The Report Committee for Mason Tyler Schoolfield

Certifies that this is the approved version of the following report:

Message Transfer Framework For Mobile Devices

Using Bluetooth Low Energy

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor: ___________________________________
Christine Julien

____
William Bard

Message Transfer Framework For Mobile Devices
Using Bluetooth Low Energy

By

Mason Tyler Schoolfield, B.B.A

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

In Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2015

Dacia – I’m glad we got to share all this.

Jonas – Keep learning, if you don’t push you’ll never grow.

Acknowledgements

I would like to thank all of my instructors in the Electrical Engineering Department at the University of Texas at Austin, who taught me complex concepts and instilled in me an appreciation of intellectual rigor. My supervisor, Christine Julien, exposed me to the breadth and capabilities of mobile computing. My reader, William Bard, taught me the importance of security and gave me insight into the implementation of encryption.

Message Transfer Framework For Mobile Devices Using Bluetooth Low Energy

Mason Tyler Schoolfield, M.S.E.

The University of Texas at Austin, 2015

Supervisor: Christine Julien

Despite the increasing availability of mobile devices offering handheld peer to peer communication capabilities, interoperability between heterogeneous mobile device platforms is hampered by the security requirements of their underlying operating systems. Bluetooth Low Energy (BLE) was introduced so that mobile devices could read from sensors with fewer security requirements, presenting an opportunity to allow disparate devices to connect and transfer data.

This paper presents a message transfer framework to facilitate arbitrary data transfer using the GATT mechanisms as provided by BLE in the Bluetooth 4.0 specification. The provided implementation library and applications for Android along with a proof of concept application for iOS 8 sustain reliable transfer speeds of 1 KB/s, allowing for a 100KB payload (a small picture, for example) to be sent wirelessly between an Android and iOS device in just over a minute.

Table of Contents

List of Tables

List of Figures

Chapter 1 - Introduction

1.1 Motivation

With almost ubiquitous access to the internet, users of smart mobile devices have come to rely on infrastructure for communication. However there are times when infrastructure cannot serve the requirements of the user, such as disaster scenarios [1], an unreliable signal (either in the developing world or a large building), or a saturated wireless spectrum such as at a large music festival [2]. Users may also want to avoid infrastructure to preserve their privacy from malicious actors or government overreach [3], or continue to communicate when mobile infrastructure access has been blocked [4].

As mobile devices gain more functionality and attention in the lower-end and developing world markets, more consumers will have access to communication features that were previously reserved for higher end hardware. For example, Google recently announced their Android One project to sell affordable smartphones in Asia [5]. Android One compliant phones implement a Google-approved reference design [6] which includes Bluetooth 4.0 and Wi-Fi Direct compliance. In the United States, Motorola sells the Moto E (used as the primary development device for this paper) which ships with Android 5.0. As mobile devices with improved hardware and software approach commodity, so too does the mobile device ecosystem approach a more functional base level of inter-device communication capabilities.

Android provides built-in applications allowing two devices to transfer files using NFC (the “Beam” application) or Bluetooth (once these devices have paired). Android also provides programmatic access for peer to peer connectivity via its APIs for NFC, Wi-Fi Direct and Bluetooth [7]. iOS provides AirDrop for file transfer to other iOS devices. iOS also provides programmatic access to similar functionality via its MultiPeer Connectivity API, an abstraction interface for peer to peer connectivity that uses Wi-Fi or Bluetooth to communicate with “nearby iOS devices” [8]. With the peer to peer support provided by both of these major mobile platforms, transmitting a small amount of information from one device directly to another should be trivial.

1.2 Problem

The baked-in applications on iOS and Android, while they allow simple transfer between homogeneous platforms, do not allow an iOS device to send data directly to an Android device (or vice-versa). The programmatic APIs offer no relief; even though Android provides lower level access to peer to peer communication protocols, a developer still cannot write an application to communicate directly with an iOS device due to the previously noted restriction that MultiPeer Connectivity API enforces to only connect with iOS devices.

