentioned above.

The *chat* experience in this project is the key feature to this application, it would however not suit well if the *peer discovery* was complex and overburdening on users. Therefore the aesthetic approach had to compliment both *chat & peer discovery* feature sets. The design of each screen had to give each unique feature set the user interface requirements needed so that each unique feature could present itself fully to the user in a functional but pleasant way. The application also was designed to offer users *minimal* feature settings and customisation options, this included the ability to have *themes* (for light & dark environments, also to conserve battery life), *notifications, vibrate* (on receiving messages) and *font size customisation.*

The visual designs that are included below in this report detail how the design of the feature functionality requirements and the design principles of this project have merged together to create the application design.

## *3.5 Visual Designs*

The following images dedicate the original design mock-ups for the project, each *figure* includes the heading of the screen state shown alongside annotations to further detail the design choices of each screen state. Also included are design mock-ups of the screen in various alternative states such as when an Android OS overlay is in place or when the keyboard is shown.

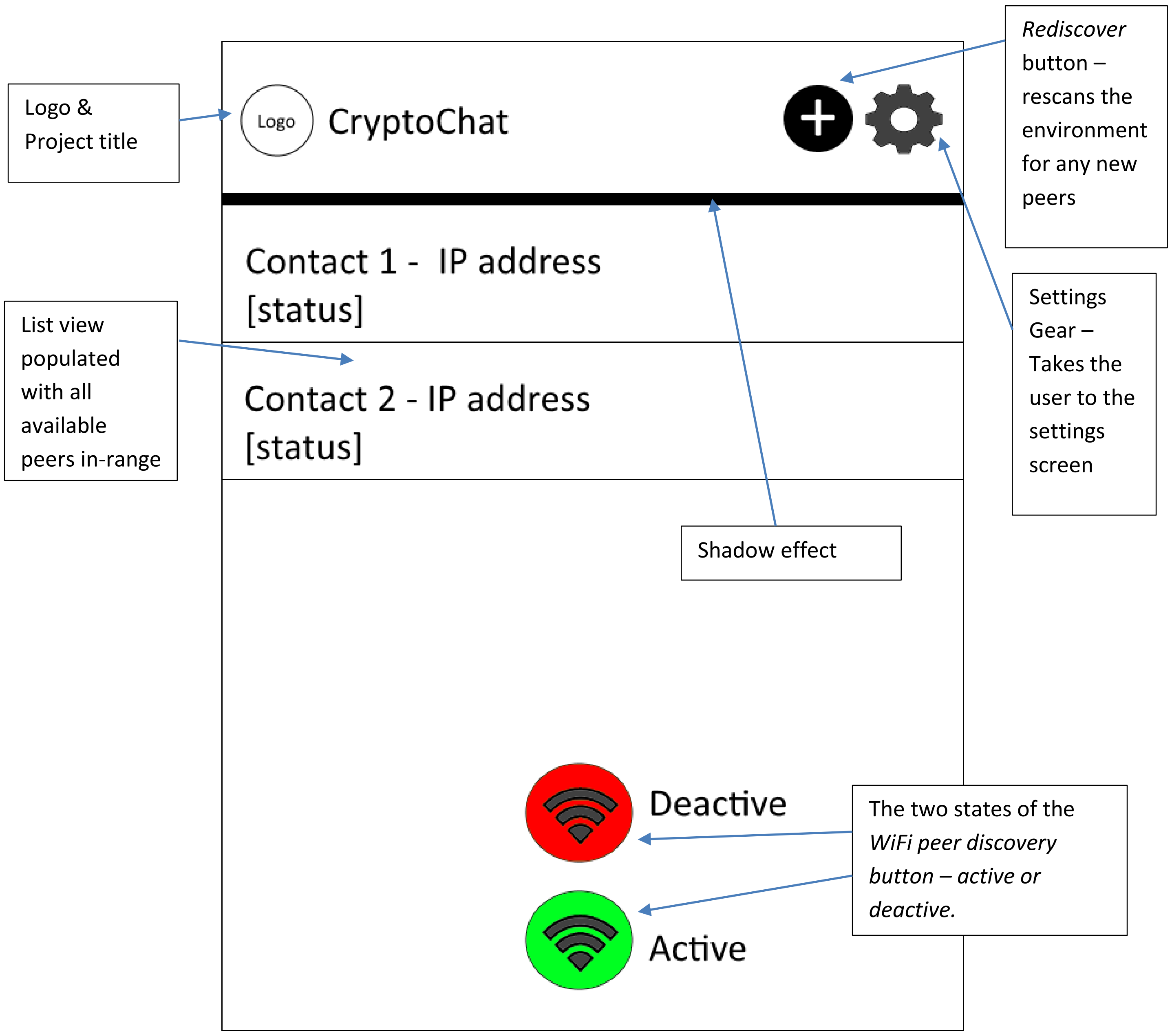

Logo & Project title
Logo
CryptoChat
*Rediscover* button – rescans the environment for any new peers
Settings Gear – Takes the user to the settings screen
Contact 1 - IP address [status]
Contact 2 - IP address [status]
List view populated with all available peers in-range
Shadow effect
Deactive
Active
The two states of the *WiFi peer discovery button – active or deactive.*

*Figure 1: Start Screen Design*

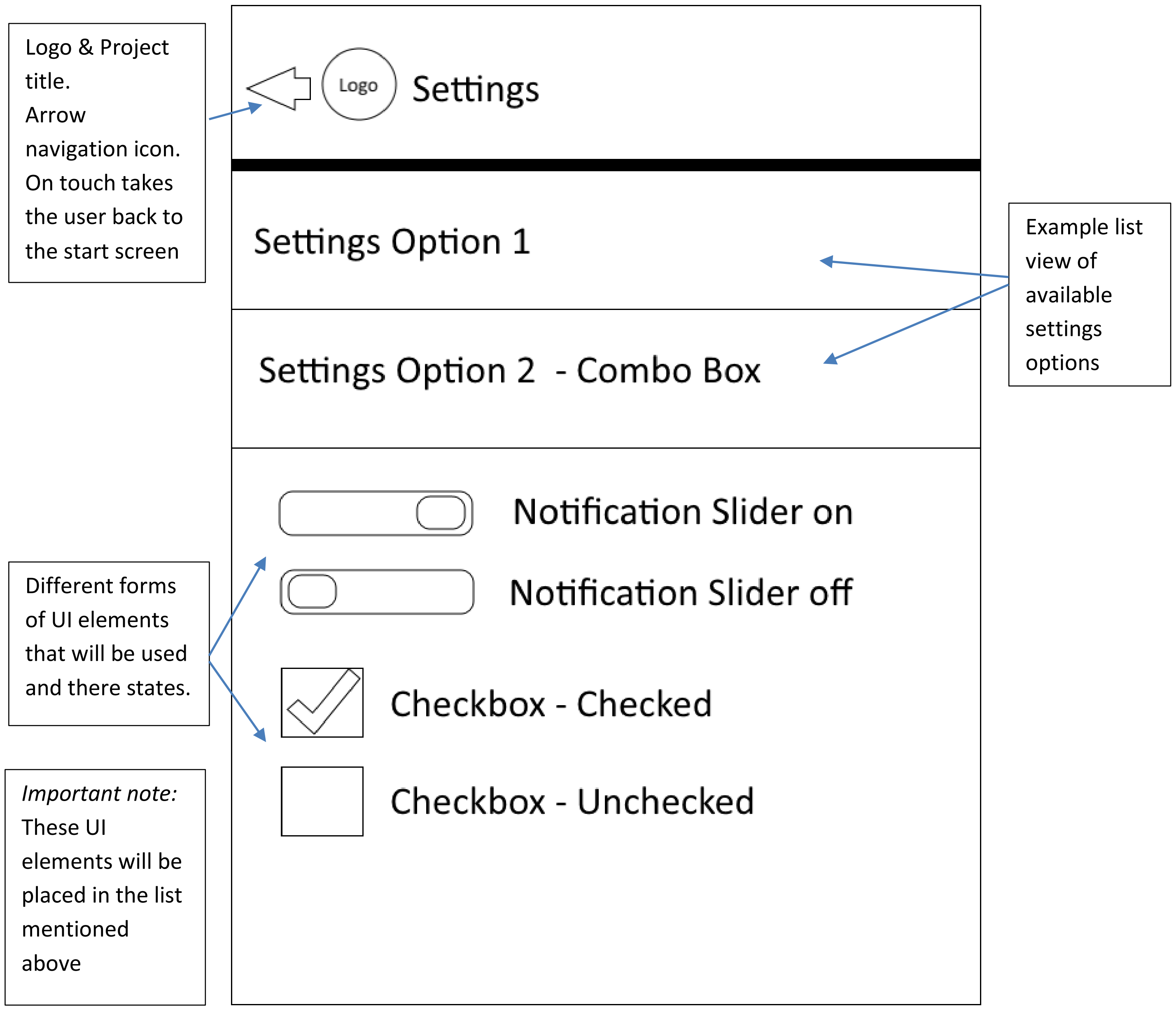

Logo & Project title.
Arrow navigation icon. On touch takes the user back to the start screen
Logo
Settings
Settings Option 1
Settings Option 2 - Combo Box
Example list view of available settings options
Notification Slider on
Notification Slider off
Checkbox - Checked
Checkbox - Unchecked
Different forms of UI elements that will be used and there states.
*Important note:* These UI elements will be placed in the list mentioned above

*Figure 2: Settings Screen Design*

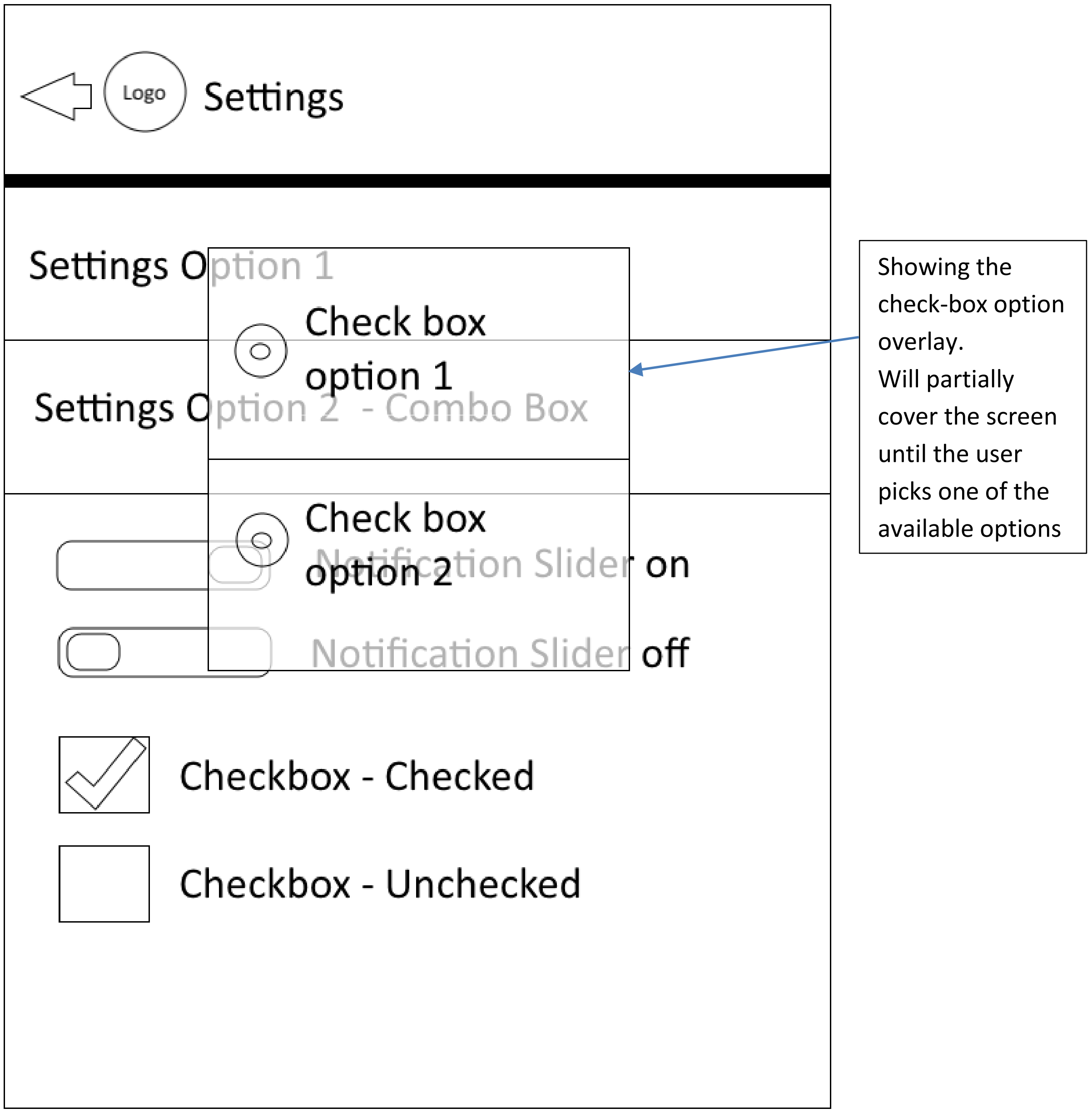
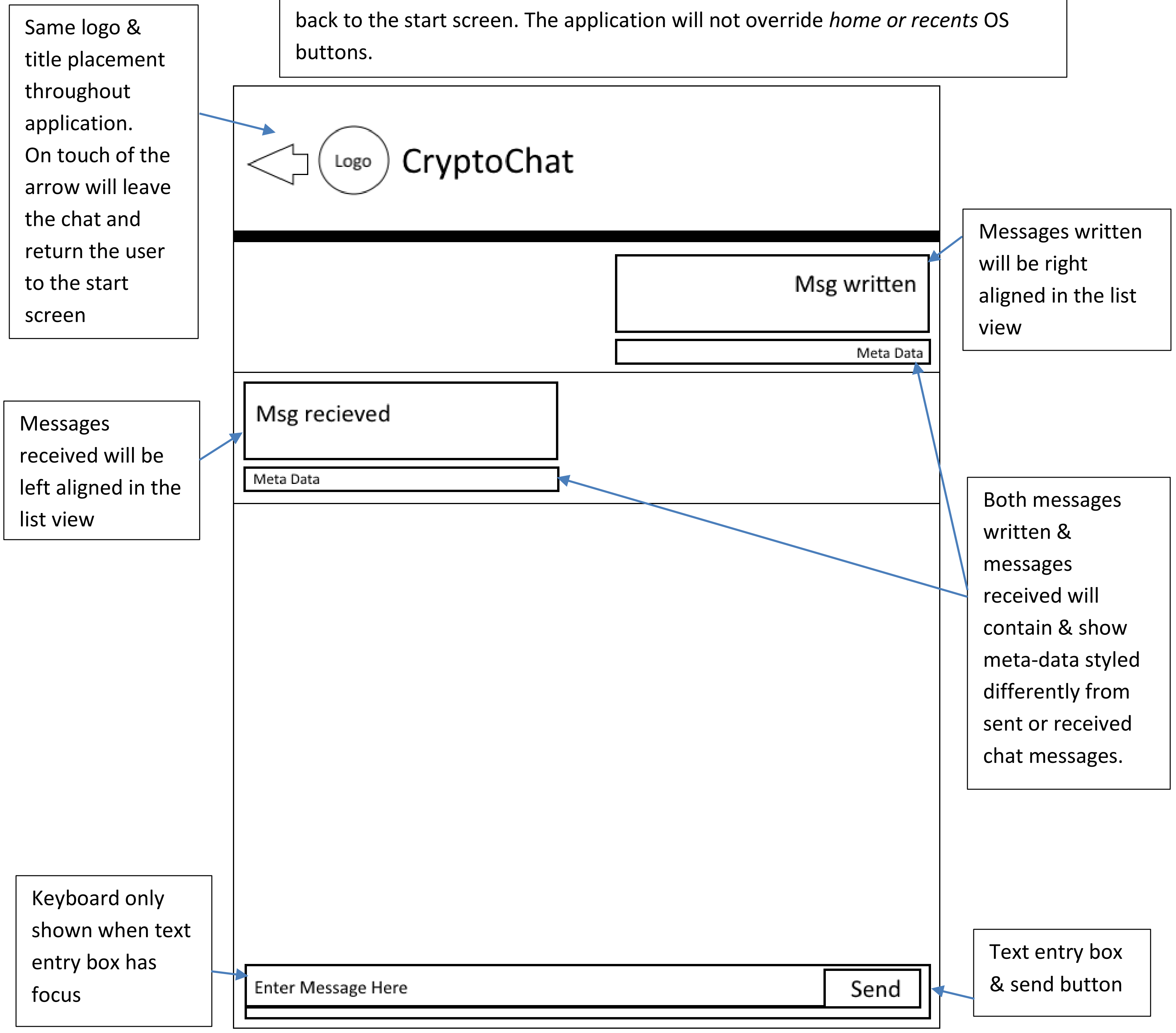


*Figure 3: Settings Screen Design with Android OS overlay*

*Important note:*
Verification will take place on arrow touch & on Android OS *back* button touch events which will be the two navigational methods of exiting the chat screen back to the start screen. The application will not override *home or recents* OS buttons.