Both platforms offer Bluetooth support, a technology for exchanging data over short distances. Primarily used as a way to communicate with peripheral devices such as headsets and keyboards, Bluetooth also offers the capability for devices to transmit data between each other. While Android devices can use Bluetooth to transfer files and arbitrary data, iOS does not support the Bluetooth profile necessary to transfer data [9]. Thus as a method of data transfer, classic Bluetooth is relegated to the realm of being proprietary per OS, and no more interoperable than a technology like Android's WiFi p2p or iOS's MultiPeer Connectivity API. Since Android has 80% of the worldwide market [10] one could almost assume that writing an application specifically for Android would provide sufficient market penetration – however in the US and Europe market share is much more evenly split between iOS and Android devices [11].

1.3 Vision

Bluetooth Low Energy (commonly abbreviated BLE) was added to the Bluetooth 4.0 specification and was designed for a handheld device to gather data from a sensor. Chips were designed to be cheap to manufacture, low power so that sensors can remain in place for long periods of time, and use an open and simple protocol to make it easy for devices to communicate. Authentication and encryption, while offered, are not required for connectivity.

Starting with the iPhone 4S, iOS 5 was able to act as a handheld BLE scanner as well as impersonate a remote sensor providing data. Starting with Android 4.3, Android devices were able to act as a handheld BLE scanner, and with Android 5.0, new Android devices gained the ability to act as a remote sensor providing data. With mobile devices gaining the capability to act as these roles (that of scanner and that of sensor), any mobile

OS adhering to the Bluetooth 4.0 standard gained the ability to transfer information to another compliant OS using BLE.

1.4 Hurdles

Because BLE was designed to pull readings from sensors, it was optimized to transmit small amounts of data at close proximity, with the bulk of the data originating from the sensor and not the reader. Thus data transfer rates between two devices will be asymmetric, where the device acting as a sensor will be able to transfer outbound data at a higher rate than the device acting as a reader.

While the iOS BLE implementation is well established, the Android implementation is new and approximately only half of Android devices currently support BLE, with only 5% supporting the ability to impersonate a sensor [12].

While classic Bluetooth has long been established as a secure transfer medium, BLE as specified by the Bluetooth 4.0 specification has a weak encryption handshake that is exploitable [13]. As with classic Bluetooth, pairing takes place at the OS level and cannot be handled by an application.

1.5 Scope and Report Organization

This report will describe a framework, SimpBLE, which provides for the transfer of arbitrary data using Bluetooth Low Energy constructs as a transport layer. This report will provide an implementation library compatible with Android versions 4.3 through 5.1, popular use cases a developer might choose to implement, and finally Android implementations of SimpBLE that demonstrate the feasibility of using common consumer grade Android devices for message communication, secure and otherwise. Compatible transport-level mechanisms will be implemented on iOS to demonstrate the feasibility and performance of cross-platform communication over BLE.

Chapter 2 – Bluetooth Low Energy (BLE)

Detailing the lower level implementation details of Bluetooth Low Energy is outside the scope of this paper. However a valid understanding of how devices communicate over BLE will help the reader appreciate the utility of SimpBLE. The following description is thus meant to be informative but not exhaustive.

2.1 High Level Description of Bluetooth Low Energy

For the purposes of this report, the layers of Bluetooth Low Energy from a software developer's perspective are distilled in Figure 2.1 (adapted from [14] and [15]).

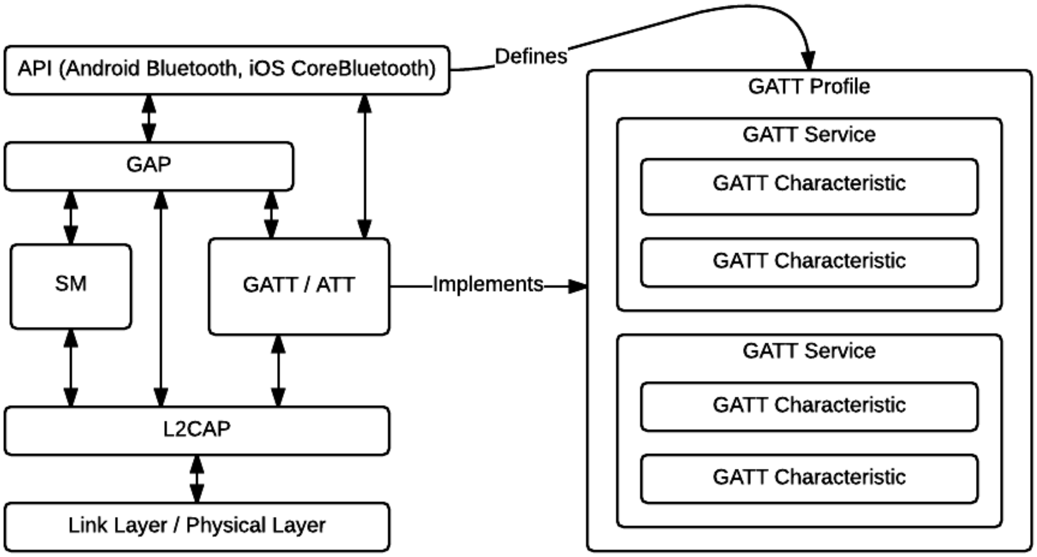


Figure 2.1 – Bluetooth Low Energy Protocol Layers

2.1.1 Link Layer / Physical Layer

As with Classic Bluetooth, BLE operates in the 2.4 GHz ISM band. However BLE divides this spectrum into 40 channels of 2 MHz each (instead of 79 channels of 1 MHz each). Of these 40 channels, 3 are for advertising services and are positioned at 2402, 2426, 2480 MHz to avoid interference with the most commonly used 802.11 frequencies. The 37 remaining channels are assigned for data transmission, using Adaptive Frequency Hopping to rapidly switch channels to avoid interference.

2.1.2 L2CAP

The Logical Link Control and Adaption Layer Protocol (L2CAP) is responsible for multiplexing the data it receives from the higher layer protocols and communicating with the Link Layer below it.

2.1.3 GAP

The Generic Access Profile (GAP) is responsible for the logical connection functionality for the BLE stack, governing how devices connect with each other. GAP defines four roles of Central, Peripheral, Observer, and Broadcaster. Broadcaster and Observer roles do not allow connections so this paper will focus on the connection and data transfer capabilities of the Central and Peripheral roles.

2.1.4 SM

The Security Manager (SM) provides crytographic methods for encryption and authentication that can be used by the other layers for secure communication through the Security Manager Protocol (SMP). The SMP encryption layer in BLE uses AES-128 with a CCM encryption engine, which for session encryption is secure as there are no practical attacks. However the key exchange was designed by the Bluetooth SIG and isn't a standard well tested protocol as with Diffie-Hellmann. It has been shown to be flawed as the Temporary Key (TK) can be brute forced in less than one second on an Intel i7 [13]. Once the TK is determined the subsequent keys are trivial to find.

2.1.5 ATT and GATT

The Generic Attribute Profile (GATT) is the framework that BLE uses to store and transfer data. A device in the GAP role of Peripheral traditionally hosts a GATT Server, which maintains a database of information ready for consumption. Traditionally a GATT Client in the Central role interfaces with the GATT Server to pull data formatted as Attributes (as provided for by the Bluetooth 4.0 specification). These role combinations are common enough that the terms Central and Client are often used interchangeably, as are the terms Peripheral and Server.

To communicate with the GATT Server and consume the data provided by these Attributes, the Bluetooth 4.0 specification requires the Attribute Protocol (ATT) to be implemented for Bluetooth Low Energy. The GATT Client and Server communicate using the Attribute Protocol, which (as its name suggests) allows for operations on these Attributes such that the GATT Client can read data from and write data to the GATT Server.