Same logo & title placement throughout application. On touch of the arrow will leave the chat and return the user to the start screen

Messages written will be right aligned in the list view

Messages received will be left aligned in the list view

Both messages written & messages received will contain & show meta-data styled differently from sent or received chat messages.

Keyboard only shown when text entry box has focus

Text entry box & send button

*Figure 4: Chat Screen Design*

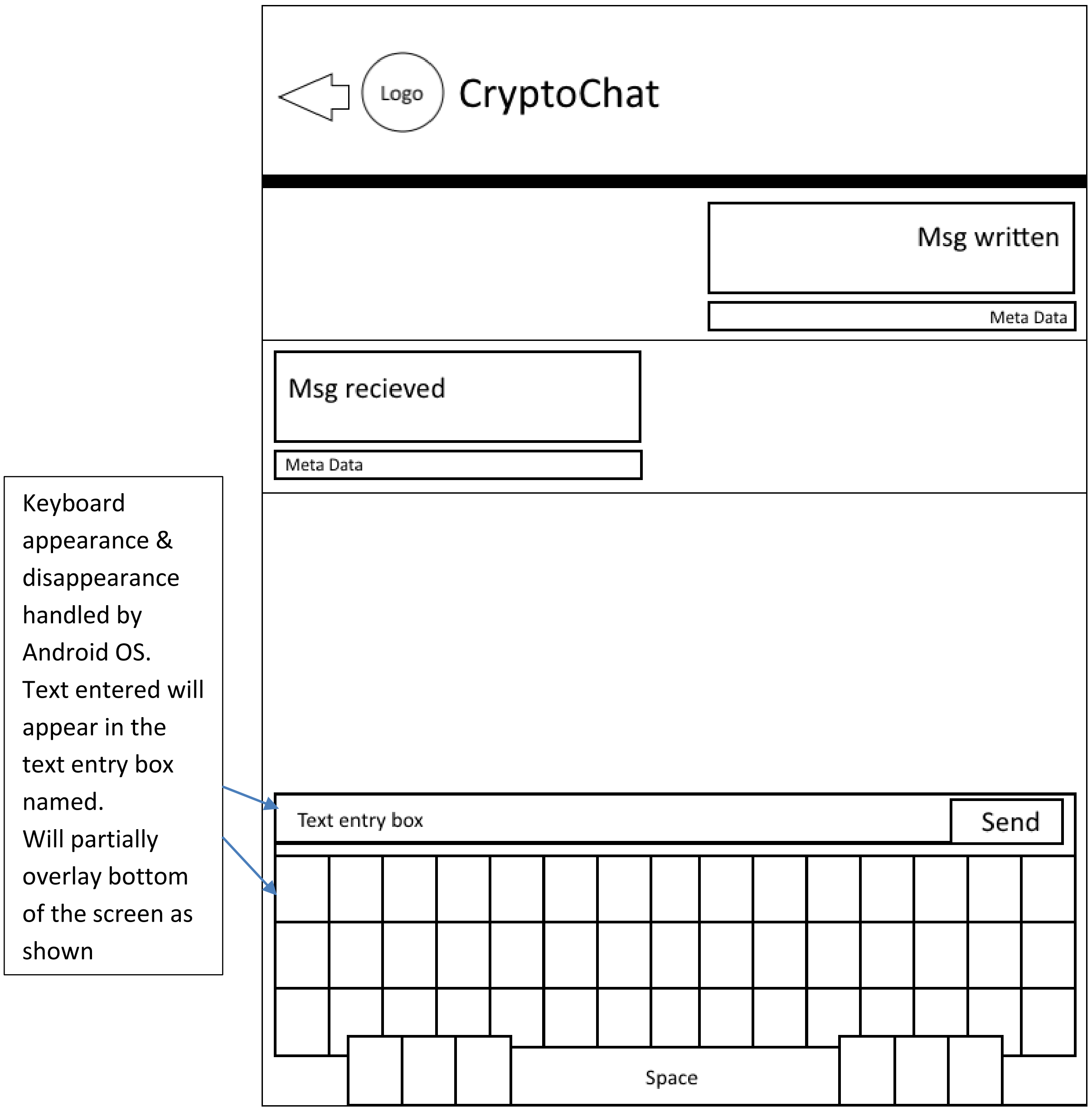

Logo
CryptoChat
Msg written
Meta Data
Msg recieved
Meta Data
Keyboard appearance & disappearance handled by Android OS. Text entered will appear in the text entry box named. Will partially overlay bottom of the screen as shown
Text entry box
Send
Space

*Figure 5: Chat Screen Design with keyboard*

## 3.6 System Requirements

The following table will detail the system requirements that have been agreed upon for this project, these will include the *functional and non-functional* requirements that will be implemented to ensure the application can deliver the experience required.

| Functional Requirements | Non-functional requirements |
|---|---|
| **Description:**<br>Must be able to have secure, direct communication with a party/peer. This may later be extended to include more than one party/peer. | **Description:**<br>Should adhere to HCI policies and the design principles set out in the design principles section of this report. It should adhere to some of the principles set forth in Android's *material design [9].* |
| **Description:**<br>Must consider the overall usage of system resource to ensure a *minimal memory footprint*, *minimal data use*, *CPU usage* and *power usage*.<br>The system will also consider such factors as *end-to-end delay times* between message exchanges and *throughput – e.g. the number of bytes sent per minute.* | **Description:**<br>Must ensure a consistent experience throughout the application, the layout and interactions of the application must be simple, straight-forward and easy to navigate & use. |
| **Description:**<br>Must implement the *Spray and Wait* network routing protocol for out of range message recipients using the *vanilla* version of the protocol *[10]*. | **Description:**<br>Messages must be sent with confidentiality and integrity intact. The privacy between messages is paramount thus the security measures that will be used must be able to secure the dialogue exchanged between users so that it can ensure and uphold the integrity of a *private* digital communication exchange between users and eventually groups of users. |
| **Description:**<br>To implement some customisation options and provide the option of external settings that can be alternated and changed to suit the needs of the user. These must be saved and kept so that a user can have the same experience every time he/she launches the application. | **Description:**<br>Must ensure that the style options and colour scheme(s) that will be incorporated within the application remain consistent throughout and adhere to the design principles set out for this project. |

## 3.7 Use Cases

After conducting research into this project and specifically looking at similar instant messaging applications available, I have created below the following use cases that will define & state how a user will interact with the application and how they will proceed to use the full feature set of this application.

| **Use Case:-** | **1. Connecting to a peer** |
|---|---|
| **Actors:-** | User A, User B. |
| **Description:-** | The two users (A & B) will start a connection attempt. |
| **Pre-Conditions:-** | Available peer(s) to make a connection attempt with. |
| **Basic Flow:-** | 1.) Peer discovery has completed and User A has found User B.<br>2.) User A touches the screen's list box container and presses on User B's entry in the shown list of available peers.<br>3.) User B will then receive a prompt to accept the connection.<br>4.) User B accepts the connection and the connection attempt is made. |
| **Alternate Flow:-** | 1A.) User B starts the process instead of User A and therefore roles are reversed.<br>4A.) User B declines the connection request and no connection attempt is made. |
| **Post-Conditions:-** | User A and User B are now connected with each other and can begin key exchange to then chat securely. |

| **Use Case:-** | **2. Send messages** |
|---|---|
| **Actors:-** | User A, User B. |
| **Description:-** | User A will send a message to User B. |
| **Pre-Conditions:-** | User A & User B have a secure connection.<br>*Dependency:: Connected to a peer* |
| **Basic Flow:-** | 1.) User A types a message to send to User B.<br>2.) User A presses the send button. |
| **Alternate Flow:-** | 1A.) User B starts the process instead of User A and therefore roles are reversed. |

| **Use Case:-** | **3. Disconnecting from a peer** |
|---|---|
| **Actors:-** | User A, User B. |
| **Description:-** | User A will disconnect from User B. |
| **Pre-Conditions:-** | User A & User B have a secure connection. *Dependency:: Connected to a peer* |
| **Basic Flow:-** | 1.) User A will have pressed on the back arrow button in the top corner of the screen to exit.<br>2.) User A will receive a prompt that will display a verification message on whether or not he/she wants to leave.<br>3.) User A accepts to leave the conversation and is returned to the start screen. |
| **Alternate Flow:-** | 1A.) User B starts the process instead of User A and therefore roles are reversed.<br>1B.) User A presses the Android OS *back* button to commence leaving the chat.<br>1C.) User A presses the Android OS *home* button to immediately leave the chat.<br>1D.) User A presses the Android OS *recents* button to see all open applications, in the process again immediately leaving the chat.<br>4A.) User A declines the prompt to leave the chat and therefore stays talking and the connection is still active. |
| **Post-Conditions:-** | User A will have left the chat. |
| **Post-Conditions:-** | User B will have received User A's message. |

| **Use Case:-** | **4. Alter Settings** |
|---|---|
| **Actors:-** | User A, User B |
| **Description:-** | User A will alter his/her settings |
| **Pre-Conditions:-** | User A will be on the start screen |
| **Basic Flow:-** | 1.) User A will press the gear icon to navigate to the settings screen.<br>2.) User A will observe and if he/she wishes alter the available settings that are present.<br>3.) User A will press the back arrow in the top left corner of the screen to return to the start screen. |
| **Alternate Flow:-** | 1A.) User B starts the process instead of User A and therefore roles are reversed. |

|  |  |
|---|---|
|  | 3A.) User A will press the Android OS *back* button to return to the start screen. |
| **Post-Conditions:-** | User A will be on the start screen and his/her new settings choices will have been saved. |

| **Use Case:-** | **5. Re-search for new peers** |
|---|---|
| **Actors:-** | User A. |
| **Description:-** | User A will re-search for any new peers that have become available. |
| **Pre-Conditions:-** | User A will be on the start screen. |
| **Basic Flow:-** | 1.) User A will press the *re-search* button in the action bar of the application to re-begin the discovery of peers' process. |
| **Alternate Flow:-** | None. |
| **Post-Conditions:-** | User A will have a populated list of peers that are available to connect with. |

| **Use Case:-** | **6. Store *wait* messages** |
|---|---|
| **Actors:-** | User A, User B. |
| **Description:-** | User A will store messages that will be sent to User B when they next connect. |
| **Pre-Conditions:-** | User A will be on the chat screen and User B will have disconnected.<br>*Dependency:: Peer disconnected* |
| **Basic Flow:-** | 1.) User A will type a message.<br>2.) User A will press the send button.<br>3.) User A will be notified of the message being saved upon exiting the chat. |
| **Alternate Flow:-** | None. |
| **Post-Conditions:-** | User A will have a file named after the disconnected peer's MAC address containing the stored *wait* message(s). |

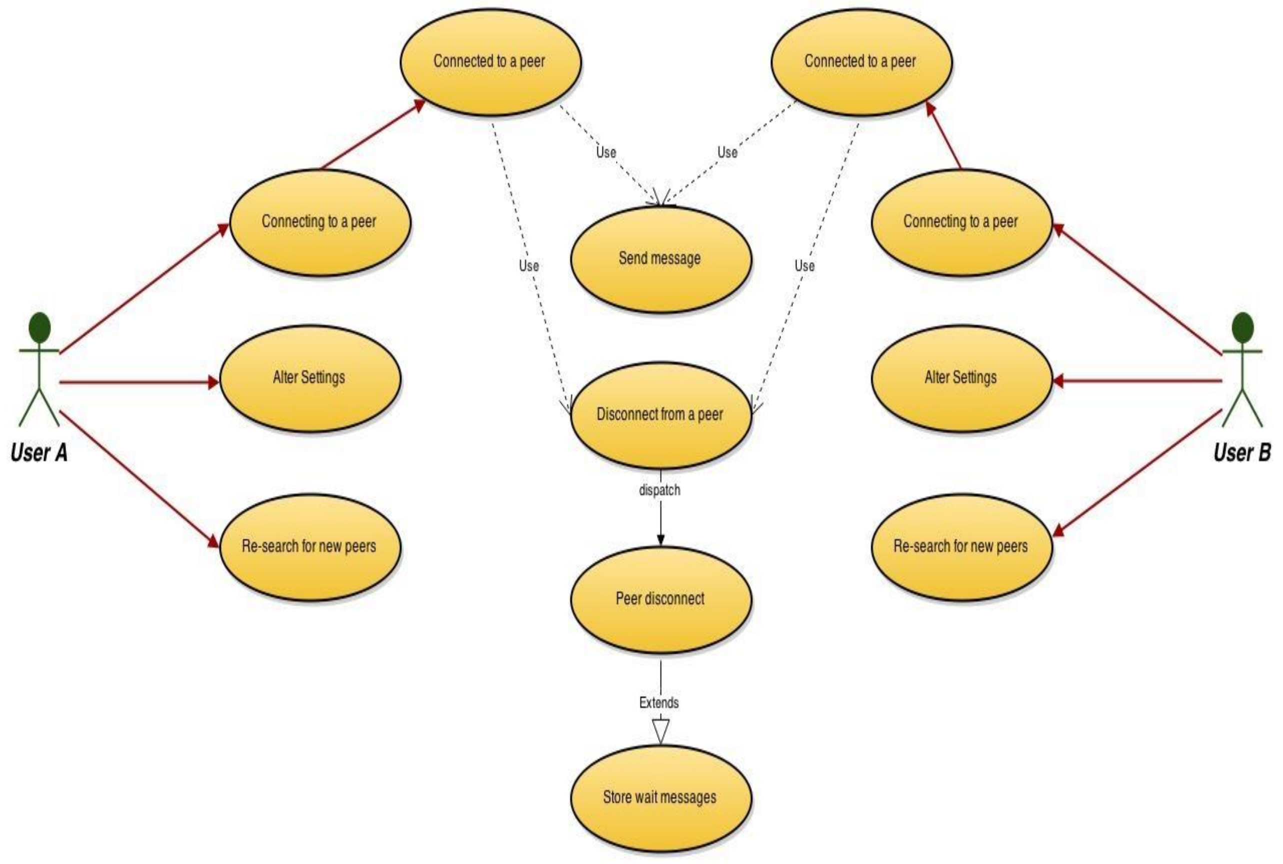


*Figure 6: Use-Case Diagram*[1]

The above use-case diagram details how the user(s) will interact with the system, and importantly with each other. Various states of the system are shown that require some further explanation for clarity. The use case *connecting to a peer* will enter the system into a *connected to a peer state*, this will enable the user to *send messages* or *disconnect from a peer*. Only when the system is in this state can the user execute those use-cases. Similarly for the use case *store wait messages* to be executed, a peer has to have disconnected and left the other user in the system state *peer disconnected.*

[1] Created using https://www.draw.io/

# 4. Implementation

## 4.1 Acknowledgments

The implementation of this project would not have gone as relatively smoothly as it did without the following resources that will be accredited.

As mention previously the WiFi peer to peer example application *[6]* gave a great starting point to incorporate into my existing project. It provided the basic functionality for the chat exchanges and saved roughly two-three weeks development time on this project. The documentation and guides that were provided by Google's Android developers' website, specifically the guide on WiFi peer-to-peer *[11],* helped immensely in providing point of reference code snippets and importantly understanding in how the underlying software technology functioned.

For the security side of implementation, the documentation provided by Oracle on the cryptography architecture of the Java programming language *[12]* again provided a resource which allowed me to gather a knowledge base on using these quite sophisticated & complex functions. The example code on the *Diffie-Hellman key exchange policy [13]* (which will be discussed in much further detail later) provided a solid understanding of how to implement the required security functionality.

The Eclipse IDE[2] (Integrated Development Environment) alongside the ADT (Android Development Tools) plugin[3] provided the core Windows development environment and kept the development of this project running smoothly. The Eclipse IDE in particular was very comfortable to me with hours of prior experience developing in Java, it allowed me to quickly set up the Android development environment and thus improved my productivity on this project.

Finally, Cardiff University's School of Computer Science gladly provided another device in the form of a Moto-E[4] smartphone that without this essential item, I would not have be able to carry out the testing and debugging of the application that was required.

## 4.2 Change after Final Design

Important to note one small change that occurred after the final design that became apparent during development. The *WiFi toggle button* which was originally centred was moved to the right side of the screen to avoid overlaying the list view.

[2] Eclipse available at https://eclipse.org/
[3] Android Development Tools plugin available at http://developer.android.com/sdk/installing/installing-adt.html
[4] Moto E product page at http://www.motorola.com/us/smartphones/moto-e-1st-gen/moto-e.html

## 4.3 Overview of CryptoChat UI

The implementation of CryptoChat initially started with the creation of the two core *Activities* (screen) files, these being the file *MainActivity.java* which actually handles (as referred to in this report) the start screen and the chat window. And the file *SettingsActivity.java,* which handles the settings screen. It is important to note now how Android and specifically the screen states interact and change. To create a dynamic UI in Android the use of *Fragments* are required, these are *"a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities)" [14]*. Therefore it can be said that an *Activity* defines the static environment of a screen state which include items such as the title bar containing the logo & options and generally things that don't change often or at all. A *Fragment* defines the content that frequently changes or content that should only be displayed when required. Thus with *Activities & Fragments* interacting with each-other, Android has the means to create vivid and complex dynamic user interfaces.

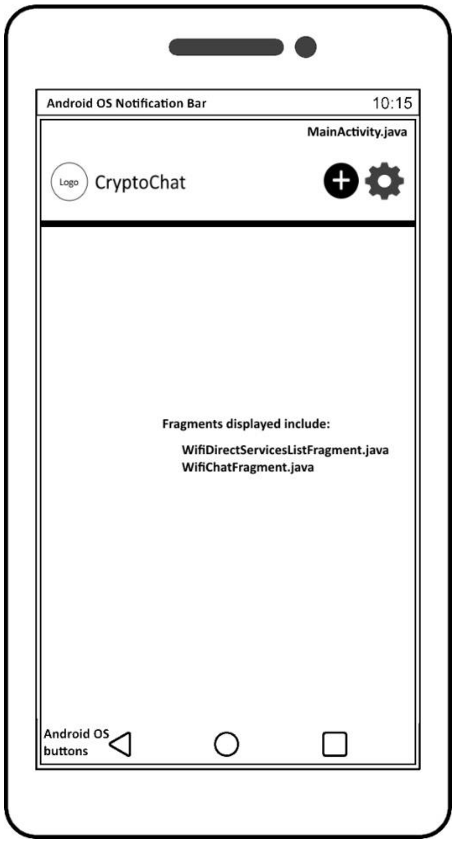

Android OS Notification Bar
10:15
MainActivity.java
Logo
CryptoChat
Fragments displayed include:
WifiDirectServicesListFragment.java
WifiChatFragment.java
Android OS buttons

*Figure 7: MainActivity.java and the fragments it contains*[5]

[5] Android phone template provided by http://cache.preserve.io/kpj5tgwj/index.html

*Figure 7* shown on the previous page indicates the fragments that *MainActivity.java* contains. To detail, *WifiDirectServicesListFragment.java* is a *List Fragment* (a special fragment type that will only contain a list view of items), this is used to display the available peers onto the start screen if the user has completed the peer discovery process.

*WifiChatFragment.java,* is as it states the chat fragment that will replace *WifiDirectServicesListFragment* when a user connects with another and begins to chat away. On disconnecting from a peer, that disconnecting user is return an empty *MainActivity* with the *WiFi toggle button disabled* waiting for the user to re-begin his/her peer discovery process by enabling (pressing) that button.

Similar, *Figure 8 (shown below)* indicates the fragment that *SettingsActivity.java* contains which is *SettingsFragment.java.* Note the choice of separating out the *MainActivity* and the *SettingsActivity,* since the *MainActivity* already contained the core functionality of the application I thought it was in the best interest of the code to separate out customisation options from the functionality of the application since ultimately the settings are an extension to the application. The *SettingsActivity,* as shown in my designs is only accessible from the start screen during the peer discovery phase or when the *MainActivity is empty (WiFi toggle button is disabled).*

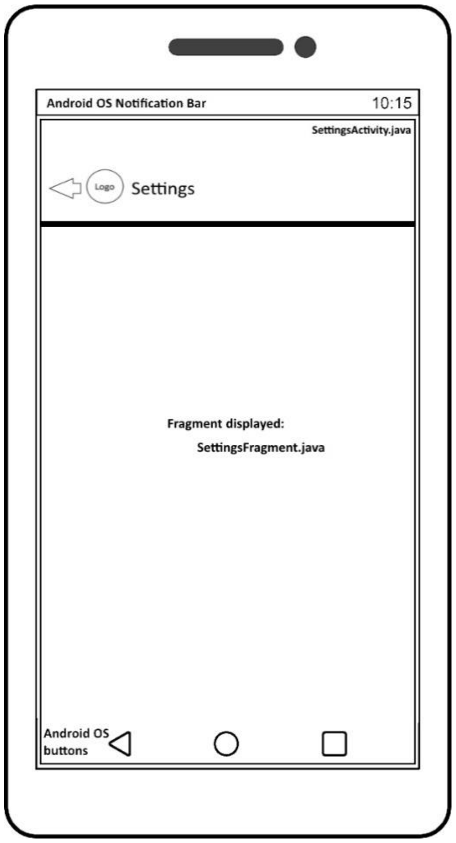


*Figure 8: SettingsActivity.java and the fragment it contains*[6]

[6] Android phone template provided by http://cache.preserve.io/kpj5tgwj/index.html

## 4.4 Overview of Cryptography Measures

The following section will give an overview of the cryptography measures that have been implemented into this application to provide security and privacy for a communication exchange. The project makes use primarily of the *Diffie-Hellman key exchange [15]* to generate a "common secret key" between parties and uses an AES (Advanced Encryption Standard) Cipher alongside CBC (Cipher Block Chaining) and PKCS7Padding (Cryptographic Message Syntax Standard) *[16]* for encryption & decryption of messages. The *Spray and Wait* network routing protocol makes use of two methods of encryption/decryption and will be detailed further on is this report.

### 4.4.1 Diffie-Hellman Key Exchange

The *Diffie-Hellman key exchange [15]* is vital to this project and is the underlying foundation towards achieving the security and privacy required for sending messages between parties/peers. The key exchange process takes place in this application after a connection attempt has been successfully made between the parties/peers.

The architecture of the connection will now need to be detailed. During the connection attempt one of the devices is defined as the *Group Owner,* he/she acts as the server to which the other will connect as the *Client.* The code for these socket set-ups was provided by the demo application *[6]* and can be found in the files; *GroupOwnerSocketHandler.java* & *ClientSocketHandler.java.* On the device which has been chosen as the *Group Owner,* that device initialisation the key exchange.

The *Group Owner* defines the prime root *p* and the primitive root modulo *q*, he/she will then generate their public & private key. The private key is a randomly generated value *x satisfying* $0 < x < p-1$. From the private value, the public key value y is created by doing $y = (q^x) \pmod p$ *with* $0 < y < p$. The group owner then sends this public key to the *Client.*

When the *Client* receives our *Group Owner's* public key it is important to note that he/she does not know our primitive root *p* and primitive root modulo *q,* these are sent alongside the key. The *Client* then generates their public key & private key using the same process as above, and will finally now generate his/her *secret key.* Using the *Group Owner's public key (y1)* we generate our secret key: $s = (y1^x) \pmod p$. The *Client* now has a secret key, he/she finally sends his/her public key to the *Group Owner.*

The *Group Owner* receives the *Client's public key (y2)* and already knows *p & q*, and his own secret value *x* thus he/she can also create his/her secret key: $s = (y2^x) \pmod p$. The key exchange process is now complete. The key exchange remains secure because the secret value *x (*held by the *Group Owner & Client)* which is required to derive *s* is not transmitted across the connection. Is it important to note here that we generate a common secret with a party/peer, but we do not authenticate who that party/peer claims to be.

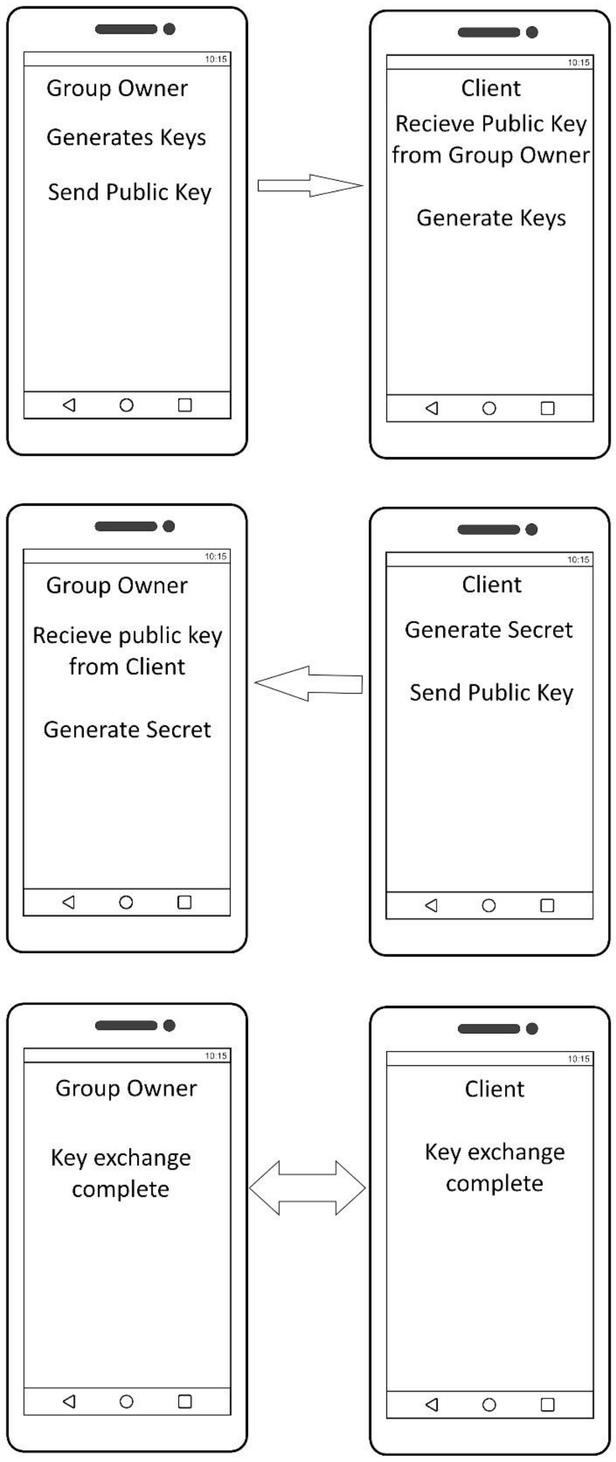


*Figure 9: Key Exchange Diagram*[6]

### *4.4.2 AES Cipher*

The Cipher chosen for the encryption and decryption methods is an AES (Advanced Encryption Standard) Cipher, it is a symmetric block cipher for encrypting texts which can be decrypted with the original encryption key, perfectly suited to what this project requires. The encryption & decryption methods both use the above *Diffie-Hellman secret key.* The *secret key* generated is in-fact much larger than the maximum key bit length that this Cipher (256 bits) can use, therefore the *secret keys* that are generated are *hashed* using SHA-256 to produce the required 256 bit *secret key* length.

The Cipher makes use of CBC which is short for *cipher block chaining* as its mode. The mode consists of an initialization vector which you XOR the first block of plaintext against. Then the block of plaintext is encrypted. The next block of plaintext is XOR'd against the last encrypted block before you encrypt this block and the process repeats. In this application the IV is randomly generated upon each new encryption, it is sent along with the encrypted message. The IV itself is not encrypted and is extract from the first 16 bytes of the data package sent (a data package consists of an IV & ciphertext).