2.2 The Bluetooth Profile Abstraction

To aid in standardization the Bluetooth SIG provides Profiles - definitions for common applications and behaviors that these applications should use for communication. The Bluetooth SIG highlights the frequently implemented Heart Rate Profile [16] as a common example: a smart phone with a heart rate monitor app scans its range for a smart heart rate monitor. It picks up the heart rate sensor attached to the user’s chest strap and subscribes to be notified of heart rate measurements. The heart rate sensor pushes its data to the subscribed monitoring app. The monitoring app can also pull the location of the sensor on the body as well as modify control parameters on the sensor itself.

Because Profiles such as these are standardized, an application developer can write the module for a communication interface and be assured that it will work with any peripheral whose Bluetooth implementation recognizes and adheres to the same advertised Profile.

For the BLE specification in particular, the Bluetooth SIG has defined a distinct set of Profiles that are considered part of the GATT Profile [17]. These Profiles allow the developer of software for a Peripheral device hosting a GATT Server to know how to make a sensor’s data available, the same way these Profiles allow the developer of software for a Central device hosting a GATT Client to know where to look for data and what contract of data transfer to expect.

2.3 Services and Characteristics

Bluetooth Low Energy Profiles are implemented through GATT Characteristics grouped into GATT Services. In the case of the Heart Rate Profile, the Heart Rate Service exposes three Characteristics – Heart Rate Measurement, Body Sensor Location, and Heart Rate Control Point. This Service and its Characteristics are forms of Attributes, and as such their access is governed by the ATT protocol. As Attributes, they are also identified by 128 bit Universally Unique Identifiers (as defined in ITU-T X.667 and ISO/IEC 9834-8). To facilitate development, Shortcut UUIDs of only 16 bits are available for commonly used Services and Characteristics and are provided by the Bluetooth SIG; programmatically these are extrapolated to full 128 bit UUIDs using a Bluetooth SIG- provided base UUID [18]. Profile specifications reference these UUIDs so that developers know what UUIDs to use when building their applications – a developer can check the Bluetooth SIG to find that the Heart Rate Service UUID is 0x180D and the Heart Rate Measurement UUID is 0x2A37.

Services are valuable constructs in that they group and make visible Characteristics. The handle to data, however, is provided solely through Characteristics that a GATT client can read from or write to, depending on how these characteristics are configured. An application developer will implement a GATT Client to search for a GATT Service and pull its GATT Characteristics. The Client will then use the Attribute Protocol to act on advertised Characteristics by reading a value, writing a value, or asking to be notified by a Peripheral when a value has changed. These operations form the backbone through which devices transfer data using BLE.

2.4 Using Bluetooth Low Energy

The Bluetooth SIG approved GATT based profiles illustrate the kind of services that Bluetooth Low Energy was designed for: Blood Pressure, Health Thermometer, Heart Rate, Running Speed and Cadence, etc. All of these demonstrate the intention that Bluetooth Low Energy was designed for a Central device with a reasonable amount of power to interface with a Peripheral device (providing data) that is meant to be longer lived. BLE was not designed for large amounts of data (the default payload per packet in Android is 20 bytes), but rather to achieve the goals of low power consumption and low latency.

2.4.1 Intended functionality, example

The use case for a Heart Rate Profile application is implemented as follows: a smart phone with a heart rate monitor app assumes the role of Central and scans for devices advertising a service with the UUID 0000180D-0000-1000-8000-00805F9B34FB (the GATT Heart Rate Service). The Central finds the heart rate monitor which is advertising this service as a Peripheral, and subscribes to the Characteristic identified by UUID 00002A37-0000-1000-8000-00805F9B34FB (the GATT Heart Rate Measurement Characteristic) by updating the Heart Rate Measurement Client Characteristic Configuration descriptor. The heart rate sensor updates the value of the subscribed Characteristic on its GATT Server, which pushes the updated value to the subscribed Central application.

2.4.2 Extended, Novel Uses

Consider this application – one friend wants to send a picture from his Android Moto E smartphone to his friend’s iPhone 5. The Wi-Fi router at the coffee shop where the friends are chatting over coffee is out of service and one of the friends has exceeded his mobile phone LTE data allotment for the month. Both friends pull up the same app. The Android user selects the picture and scans for the iPhone, which was placed in receive mode. Both phones connect, and the Android phone sends a segmented byte stream of the picture in question to the iPhone. The app on the iPhone re-assembles the bytes into a picture file and saves it to the phone’s picture gallery.