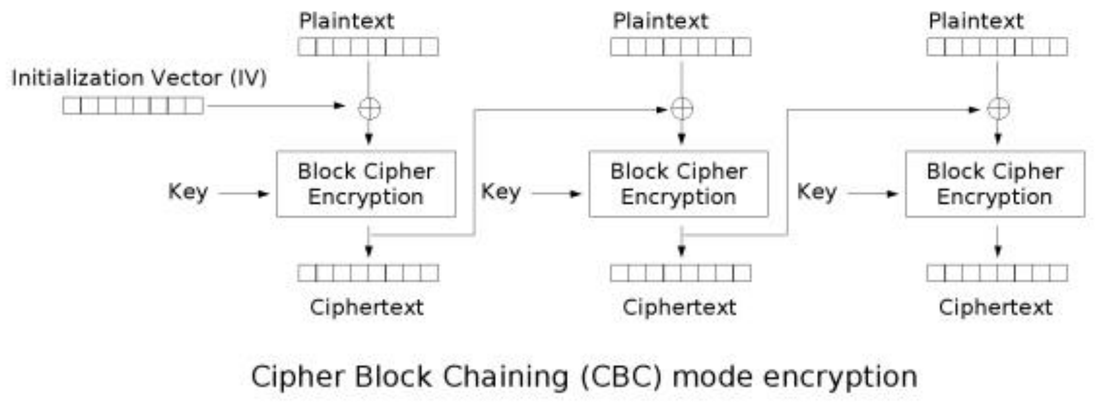


*Figure 10: CBC Mode Diagram*[7]

Finally, PKCS7Padding is used to apply padding (extra data) to the input data so that every message size that is encrypted is a power of 8 (bytes), this is required by the AES cipher to encrypt and decrypt successfully.

[7] Image provided by Wikimedia commons - http://commons.wikimedia.org/wiki/File%3ACbc_encryption.png

## *4.5 Overview of Spray & Wait Routing Protocol*

This application implements the Spray & Wait network routing protocol, specifically it makes uses of the *vanilla* version *[10]* of the protocol. It is however not a necessarily 'true' implementation at-least in its current form. The protocol states that when we enter the *wait phase* whereby wait messages are created, that they are devised a number, that number represents the number of copies that are allowed to be sent to *L* distinctive relays during the *spray phase [10]*. The implementation in this application does not send out any copies but they are rather held locally in a file thus therefore the number of distinctive relays is currently set to be a maximum of 1.

It is important to clarify how the creation of wait messages are implemented in this application. On the disconnection of a connected party/peer, the disconnecting party/peer is simply returned to the start screen. The other party/peer will still be on the chat screen, he/she is notified of the disconnection and any messages that are then typed & sent from that device will be stored. When that party/peer decides to disconnect from the chat exchange, the wait messages are saved into a file named after the locally administered MAC address of the party/peer than had previously disconnected. These messages are encrypted using a simple *symmetric key*.

Whenever the party/peer that had disconnected comes into range again of the user holding the *wait message file*, we verify that the locally administered MAC address that is being broadcast matches the one named after our *wait message file* and then if they match a connection attempt is made. If this is successful, the wait messages are decrypted using the *symmetric key* and stored locally. The file is then deleted. The key exchange process will take place and finally both user's will be notified of the wait messages being sent or received obviously the user receiving the message will see them but the user's sending will simple get a message appear saying *"wait message sent" (*this message will be displayed for the number of messages sent i.e. 3 wait messages sent, the notification of the sending will appear 3 times). Important to note that the *wait messages* are encrypted using the *DH (Diffie-Hellman) secret key* and therefore are always sent securely, and the *wait message file* itself is never sent but stored locally. The *wait message file* is stored inside the cache of the application alongside the saved settings, the file can only be accessed with root access enabled on a device.

This approach therefore retains its integrity since we don't compromise the relevantly insecure *wait message file* and we also implement the *verification* of the wait messages by comparing the name of the file (which again will be the locally administered MAC address of the peer/party that disconnected) with that of the locally administered MAC address that is being broadcast to our device holding the wait messages therefore we make sure the messages are sent to the correct party/peer. The protocol works regardless of the party/peer's status i.e. whether their device is *Group Owner* or *Client* and multiple wait messages can be stored & sent/received*. Figure 11* demonstrates the protocol in action.

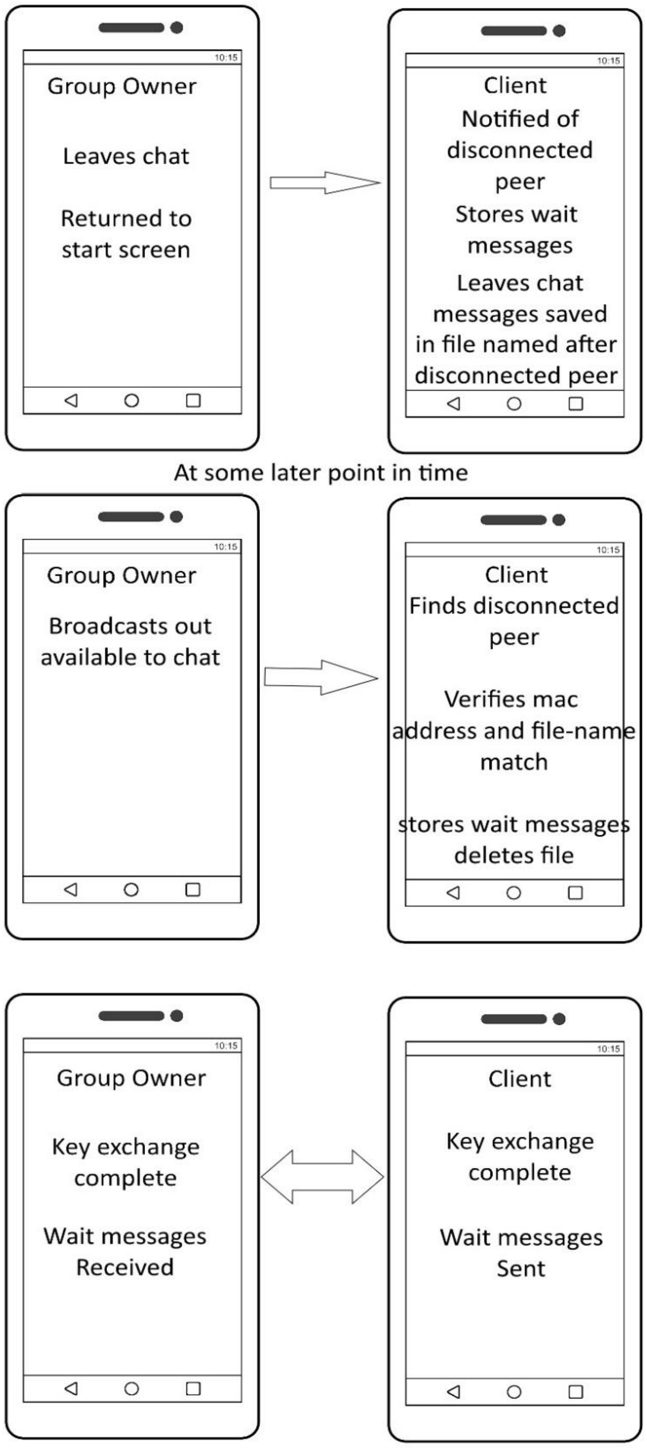

10:15
Group Owner
Leaves chat
Returned to start screen
Client
Notified of disconnected peer
Stores wait messages
Leaves chat messages saved in file named after disconnected peer
At some later point in time
Group Owner
Broadcasts out available to chat
Client
Finds disconnected peer
Verifies mac address and file-name match
stores wait messages deletes file
Group Owner
Key exchange complete
Wait messages Received
Client
Key exchange complete
Wait messages Sent

*Figure 11: Spray & Wait Network Routing Protocol Diagram*[6]

## 4.6 Problems Encountered

### 4.6.1 Connected Peers & the Wait Message File

One of the slight issues that occurred during the implementation was the ability to find out who each device was connected with, this was due to the none-exchange of actual contact information. The initial implementation of the application simple got the universal MAC that was located and shown in the settings of the Android phone. This turned out to be an incorrect solution since on connection to another device the MAC addresses are then *locally administered,* therefore the second least significant-bit of the most significant byte of the address changes. This bit is also referred to as the U/L bit, short for Universal/Local, which is defined by the IEEE standards as; "The Universally/Locally administered address bit is the next bit following the Individual/Group address bit. The U/L bit indicates whether the MAC address has been universally or locally assigned" *[18].* This was a particular problem for the saving of the *wait message file* since the original universal MAC address that I would collect would be different from the one used to save the *wait message file* and thus when it came to comparing addresses, they would never be the same.  To solve this issue, the *locally administered* MAC addresses are exchanged between devices during the key exchange process so that the *wait message file* would be saved with the correct address.

### 4.6.2 Initialisation Vector

The initialisation vector is used during the encryption and decryption process of messages & *wait messages* that are exchanged after the creation of the DH *secret key.* It is used to apply randomness to the AES Cipher when using CBC, this was for the initial weeks of implementation a simple 256 bit length string of the same value. It became apparent after researching (and specifically this article *[19]),* that the IV should be purely random. Thus to solve this I had figure out how to pre-append the IV to a data package so that on decryption, the message can be decrypted successful. Without the same IV used during encryption, the decryption would fail. The IV is not encrypted and is separate from the encrypted message in the data package.

The Eclipse IDE then produced another slight problem, the apparent security threat of using the pseudo random number generator (to generate my random IV) used in Android OS version's 4.1-4.3 (versions this application supports), the article in *[7]* documents the threat and provided the code to apply to fix this issue.

## 5. *Source Code Explained*

### 5.1 *Key Exchange*

The following code snippet can be found in *MainActivity.java*, in the function *onConnectionInfoAvailable(WifiP2pInfo p2pInfo),* this code executes when a connection attempt is made, here we see that the key exchange is initialised by whoever is set to be the *Group Owner,* the *threadHandler* is a Thread that handles the server/client connection:

```
if (p2pInfo.isGroupOwner) {
            Log.d(TAG, "Connected as group owner");
            isGroupOwner = true;
            isAPeer = false;

            try {
             threadHandler = new GroupOwnerSocketHandler(((MessageTarget)
this).getHandler());
             threadHandler.start();
            } catch (IOException e) {
                Log.d(TAG, "Failed to create a server thread - " + e.getMessage());
            }

            // Initialise our key so that it can be sent to the peer(s)
            try {
             myKey = new DHKeyAgreement();
             myKey.initalise();
             } catch (Exception e) {
                    // Error initialising key, log what went wrong
                    Log.e(TAG + "- Key Init error: ", e.toString());
             }
        } else {
            Log.d(TAG, "Connected as peer");
            isGroupOwner = false;
            isAPeer = true;

            threadHandler = new ClientSocketHandler(((MessageTarget)
this).getHandler(),
                    p2pInfo.groupOwnerAddress);
            threadHandler.start();
        }
```

*ChatManager.java* is the core file that handles the sending of messages, it contains the class *ChatManager* which is defined as a Runnable object, which runs on a Handler (“A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. “Each Handler instance is associated with a single thread and that thread's message queue. When you create a new Handler, it is bound to the thread / message queue of the thread that is creating it -- from that point on, it will deliver messages and runnables to that message queue and execute them as they come out of the message queue.

There are two main uses for a Handler: (1) to schedule messages and runnables to be executed as some point in the future; and (2) to enqueue an action to be performed on a different thread than your own *[17]")*.

When we set the *ChatManager* object (*code found in MainActivity.java line 682-684)*:

```
    @Override
    public boolean handleMessage(Message msg) {
...
       case MY_HANDLE:
                  Object obj = msg.obj;
                  (chatFragment).setChatManager((ChatManager) obj);
```

To the *MainActivity* thread's MessageQueue, it is at this point, that our *Group Owner's* key is sent to another device;

```
    public void setChatManager(ChatManager obj) {
        chatManager = obj;

        // Send (Group Owner)Alice's public key to (Peer)Bob
        if (MainActivity.isGroupOwner) {
             byte[] address = MainActivity.myMacAddress.getBytes();

              // Pre-append the MAC address to public key
              byte[] combined = new byte[address.length +
MainActivity.myKey.keyPair.getPublic().getEncoded().length];
              System.arraycopy(address, 0, combined, 0, address.length);
              System.arraycopy(MainActivity.myKey.keyPair.getPublic().getEncoded(),
0, combined, address.length,
MainActivity.myKey.keyPair.getPublic().getEncoded().length);

            chatManager.write(combined);
        }
    }
```

`public boolean handleMessage (Message msg)found in MainActivity.java, Lines 524-690` is the method that handles the receiving of *Message* objects. Each object contains the following variables;

```
// Contains user-defined message tags to define each message
Public int what;

// For the purpose of this application it used to store the byte data of each
// messages sent
Public Object obj;

// Arg2 is not made use of. Arg1 simple keeps track of the byte size of each data
// package
Public int arg1;
Public int arg2;
```

The remaining *Key Exchange* code can be found in MainActivity.java inside the function `public boolean handleMessage (Message msg)` inside the case - `case KEY_EXCHANGE, Lines 591-661` which details how both the *Group Owner* and *Client* handle their respective data packages containing the locally administered MAC address of each other's device and importantly the public key that will be used to create the DH *secret key* used later in encryption/decryption.

## 5.2 Spray & Wait Routing Protocol

In the function `public boolean handleMessage (Message msg) again found in MainActivity.java, Lines 524-690` we have the code for the beginning of the *spray phase* which occurs after a peer disconnects*;*

```
case DISCONNECT:
             if (isInChat) {
                    // The group owner should have received a copy of a peer's MAC
address in key exchange
                    // A peer will have gotten his copy when the group owner sent
over his public key
                    // Thus both should have copies of each others MAC addresses
                    (chatFragment).pushMessage("Buddy Disconnected");

                    if (connectedPeers.get(0) != null) {
                           (chatFragment).pushMessage("Mac: " +
connectedPeers.get(0));
                           // We are now in the spray phase
                           SPRAY_PHASE = true;
                           // Create the file to store the "wait" messages
                           waitIncrement = 0;
                           waitMessages.clear();
                           waitFile = new File(getFilesDir() + "//" +
connectedPeers.get(0));
                    }
             }
             break;
```

We create our *wait message file*, clear out a *List* of strings containing any previous *wait messages* and reset a wait increment value that is used during a while loop for the sending of any *wait message(s).*

When a user then disconnects and has *wait messages* stored, the method `public void disconnect () found in MainActivity.java Lines 886-934`, saves the *wait messages* to the *wait message file;*

```
if (waitMessages.size() > 0) {
             String encryptMsg = new String();

             for ( String msg : waitMessages)
                    encryptMsg = encryptMsg.concat(msg);

             try {
                    encryptMsg = crypto.encrypt(encryptMsg,
WAIT_SECRET_KEY.getBytes());
                    } catch (Exception e) {
                          // TODO:
                          e.printStackTrace();
                    }
             // Save Encrypted Message to device in file name of disconnected peer
                    Utils.writeToFile(waitFile, encryptMsg.getBytes());
                    waitMessages.clear();
        }
```

Therefore we now have the *wait message file* present on the device, once peer discovery is again re-enabled and our device is actively searching for other user's, any new user that our device comes across will have its MAC address compared with the filename of the *wait message file* in the following code;

```
               @Override
              public void onPeersAvailable(WifiP2pDeviceList peerList) {
...
                   // On a peer change i.e. a new peer has come into service
discovery or has left it
                   // We get a list of all the available devices that are in range
                   // The "Spray & Wait" protocol comes into play here
                   // If a device's MAC address corresponds to a file available on
the device
                   // Then we have messages that are needed to be delivered
                   // Thus we notify our device that this is the case
                   for (int i = 0; i < peers.size(); i++) {
                    boolean waitMessages =
checkForWaitFile(peers.get(i).deviceAddress);
                    WiFiP2pService service = new WiFiP2pService();
                     service.device = peers.get(i);
                     service.instanceName = "cryptochat";
                     service.serviceRegistrationType = peers.get(i).primaryDeviceType;

                    if (waitMessages) {
                          MainActivity.WAIT_PHASE = true;
                          connectP2p(service);
                    }
                   }
              }
          };
```

The *checkForWaitFile method* simple checks if the file exists, if it does then it decrypts, stores and deletes the *wait message file,* if it doesn't exist it simple does nothing.

We then simple keep track of the fact we have wait messages to deliver (if *checkForWaitFile returns true)* with the Boolean *WAIT_PHASE* and make the connection to that device. Finally in the case - `case MESSAGE_READ:` the wait messages are sent, this case specifically only occurs after the key exchange process has been completed;

```
else if (WAIT_PHASE) {
                    (chatFragment).pushMessage("Messages to send");

                    while (WAIT_PHASE) {
                           String waitMsg = null;

                           try {
                                  if (waitIncrement < waitMessages.size()) {
                                         waitMsg =
MainActivity.crypto.encrypt(waitMessages.get(waitIncrement), SECRET_KEY);
                                                (chatFragment).pushMessage("Wait
Message sent");
                                  } else {
                                         WAIT_PHASE = false;
                                         waitMessages.clear();
                                         break;
                                  }
                           } catch (Exception e) {
                                         e.printStackTrace();
                                  }
                           chatFragment.sendMessage(waitMsg.getBytes());

                           // Sleep the thread, giving a small fraction of time in-
between broadcasts
                           try {
                                  Thread.sleep(500);

                                         } catch (InterruptedException e) {
                                                // This is a very VERY bad exception
                                                e.printStackTrace();
                                         }
                           MainActivity.waitIncrement++;
                    }
             }

             break;
```

We encrypt the wait message with the AES Cipher that is located in the *Crypto class* (detailed below) with the *secret key* generated from DH. We then send the byte data to our *ChatHandler* object which is located in the *chatFragment* that then sends the data to the other device. We importantly sleep the thread to give a slightly needed delay between data bursts sent out since otherwise data packages (messages) would merge together.

## 5.3 Crypto Class

The *Crypto* class as mentioned above is located in the *Crypto.java* file, it contains the AES Cipher's encryption and decryption methods;

```
// Encryption method
public String encrypt(String plainText, byte[] encryptionKey) throws Exception {

		SecretKeySpec key = new SecretKeySpec(encryptionKey, "AES");
		iV = new IvParameterSpec(generateIV());
		cipher.init(Cipher.ENCRYPT_MODE, key, iV);
		// Encrypt msg before attaching IV;
		byte[] encrypted = cipher.doFinal(plainText.getBytes("UTF-8"));
		byte[] data = new byte[iVLen + encrypted.length];
		// Append together plain IV and encrypted cipher text
		System.arraycopy(iV.getIV(), 0, data, 0, iVLen);
		System.arraycopy(encrypted, 0, data, iVLen, encrypted.length);

		return Base64.encodeToString(data, Base64.DEFAULT);
}
```

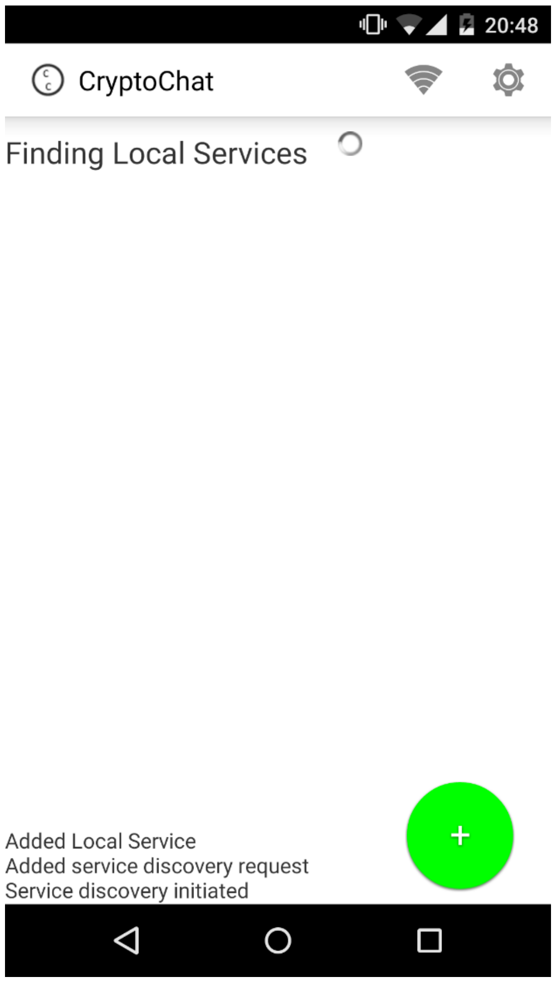
```
// Decryption method
public String decrypt(String cipherText, byte[] encryptionKey) throws Exception {
		// Get the initialisation vector used during encryption
		byte[] ciphertextBytes = Base64.decode(cipherText, Base64.DEFAULT);
		iV = new IvParameterSpec(ciphertextBytes, 0, iVLen);
		// Separate the IV bytes from the actual encrypted message
		ciphertextBytes = Arrays.copyOfRange(ciphertextBytes, iVLen,
ciphertextBytes.length);
		// Create our secret key and finally decrypt the message
		SecretKeySpec key = new SecretKeySpec(encryptionKey, "AES");
		cipher.init(Cipher.DECRYPT_MODE, key, iV);
		String decrypt = new String(cipher.doFinal(ciphertextBytes), "UTF-8");

		return decrypt;
}
```

The above methods detail the creation of a *SecretKeySpec* provided by the byte array *encryptionKey* (which for all messages sent across the connection is our DH *secret key)* and it shows how the IV is both appended and extracted in the respective methods. The only function to note is *generateIV ()*; note – IV is always 16 bytes in length, iVLen = 16

```
// Generate a random initialise vector for encryption/decryption
@SuppressLint("TrulyRandom")
public byte[] generateIV() {
	SecureRandom random = new SecureRandom();
	byte[] ivBytes = new byte[iVLen];
	random.nextBytes(ivBytes);
	return ivBytes;
}
```

## 6. Results and Evaluation

The project implementation meets all of the core requirements of functionality and design, it also implements some of the sub-core aims that were set out in the ‘Specification & Design’ section of this report. The *group messaging* and *picture messaging* sub core aims of this project proved to be a step too ambitious for the scope of this project, mainly due to the limited time-frame available coupled with my minimal knowledge of Android development, the learning curve (to get up to speed) took somewhat longer than had originally anticipated. However, the project and namely the application does achieve what it has originally set to-do; to provide secure communications without the use of any network infrastructure and importantly most features were implemented including the spray and wait network routing protocol along with all of this being done in a very friendly & user centred approach.

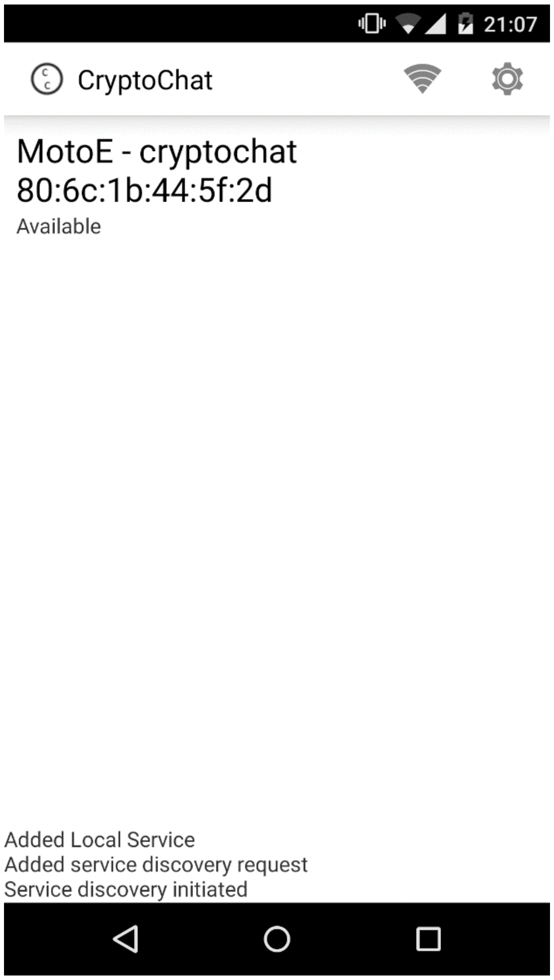


Added Local Service
Added service discovery request
Service discovery initiated

Figure 12: CryptoChat Start Screen – Peer search active

*Figure 12* shows the developed start screen, the application follows the design principles of being user-friendly and minimal. The *WiFi enable button* is enabled in the screenshot and thus peer discovery is active, if it was pressed again it would appear red and the user would be informed he/she is no longer in active peer discovery. The gear icon will take the user to the settings page, the WiFi icon will re-search for any new peers.

*Figure 13* shows the look of the start screen after a peer has been found;



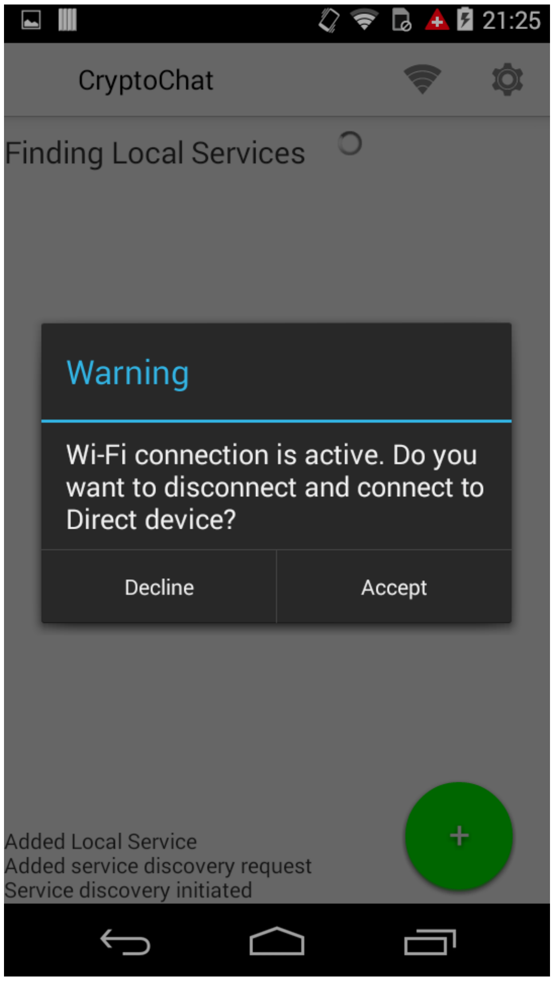
MotoE - cryptochat
80:6c:1b:44:5f:2d
Available

Added Local Service
Added service discovery request
Service discovery initiated

*Figure 13: A peer has been found*

Any other available peers will not appear in the List view until a new peer discovery attempt is made (by pressing the WiFi icon in the action bar). Any peers found will be displayed such as the one in the screenshot defines; the name of the device broadcast, the service he/she is using (currently set to be *'cryptochat'* thus only users who have this application can talk to each other), finally we have the locally administered MAC address of that device along with its status.

Once a user decides to start a chat exchange, he/she chooses a device from the List of available peers to chat with. That user will then receive a prompt asking if they would like to have a *WiFi Direct* chat exchange.



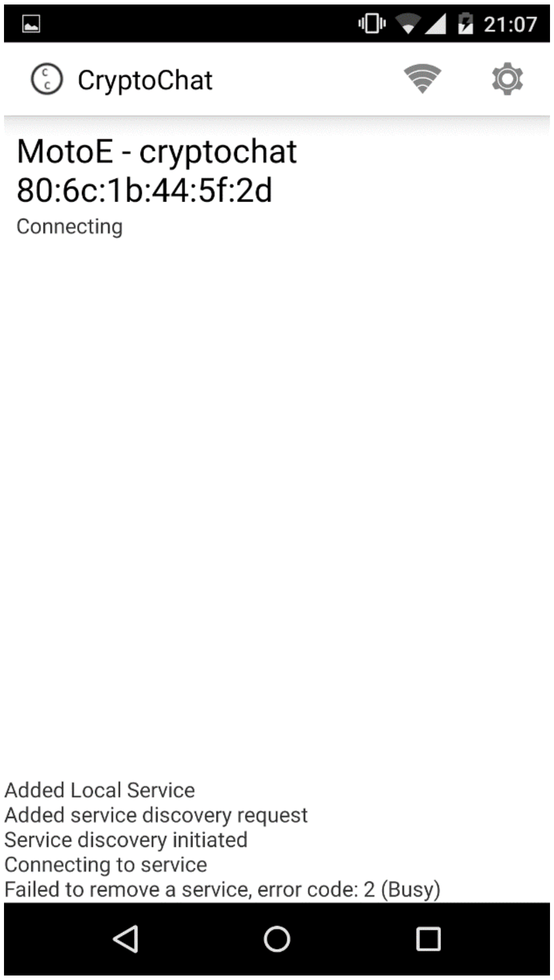
Wi-Fi connection is active. Do you want to disconnect and connect to Direct device?

Added Local Service
Added service discovery request
Service discovery initiated

*Figure 14: Shows a user has found our device and wants to connect*

Is it important to note that this device as can be seen in *Figure 14* is still in the peer search process, this has no effect on this incoming connection attempt. Peers can still connect to this device (as can be seen) and peer discovery will be stopped on the acceptance of this connection attempt or continue on its decline.

The text shown throughout *Figures 12-14* is for debugging purposes and currently provides minimal error assistance, for example; if that text details a failure to add a local service and a failure to initialise search discovery, this is likely to be related to the WiFi being in-active. It also details connection errors that are displayed, *Figure 15* shows an example of one these;

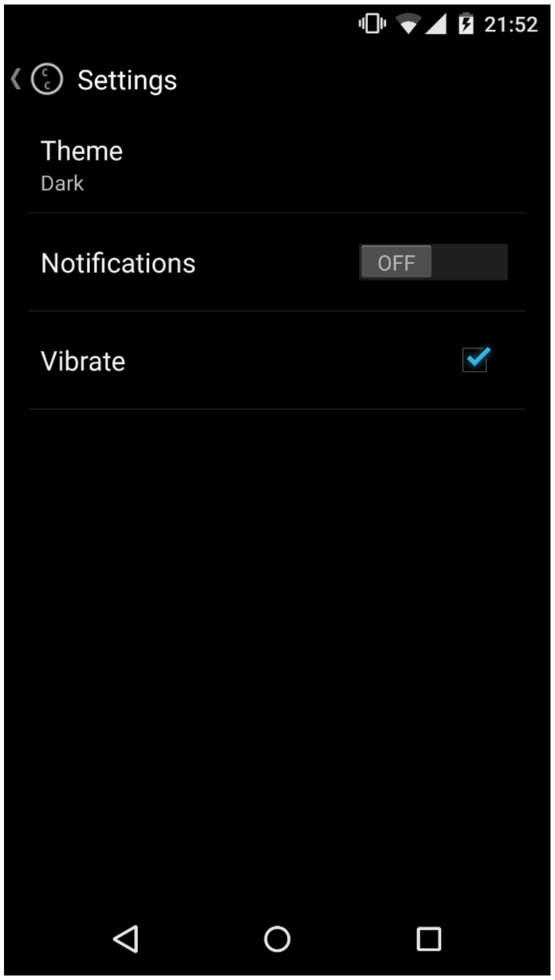


```
Added Local Service
Added service discovery request
Service discovery initiated
Connecting to service
Failed to remove a service, error code: 2 (Busy)
```

*Figure 15: Debugging text, importantly error code 2, a connection attempt has already begun*

The application in its current prototype form does not handle many errors all that well, for example the prior example of WiFi being in-active, should inform the user that he/she needs to activate their WiFi. However it currently does not, implementation time was focused upon the feature set. Error handling and informative user notifications will be detailed later in the future work section of this report.