The popular mobile app FireChat [19] implements the message sending portion of the above functionality using Bluetooth Low Energy as part of its “Nearby” Chatroom functionality. Messages are not sent to a particular device, but are sent to a virtual chatroom such that any device that is part of this chatroom will receive the message.

Other protocols can be used (via the iOS MultiPeer Connectivity API) if the conversing devices are all running iOS.

The Kwikset lock company recently released its Kevo Smart Lock for a physical door [20]. The Kevo app (available on Android and iOS) uses BLE to control the physical Kevo Smart Lock. It's capable of unlocking the door with no user interaction, advertising in BLE Peripheral mode on a low duty cycle such that the physical lock can scan for and find the Kevo app. [see Figure A.1]

There are no Bluetooth SIG approved GATT Profiles to enable the above use cases. These software manufacturers instead use their own ad-hoc and unpublished Profiles, which is a welcome feature of BLE [21]. Given that the use cases for these applications are dissimilar from the SIG approved GATT Profiles, why did these device and software manufacturers choose to use BLE? In the case of FireChat, BLE provides the only mechanism through which an iOS and Android device can transfer data. In the case of Kevo, the low power capabilities of BLE allow the batteries in the door lock to last a year [22], and allow the user's phone to constantly advertise on a very low duty cycle so that the lock can scan for and detect the phone when in proximity. Both of these applications illustrate currently active applications for data transfer over BLE.

2.5 Suitability of Bluetooth Low Energy for message transport

From a transport perspective, classic Bluetooth BR/EDR is much better suited than BLE for transferring data. BLE has a lower bandwidth and is therefore not as good at transferring large amounts of data. However for small amounts of data such as email and small pictures these drawbacks are not as pronounced. BLE's less stringent security requirements also facilitate the transfer of non-sensitive data.

No Pairing Necessary. BLE does not require that devices pair to transfer data between each other. By avoiding the need to pair at the OS level, an application can choose its preferred and custom method of authentication and encryption. If authentication and encryption are not needed, the application does not have to implement these layers.

Quick Connections. The Bluetooth SIG indicates that devices can connect and transfer data in as little as 3ms [23]. In practice connection and transfer times of just over 1 second have been observed [24].

Mobile Device Penetration. The two most prominent mobile operating systems, iOS and Android, held over 96% of the worldwide smartphone OS market in 2014 [25]. iOS and Android themselves present varying levels of Bluetooth Low Energy support.

Apple introduced full BLE support in early 2011. In early 2015 the app store reported that 98% of devices are versions of iOS that support BLE. In contrast, Android only introduced partial BLE support (Central mode) with the second maintenance release of version 4.3 (Jelly Bean MR2) in July of 2013. Android finally introduced full BLE support (Peripheral mode) in late 2014 with version 5.0. As of April 2015 the Google Play Store reports Android versions 4.3 through 5.1 hold 50% of the Android market share [12]. Note however that only devices designed at production for Android 5.0 will support Peripheral mode; older devices are not guaranteed to gain this functionality [26].

Mobile Device APIs. The iOS CoreBluetooth API is well established since its release in 2011 with iOS 5. Work from a previous paper [27] demonstrated the capabilities of CoreBluetooth, and implementations of iBeacon functionality are now implemented commercially [28].

Android's API Level 18 introduced Bluetooth Low Energy functionality, and API Level 21 extended this functionality to match that of iOS's CoreBluetooth library.   API Level 21 (Lollipop) was released in November of 2014 [29].

Good Range for the Power. For traditional Bluetooth, a class 2 device (mobile phone, Bluetooth headset) transmitting at 2.5 mW has a range of 10 meters, or 33 feet [30]. Experiments with Bluetooth Low Energy with the iPhone 5s have shown reliable connectivity at 50 meters [24]. According to the Bluetooth SIG, the range is better than traditional Bluetooth due to BLE's use of a larger GFSK modulation index [23].