*Figure 16*


21:52

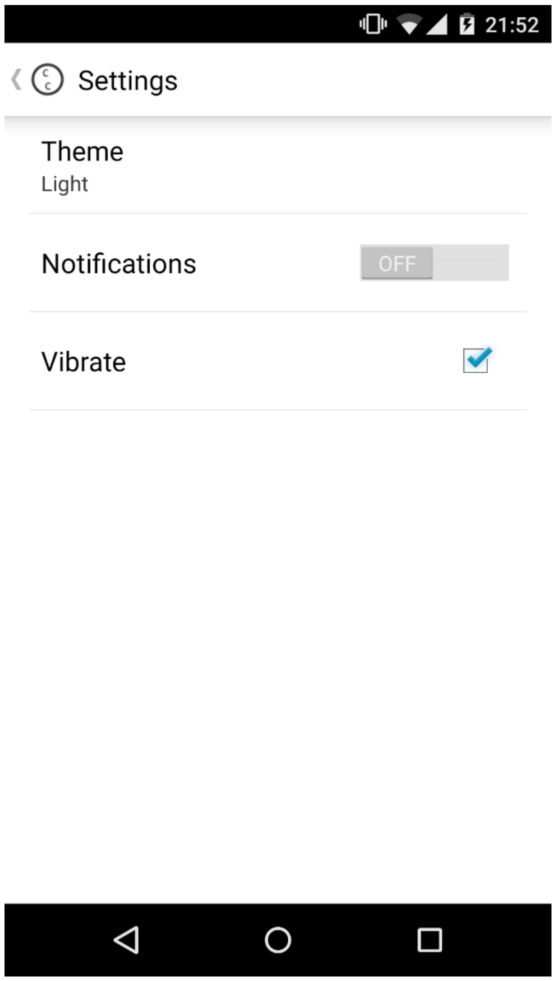
Settings

Theme
Dark

Notifications OFF

Vibrate ✓

*Figure 17*


21:52

Settings

Theme
Light

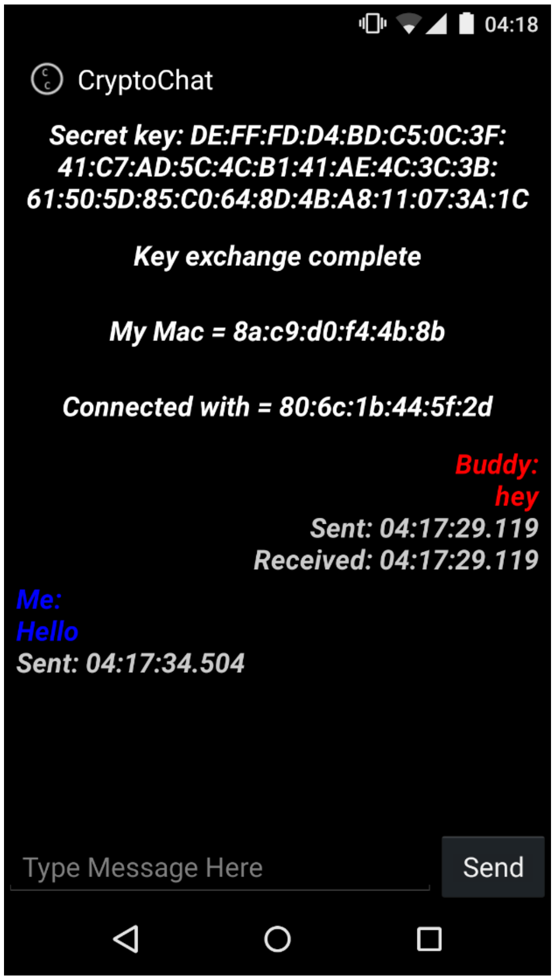
Notifications OFF

Vibrate ✓

*Figure 16 & Figure 17: Settings screens – showing theme support & available options*

The settings screen shown in *Figures 16 & 17* show that this application supports the use of themes, these are currently limited to a simple light & dark variant but can be expanded upon in the future. The themes are applied across all of the screen states, and the text displayed in the screen states is adjusted accordingly to the theme that is currently applied. The notification toggle is in place but currently no actual notification support is implemented. The vibrate option is a simple 'vibrate on message received' option.

*Figure 18*



04:18

CryptoChat

***Secret key: DE:FF:FD:D4:BD:C5:0C:3F:41:C7:AD:5C:4C:B1:41:AE:4C:3C:3B:61:50:5D:85:C0:64:8D:4B:A8:11:07:3A:1C***

***Key exchange complete***

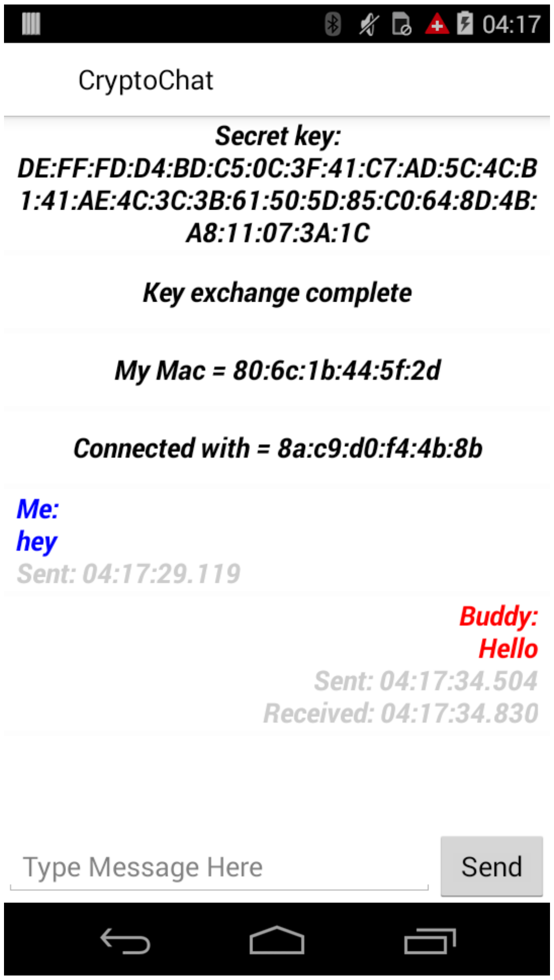
***My Mac = 8a:c9:d0:f4:4b:8b***

***Connected with = 80:6c:1b:44:5f:2d***

***Buddy:***
***hey***
***Sent: 04:17:29.119***
***Received: 04:17:29.119***

***Me:***
***Hello***
***Sent: 04:17:34.504***

*Figure 19*



04:17

CryptoChat

***Secret key: DE:FF:FD:D4:BD:C5:0C:3F:41:C7:AD:5C:4C:B1:41:AE:4C:3C:3B:61:50:5D:85:C0:64:8D:4B:A8:11:07:3A:1C***

***Key exchange complete***

***My Mac = 80:6c:1b:44:5f:2d***

***Connected with = 8a:c9:d0:f4:4b:8b***

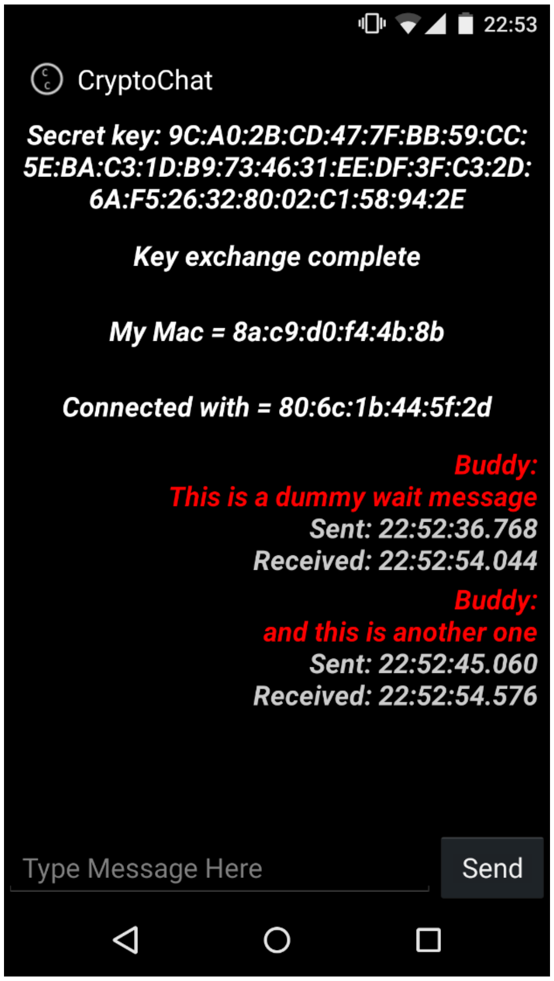
***Me:***
***hey***
***Sent: 04:17:29.119***

***Buddy:***
***Hello***
***Sent: 04:17:34.504***
***Received: 04:17:34.830***

*Figures 18 & 19: Chat screens – Key exchange completed and messages securely exchanged*

The chat screens shown above detail the respective views of the devices that are commencing the secure message exchanges. The first 4 entries in the List view (the white text in the dark theme *Figure 18,* and the black text in the light theme *Figure 19)* are again debug text to show the successful exchange of information has taken place and that both peers/parties have calculated the correct (same) *secret key.* The red text displayed is a message that has been received by that device, and a blue message is one that has been sent by that device. The meta-data (sent & received) are both styled in a light grey colour to distinguish it from the actual message exchanges. Note that the meta-data timings are now entirely synced specifically the Moto-E test device (*Figure 19),* due to its lack of sim card the device became out of sync by a fraction of milliseconds after an extended period of time turned on.

*Figure 20*


22:53

CryptoChat

Secret key: 9C:A0:2B:CD:47:7F:BB:59:CC:5E:BA:C3:1D:B9:73:46:31:EE:DF:3F:C3:2D:6A:F5:26:32:80:02:C1:58:94:2E

Key exchange complete

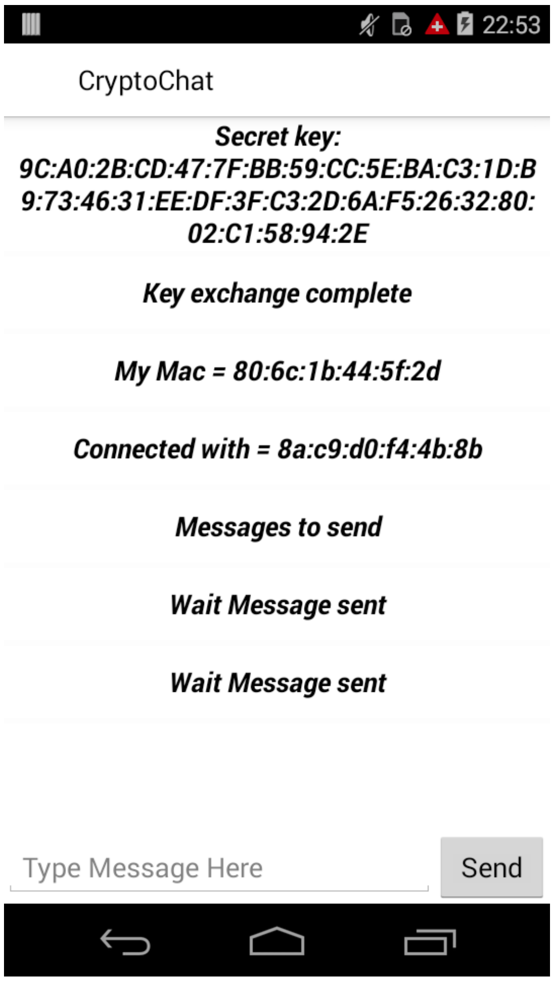
My Mac = 8a:c9:d0:f4:4b:8b

Connected with = 80:6c:1b:44:5f:2d

Buddy:
This is a dummy wait message
Sent: 22:52:36.768
Received: 22:52:54.044

Buddy:
and this is another one
Sent: 22:52:45.060
Received: 22:52:54.576

*Figure 21*


22:53

CryptoChat

Secret key:
9C:A0:2B:CD:47:7F:BB:59:CC:5E:BA:C3:1D:B9:73:46:31:EE:DF:3F:C3:2D:6A:F5:26:32:80:02:C1:58:94:2E

Key exchange complete

My Mac = 80:6c:1b:44:5f:2d

Connected with = 8a:c9:d0:f4:4b:8b

Messages to send

Wait Message sent

Wait Message sent

*Figures 20 & 21: Chat screen – wait messages have been exchanged*

The above *Figures 20 & 21* detail the exchange of *wait messages*, here you see *Figure 21* was the device that had held the *wait messages* and was the device sending them. *Figure 20* shows the device receiving those *wait messages*, remember that the *wait messages* are only sent after key exchange has taken place and a secure means of communication with the other device has been created. After the *wait messages* are exchanged, normal secure message exchanges can then re-commence between the devices. Notice the *secret key* is completely different from the previous screenshots, demonstrating the strength of the DH approach.

## *6.1 Quantifiable Results (CPU, Battery, Network)*

The quantifiable results will show the results of data gathering and monitoring that has been undertaking whilst the application has been running on the Android test device. The test device used during the tests was my personal device, the Nexus 5. It must be stated that during testing my sim-card for the device was removed and most background processes were closed but some do remain and thus populate the results with some unwanted data but as CryptoChat was always the forefront application running during these tests it does not obscure the results.

The CryptoChat APK installed on the Nexus 5 device takes up 2.44mb of space, it has a cache of 0.3mb (saved settings options), and this will grow when a user creates a *wait message file* for a limited period of time at-least. To compare a few similar messaging applications, WhatsApp on the device takes up 33.50mb and (Facebook) Messenger takes up 22.37mb.