Chapter 3 - SimpBLE Design

This paper presents the SimpBLE framework to help developers use Bluetooth Low Energy as a transport method for arbitrary data. Android was selected as the development language for the initial implementation of SimpBLE due to its market penetration worldwide and relative affordability compared with similar iOS devices. To verify compatibility with iOS, a barebones iOS implementation of SimpBLE transport methods was implemented for iOS 8.2.

3.1 GATT Profile Description

In the same spirit of standardization that the Bluetooth SIG provides defined GATT Profiles, this paper presents the basis for a SimpBLE GATT Profile. This Profile defines the roles of Connection Initiator and Connection Responder, where the Initiator is a GATT Client and the Responder is a GATT Server**.**   The Connection Responder will provide a Message Transfer Service. The Service will be identified by the following UUID randomly generated from Java's UUID class: 73A20000-2C47-11E4-8C21-0800200C9A66.

For purposes of clarity, this UUID will also serve as a base for all Characteristics used by the Service so that they may be represented by a 16 bit UUID. Below is a table of the Characteristics for the GATT Service that SimpBLE will implement.

Message Transfer Service UUID: 73A20000-2C47-11E4-8C21-0800200C9A66			
Characteristic	Function	16 bit UUID	GATT Type
DATA_CtoS	Write Data from Client to Server	0x0101	WRITE
DATA_StoC	Write Data from Server to Client	0x0102	NOTIFY
DATA_ACK	Acknowledge Receipt of Message	0x0105	READ / WRITE

Table 3.1 - SimpBLE GATT Profile Attributes

The SimpBLE service and associated characteristics will be advertised when SimpBLE is invoked in Peripheral mode, and will be scanned for when SimpBLE is invoked in Central mode.

3.2 Architecture

SimpBLE attempts to abstract as much Central or Peripheral specific functionality as possible, so that these concepts are made transparent to the application's developer. Following are the internal details which must differentiate between the Central and Peripheral GAP roles. The components of the Figure 3.1 are described in this section.

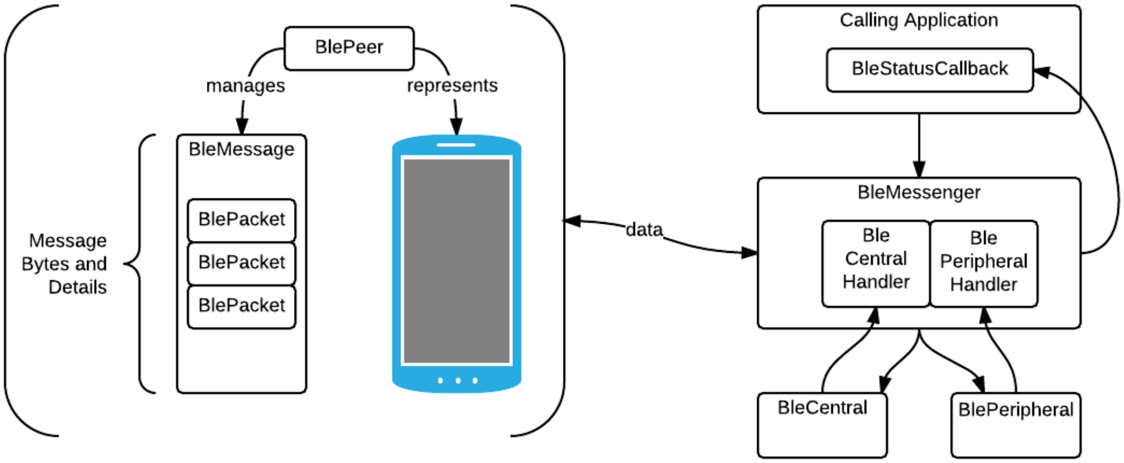


Figure 3.1 - SimpBLE Architecture

3.2.1 Data Objects

BlePacket. Represents an individual packet of data at the GATT layer. An integer to track packet sequence is mapped to a byte array which holds the raw bytes of the packet. This message sequence integer represents the sequence of that particular packet in terms of making up a parent BleMessage.