The CPU usage of the application is shown in *Figures 22 & 23* below, the two pie charts show the load of the CPU when searching for peers and for when inside the chat screen state;

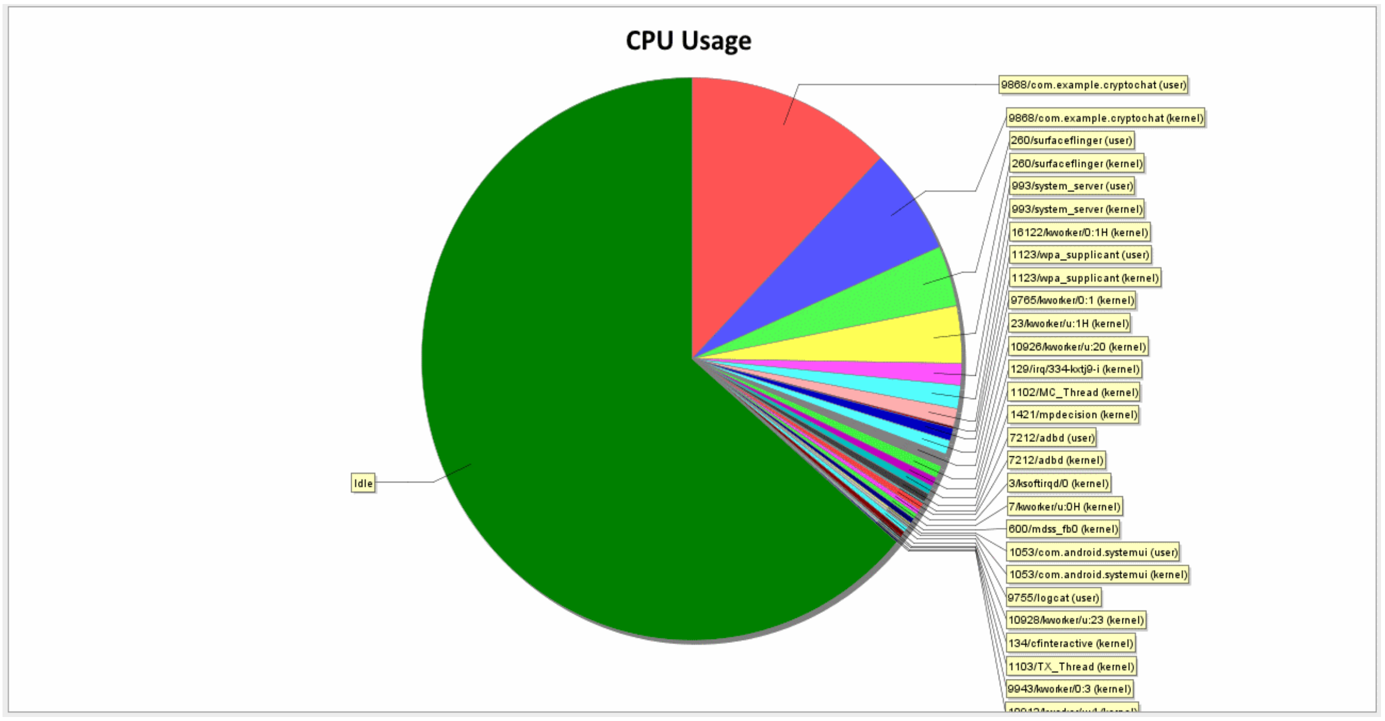


*Figure 22: CPU usage during peer discovery*

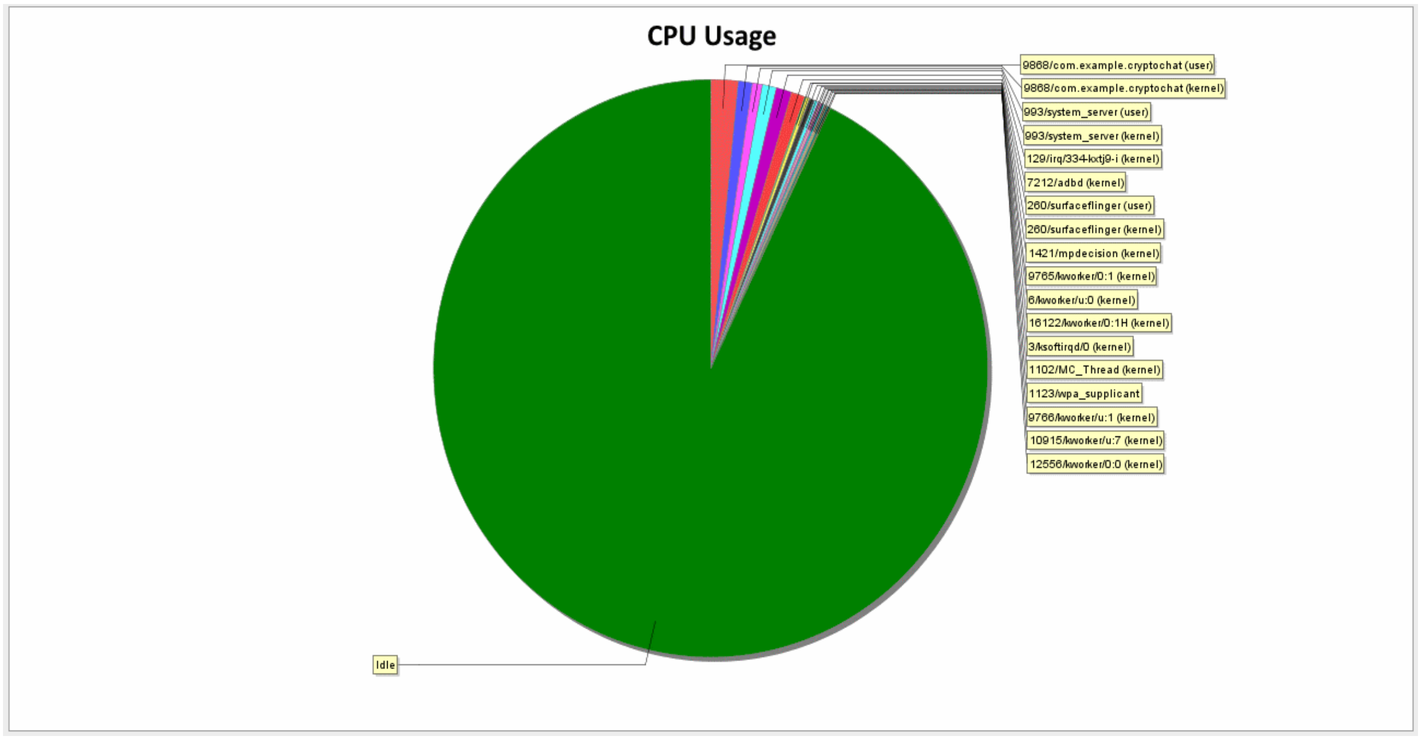


*Figure 23: CPU usage during chat screen state*

Both the pie charts show the minimal use of resources that the application demands the red/pink colour details the user demands of the application and the blue/purple colour detail the kernel demands of the application. The figures for the CPU usage during peer discovery ranged from 15-10% and the CPU usage for chat exchanges ranged from 7-4% during testing. As both of the pie charts show, the CPU is spending more resources sitting idle than it is providing to the application. This is an extremely good sign of the efficiency of the application.

Similarly for the battery consumption, the application is again extremely efficient in its use of battery power. *Appendix A* shows a timeline of battery events that were report to the Android operating system, the timeline was generated during a simple test chat exchange that took place for 20 minutes and was produced using the Google provided tool *Battery Historian [20],* it logs the events of battery consumption from all areas of the device. As you can see in the timeline, screen and WiFi are the two biggest drains (to be expected) of the device's battery. The application does not make any form of significant battery drain except for its use of WiFi, the WiFi power consumption can be specifically see in the sections *WiFi full lock* & *WiFi scan* (it is important to state that the efficiency of the WiFi is very dependent on the calibre of the hardware and by extension the value of the device, areas that are out of my control).

Finally the network efficiency and specifically the throughput of bytes sent per minute was data that was not feasible to obtain since data is only sent when messages are sent, therefore results would be entirely based upon the velocity of chat exchanges that would occur between devices and would obviously range wildly. Some interesting facts where discovered during my attempts to collect data however. The key exchange process was always 472 bytes received by the *Client* and he/she would transmit 525 bytes to the *Group Owner*. The *Group Owner* in comparison as expected transmits 472 bytes and received 525 bytes. A full one sentence message exchanged between devices consisted of 325 bytes and a message containing only one word will consist of 117 bytes. To place this into perspective, the maximum size of a single SMS message is 140 bytes therefore you see we have a quite large overhead applied to messages due to the encryption techniques that are applied. The use of WiFi offsets this overhead however due to the higher data transmission speeds that are possible, making the actually sending and receiving of the messages near instant and with the meta-data you can clearly see throughout the screenshots, the time-delay that is present between devices occurs in the hundreds of milliseconds therefore making the end-to-end delay period barely noticeable to each end user communicating.

Ultimately with all the data and results that can be seen, I feel that the application meets its aim of being very light and thoughtful in its use of system resources especially in the areas of CPU and battery usage. The minimal aesthetic approach that this application employs is another factor that contributes to its efficiency however this is offset by the limited feature set that is currently available.

# *7. Future Work*

The application contains various areas that can be improved upon alongside the implementation of the missing features proposed by this project namely the *group messaging* and *picture messaging* features for the application.

To begin, *group messaging* would be a major feature to implement and have successfully working for the usability & appeal of this application, the ability to message more than one person sounds straight-forward in principle but in reality when using *WiFi Direct* it brings up various problems that would need to be overcome. Firstly since the architecture of a connection requires a *Group Owner* and *Clients,* we are highly dependent on the *Group Owner* since he/she handles most of the work in a chat exchange. This is escalated in a group chat exchange scenario. The *Group Owner* would be the only device to know the information of *Clients,* and therefore it would have to act as a relay node to send messages from a *Client* to the remaining *Clients* in the group chat. If the *Group Owner* where to disconnect, then we would have to re-establish a connection between the remaining devices that were connected, this would be troublesome since the *Group Owner* as stated above would be initially the only device to know which devices are connected to it. A potential solution to this would be to send a copy of all the connected devices to each *Client* connected to the *Group Owner* (similarly to what is being done with MAC addresses & *wait messages* currently). We would also have the problem of group key exchange, which is again an extra burden of information that would have to be exchanged between all the devices before we have even started to begin to communicate. The *Spray and Wait* network policy would also provide another large obstacle, in its current form it would not be applicable and would require some rather extensive modifications. Specifically the *Group Owner* instead of relaying messages would first have to check for an active connection with a *Client* then if that is not available, store the message in a *wait file* to be later 'sprayed' out to *Clients.* However, with multiple disconnects the complexity of the protocol will grow, if for example two *Clients* left a group chat and then came back into range, should the device holding the *wait files* try a group chat exchange to send the *wait messages* to each device or should it connect (as it does now) to only one device? Finally, there is a limitation on the number of devices that can connect which is only stated as, "The number of devices in a Wi-Fi Direct-certified group network is expected to be smaller than the number supported by traditional standalone access points intended for consumer use" *[21],* therefore this will have an effect on limiting the appeal of this application.

It must be stated however that the problems discussed are achievable challenges to overcome, *[5]* shows that group chat is defiantly a feature that could be implemented. The complexity attached with key exchange, *Group Owner* relaying and the *Spray and Wait* network routing protocol however add a layer of problems that will have to be clearly thought through before attempting its implementation in this application.

*Picture messaging* is another feature that would again add more appeal to this application, similarly to *group messaging* it provides some more challenges that would need to be overcome. Sending a picture would require the need for that image to be sent across devices in manageable chunks since any modern phone now produces pictures of 5MP (mega-pixel) in quality which will produce a file of 200+KB in size (depending on the image compression techniques used, most commonly JPEG). Therefore we must ensure the integrity of the data that is being sent across devices so that it can be assembled back together in the correct manner. Then we also have the challenge whether or not to add picture messages to the *Spray and Wait* network routing protocol, this would another aspect any future work would have to consider.

Finally the application in its current state does not handle error and warnings that occur all that well, in-fact most of the information is either only logged during the debugging phase with the device attached to a computer or via the debug text that is shown during connection attempts. This will need to be improved upon in any further attempts in improving the application and its user experience, so that the user is not left unknowing on how to proceed if these errors or warnings occur.

# 8. Conclusion

This project meets most of the aims that it set out to originally accomplish, specifically the application built is minimal, user-friendly and provides basic functionality for having secure communication exchanges between individuals. The *Spray and Wait* network routing protocol implemented alongside the basic functionality gives users increased freedom when communicating. The application provides this functionality but will need further development on features such as *group messaging* and *picture messaging* for it to become a more viable product.

Time challenges and the higher than expected learning curve when developing this application using the Android SDK made those features out of reach of my abilities, however the features that have been implemented answer the fundamental question asked during this report on how we can communicate securely with other people without using public infrastructure or cellular networks. The choice of using *WiFi Direct* like any technology has its pro and cons and furthermore, development of this application will have to address them as they become apparent, however I feel that using *WiFi Direct* was the best option for this application since it provides a universal standard to work upon. I also feel that the choice of using Android even though it initial provided some large obstacles and challenges eventually paid off in its masses of documentation and available resources that I could access online.

The application built is most defiantly a prototype but it is a functional one that I feel can be a solid starting point, to which can be expanded upon to implement the missing features. With further development and polish, the application could be launched on public *'app stores'* and provide a solid service to its potential users.

## *Reflections on Learning*

This project I feel has developed my skills as a programmer and more importantly I feel that this project has given me some valuable experience in following the software development life cycle and the challenges that come with it.

In hindsight I defiantly under-estimated the challenges that Android development would provide, the project had its fair-share of 'speed-bumps' along the way with some days where I was becoming very frustrating with *x* feature that just would not work, my patience and problem solving skills became important areas in which I was tested mentally and probably more so than I ever have in my degree scheme. However my patience and problem solving skills were not the most important factor during the development of this application. Rather it was my research skills and knowledge that became crucial to the successful implementation of the features this application provides, without the research that I was conducting online to find resources that would aid my problem solving, I would have likely struggled to find the solutions that I did (or even if I did these solutions would have taken much longer to find). My research skills became invaluable towards the latter stages of this project specifically concerning areas of cryptography which was a fun but challenging area that was very much mostly unknown to me.

The choice of Android and Java in this regard was a 'blessing in disguise' since the available documentation, tutorials, wikis and other resources that were used provided key insights in how to tackle certain problems, such as key exchange and theme support. I feel now that if I were to continue development of this application using Android and Java I would be starting off in a much stronger place than I was when I started this project.

In regards to my project plan and the time-frame that I set out in my initial plan, not much of this was actually followed during the development of this application. I was probably over ambitious in my goals especially with the limited time-frame available and this is something that I must be sure to carry forward with me in the future, to recognise what can be accomplished in the time provided. I also feel that my goals were not clearly stated and planned all that well during my initial plan, in the future I should take more time to think about what precisely I should be accomplishing on a week by week basis to make my project plan more of an effective tool and help guide the development of future projects.

Ultimately this project has tested and expanded upon my skillset that I have acquired through my own personal time and university studies, I have more experience now with Java & the Android SDK and feel these are valuable assets for the future of my career. Overall I have thoroughly enjoyed my time during the development of this project and the challenge that it provided and look forward to working on future projects like this.

## Glossary

| *Name* | *Description* |
|---|---|
| *DH* | Refers to the Diffie-Hellman key exchange. |
| *Secret Key* | Refers to the output key produced by the Diffie-Hellman key exchange. |
| *Wait Message File* | Refers to the file that contains the *'wait message(s)'* that will be delivered at some point to the intended recipient. |
| *Wait Message(s)* | Refers to messages that have been sent to an out of range or not available recipient that will be stored in the *'wait message file'*. |
| *AES* | Refers to the Advanced Encryption Standard. |
| *IV* | Refers to the initialisation vector used during encryption & decryption. |
| *APK* | Refers to an Android Application Package. |
| *MAC address* | Refers to the media access control (MAC) address of a device. |
| *Spray Phase* | Refers to the Spray & Wait network routing protocol; the spray phase occurs when a party/peer disconnects and the other party/peer is generating *'wait message(s)'.* |
| *Wait Phase* | Refers to the Spray & Wait network routing protocol; the wait phase occurs when a party/peer has been discovered that has a *'wait message file'* associated with it and thus the messages contained within that file are sent during this phase. |
| *Group Owner* | Refers to the device that will act as the 'server' for the connection between the devices message exchanges. |
| *Client* | Refers to the device that will act as a *'Client'* which connects to the *Group Owners* 'device during the message exchanges. |
| *JPEG* | Refers to the Joint Photographic Experts Group standard for lossy compression of digital images. |

## References

[1] Android Apps on Google Play. 2015. Available at: https://play.google.com/store/apps/top?hl=en_GB. Accessed April 6, 2015.

[2] Ofcom - The Communications Market Report: United Kingdom. 2015. Available at: http://stakeholders.ofcom.org.uk/market-data-research/market-data/communications-market-reports/cmr14/uk/. Accessed April 6, 2015.

[3] Frommer D. How Facebook Controls the Future of Messaging. The Atlantic. 2015. Available at: http://www.theatlantic.com/technology/archive/2015/03/how-facebook-messaging/389171/. Accessed April 6, 2015.

[4] WiFi Social. 2015. Available at: https://play.google.com/store/apps/details?id=com.ajaraj.wifisocial&hl=en. Accessed April 12, 2015. Accessed April 12, 2015.

[5] WiFi Direct Group Chat. 2015. Available at: https://play.google.com/store/apps/details?id=esnetlab.apps.android.wifidirect.discovery&hl=en_GB. Accessed April 12, 2015.

[6] Android.googlesource.com. Samples/WiFiDirectServiceDiscovery - platform/development - Git at Google. 2015. Available at: https://android.googlesource.com/platform/development/+/512cc91be2cce9566807bd7248da448c0a91e2ed/samples/WiFiDirectServiceDiscovery/. Accessed April 12, 2015.

[7] Some SecureRandom Thoughts | Android Developers Blog. Android-developers.blogspot.co.uk. 2013. Available at: http://android-developers.blogspot.co.uk/2013/08/some-securerandom-thoughts.html. Accessed April 12, 2015.

[8] Android 4.0 APIs | Android Developers. 2015. Available at: http://developer.android.com/about/versions/android-4.0.html. Accessed April 12, 2015.

[9] Google design guidelines. Introduction - Material design - Google design guidelines. 2015. Available at: http://www.google.com/design/spec/material-design/introduction.html. Accessed April 13, 2015.

*[10] Delay Tolerant Networks. 2015. Available at: https://books.google.co.uk/books?id=2ERN5lgs3AwC&pg=PT56&lpg=PT56&dq=spray+%26+wait+vanilla&source=bl&ots=f8DJLSGQ76&sig=PdEwUKGiZyku4F7mw4r6VsXTedE&hl=en&sa=X&ei=r5stVdagM4TmapelgdgK&ved=0CC4Q6AEwAw#v=onepage&q=spray%20%26%20wait%20vanilla&f=false. Accessed April 14, 2015.*

*[11] Creating P2P Connections with Wi-Fi | Android Developers. 2015. Available at: http://developer.android.com/training/connect-devices-wirelessly/wifi-direct.html. Accessed April 16, 2015.*

*[12] Java Cryptography Architecture (JCA) Reference Guide. 2015. Available at: http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html. Accessed April 16, 2015.*

*[13] Java Cryptography Architecture (JCA) Reference Guide (DH2EX). 2015. Available at: http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#DH2Ex. Accessed April 16, 2015.*

*[14] Fragments | Android Developers. 2015. Available at: http://developer.android.com/guide/components/fragments.html. Accessed April 16, 2015.*

*[15] Diffie-Hellman Key Agreement Method. 2015. Available at: https://www.ietf.org/rfc/rfc2631.txt. Accessed April 16, 2015.*

*[16] PKCS #7: Cryptographic Message Syntax. 2015. Available at: http://tools.ietf.org/html/rfc2315. Accessed April 16, 2015.*

*[17] Handler | Android Developers. 2015. Available at: http://developer.android.com/reference/android/os/Handler.html. Accessed April 17, 2015.*

*[18] Standard Group MAC Addresses: A Tutorial Guide. 2015. Available at: http://standards.ieee.org/develop/regauth/tut/macgrp.pdf. Accessed April 18, 2015.*

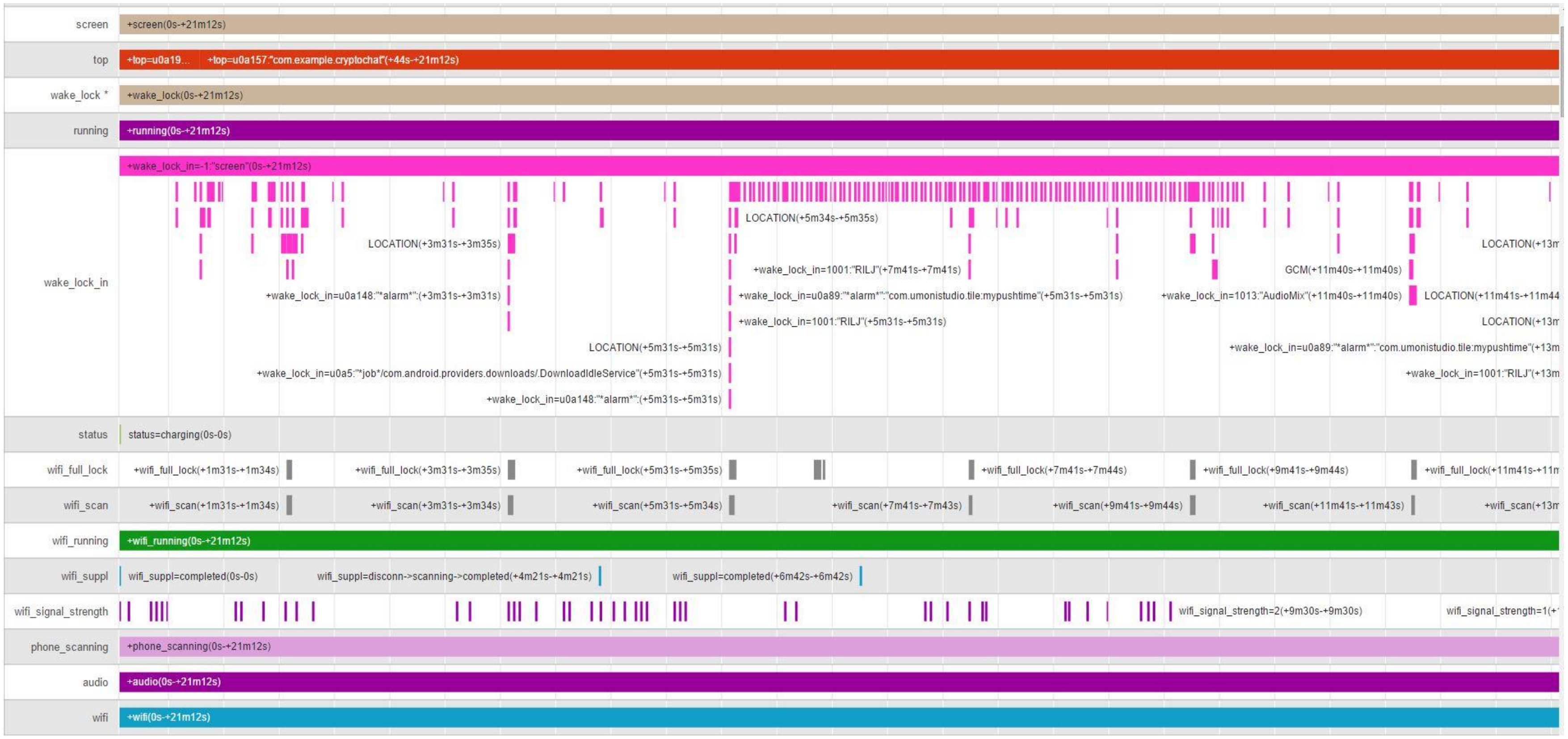
*[19] CWE - CWE-329: Not Using a Random IV with CBC Mode (2.8). 2015. Available at: http://cwe.mitre.org/data/definitions/329.html. Accessed April 18, 2015.*

*[20] GitHub - Google/battery-historian. 2015. Available at: https://github.com/google/battery-historian. Accessed April 19, 2015*

*[21] How many devices can connect? | Wi-Fi Alliance. 2015. Available at: http://www.wi-fi.org/knowledge-center/faq/how-many-devices-can-connect. Accessed April 24, 2015.*

## *Appendix A – Battery Timeline*

Screen grab of the battery information produced by *Battery Historian [20]*, full information can be found in the In the folder 'extras' (uploaded along with the source code) in the file – *battery_stats.html.*


screen
+screen(0s-+21m12s)
top
+top=u0a19...
+top=u0a157:"com.example.cryptochat"(+44s-+21m12s)
wake_lock *
+wake_lock(0s-+21m12s)
running
+running(0s-+21m12s)
wake_lock_in
+wake_lock_in=-1:"screen"(0s-+21m12s)
LOCATION(+5m34s-+5m35s)
LOCATION(+3m31s-+3m35s)
LOCATION(+13m
+wake_lock_in=1001:"RILJ"(+7m41s-+7m41s)
GCM(+11m40s-+11m40s)
+wake_lock_in=u0a148:"*alarm*":(+3m31s-+3m31s)
+wake_lock_in=u0a89:"*alarm*":"com.umonistudio.tile:mypushtime"(+5m31s-+5m31s)
+wake_lock_in=1013:"AudioMix"(+11m40s-+11m40s)
LOCATION(+11m41s-+11m44
+wake_lock_in=1001:"RILJ"(+5m31s-+5m31s)
LOCATION(+13m
LOCATION(+5m31s-+5m31s)
+wake_lock_in=u0a89:"*alarm*":"com.umonistudio.tile:mypushtime"(+13m
+wake_lock_in=u0a5:"*job*/com.android.providers.downloads/.DownloadIdleService"(+5m31s-+5m31s)
+wake_lock_in=1001:"RILJ"(+13m
+wake_lock_in=u0a148:"*alarm*":(+5m31s-+5m31s)
status
status=charging(0s-0s)
wifi_full_lock
+wifi_full_lock(+1m31s-+1m34s)
+wifi_full_lock(+3m31s-+3m35s)
+wifi_full_lock(+5m31s-+5m35s)
+wifi_full_lock(+7m41s-+7m44s)
+wifi_full_lock(+9m41s-+9m44s)
+wifi_full_lock(+11m41s-+11m
wifi_scan
+wifi_scan(+1m31s-+1m34s)
+wifi_scan(+3m31s-+3m34s)
+wifi_scan(+5m31s-+5m34s)
+wifi_scan(+7m41s-+7m43s)
+wifi_scan(+9m41s-+9m44s)
+wifi_scan(+11m41s-+11m43s)
+wifi_scan(+13m
wifi_running
+wifi_running(0s-+21m12s)
wifi_suppl
wifi_suppl=completed(0s-0s)
wifi_suppl=disconn->scanning->completed(+4m21s-+4m21s)
wifi_suppl=completed(+6m42s-+6m42s)
wifi_signal_strength
wifi_signal_strength=2(+9m30s-+9m30s)
wifi_signal_strength=1(+
phone_scanning
+phone_scanning(0s-+21m12s)
audio
+audio(0s-+21m12s)
wifi
+wifi(0s-+21m12s)

## *Appendix B – Program Listing*

CryptoChat

- ➔ .settings
- ➔ assets
- ➔ bin

.settings contains Eclipse settings
assets is an empty folder
bin contains compiler output

- ➔ extras
    - ○ *battery_stats.html* – contains the full data of battery information displayed in Appendix A
- ➔ gen
    - ○ com
        - ▪ example
            - • cryptochat
                - ○ *BuildConfig.java* – auto generated file
                - ○ *R.java* – auto generated file that states the android resources that have been used such as colours, layouts, strings etc.
- ➔ libs
    - ○ *android-support-v4* – *"*support android.app classes to assist with development of applications for android API level 4 or later. The main features here are backwards-compatible versions of FragmentManager and LoaderManager*"*
- ➔ res
    - ○ drawable-hpdi
    - ○ drawable-ldpi
    - ○ drawable-mdpi
    - ○ drawable-xhdpi
    - ○ drawable-xxhdpi

The following folders contain drawable images, namely logos and icons that are used in this application. The only difference between the logos and icons used and present in these folders is the size of the images.

- layout
    - *activity_main.xml* – defines the layout for MainActivity.java
    - *activity_settings.xml* – defines the layout for SettingsActivity.java
    - *devices_list.xml* – defines the layout for devices found during peer discovery
    - *fragment_chat.xml* – defines the layout of the chat screen state
    - *settings.xml* – defines the preferences of SettingsActivity.java
- menu
    - *main_activity_actions.xml* – defines the actions a user can undertake during the start screen
- values
    - *arrays.xml* – contains arrays that are used for the theme choice combo box
    - *colors.xml* – contains hexadecimal values of colours that are used in this application
    - *strings.xml* – contains strings that are used in this application
    - *styles.xml* – contains the styles (text and theme ) that are used in this application
- values-v11
- values-v14

➔ src

- com
    - example
        - cryptochat
            - *ChatManager.java** – Handles chat exchanges between devices
            - *Circle.java* – Simple class used to create a circle object to draw to screen
            - *ClientSocketHandler.java** – Handles client socket creation & connections
            - *Crypto.java* – Class that defines the encryption/decryption methods used in this application
            - *DHKeyAgreement.java* – Class that defines the DH key exchange/agreement procedures
            - *GroupOwnerSocketHandler.java** – Handles the group owner socket creation & connections
            - *MainActivity.java* – The main file, defines most of the application functionality
            - *PRNGFixes.java – [7]*

Note: * indicates files that were provided by *[6]*, nearly of these files have been modified and added to in some form.

- *SettingsActivity.java* – Settings, defines the activity and what fragment it should use
- *SettingsFragment.java* – Defines the content and UI of the *SettingsActivity*
- *Utils.java* – Static helper class that provides access to widely used methods
- *WiFiChatFragment.java** – Defines the chat screen and its look
- *WiFiDirectBroadcastReceiver.java** – Defines the broadcast receiver used to react to events that occur during the running of the application
- *WiFiDirectServicesListFragment.java** – Defines the view for discovered peers found in during peer discovery
- *WiFiP2pService.java** – Defines a structure to hold service information

➔ .classpath

➔ .project

➔ *AndroidMainfest.xml* – Defines properties of the application such as what activity to run/show first, what logo to use, what hardware it has the right to access etc.

➔ proguard-project.txt

➔ project.properties

## *Appendix C – Extra Resources*

The following extra resources (.zip files) have been uploaded, these contain the following:

- *Designs* – Contains all of the full-sized design images shown in *Figures 1-5*.
- *Graphs* – Contains the full-sized pie charts shown in *Figures 22 and 23* and the battery historian image, *Appendix A*.
- *Implementation Diagrams* – Contains the full-sized diagrams shown in *Figures 7-11*.
- *Result Screenshots* – Contains the full-sized screenshots shown in *Figures 12-21.*