BleMessage. Represents a message sent to or received from a BlePeer (described next). Contains a collection of BlePackets and methods to construct a full byte array of an entire message from these packets as well as methods to packetize the full byte array of an entire message into BlePackets.

BlePeer. Represents a currently connected GATT Client or GATT Server. Contains a collection of BleMessages which are pending send or are pending completion of receipt, as well as methods to act on these.

BleGattCharacteristic. Wrapper for Android's BluetoothGattCharacteristic which allows for custom callbacks that can be applied to each. Used solely in the layer that managers GAP Peripheral operations (BlePeripheral, described in the next section).

BleCharacteristic. Used to set the service definition of the GATT Characteristics that make up the Service.

3.2.2 Connection Objects and Interfaces

BleMessenger. Responsible for tracking peers using a collection of type BlePeer. Manages connections for and handles callbacks from BleCentral and BlePeripheral classes (described next), and as such is responsible for merging and abstracting GATT communication activity which would otherwise require direct knowledge of Bluetooth GATT characteristics and how these are used with the GAP Central and Peripheral roles.

BleCentral. Manages connectivity as a GATT client. Responds to requests for scanning for Peripheral devices from BleMessenger and reports connection and data events back to BleMessenger through the BleCentralHandler callback. Responsible for making direct calls to the OS's API for GAP Central and GATT client operations.

BlePeripheral. Manages connectivity as a GATT server. Creates GATT Service and Characteristics as directed by BleMessenger. Updates Characteristic values as directed by BleMessenger. Responsible for making direct calls to the OS's API for BLE Peripheral and GATT Server operations.

BleCentralHandler. Interface through which BleCentral notifes BleMessenger of connection and data events.

BlePeripheralHandler. Interface through which BlePeripheral notifies BleMessenger of connection and data events.

BleStatusCallback. Interface through which BleMessenger notifies the calling application of connection and data events.

3.3 Use Cases, Internal Implementation

The following describes the internal functionality SimpBLE implements when invoked by a calling application to perform common communication use cases.

3.3.1 Initiating a Connection

By design, one peer must act in the Central role and the other must act in the Peripheral role. The design of SimpBLE doesn't attempt to abstract out the differences between a Central and a Peripheral in terms of device discovery. A developer may intend for a device to seek out other devices yet not advertise the presence of the application's own device; conversely a device may want to passively offer data and not wish to actively scan for other peers, as the act of scanning takes more power than passively advertising at a low power. Accordingly later when the public interface of SimpBLE is described Scan functionality will be separated from Accept functionality. Figure 3.2 illustrates the connection steps described in the next two sections.

Central. For the device that enters the Central role, it issues a call to the Bluetooth LE adapter to scan for advertising peripherals. The adapter will filter to only connect to those devices that advertise SimpBLE's Message Transfer Service UUID. BleMessenger is notified of this connection via the BleCentralHandler callback, at which time it instantiates a new BlePeer object and adds it to a lookup structure, indexed by the received Bluetooth address. Once identified, the Central peer connects to the Peripheral and itemizes the characteristics offered. If the characteristics match the service definition noted in Table 3.1, the Central subscribes to the **DATA_StoC** GATT characteristic by writing the appropriate byte value to that Characteristic's Client Characteristic Configuration descriptor to enable notifications. [31] Upon verification from the connected Peripheral that this subscription is successful, SimpBLE considers this peer connected and marks the corresponding BlePeer object as having a data transport open.

Peripheral. For the device that assumes the Peripheral role, it begins advertising SimpBLE's Message Transfer Service UUID and associated characteristics. When connected to by a Central device, BleMessenger instantiates a BlePeer object and adds it to a map indexed by the Central's Bluetooth address. It then waits for the Central to subscribe to the **DATA_CtoS** characteristic, and returns a verification when that subscription occurs. At this point SimpBLE marks this peer as having a data transport open.

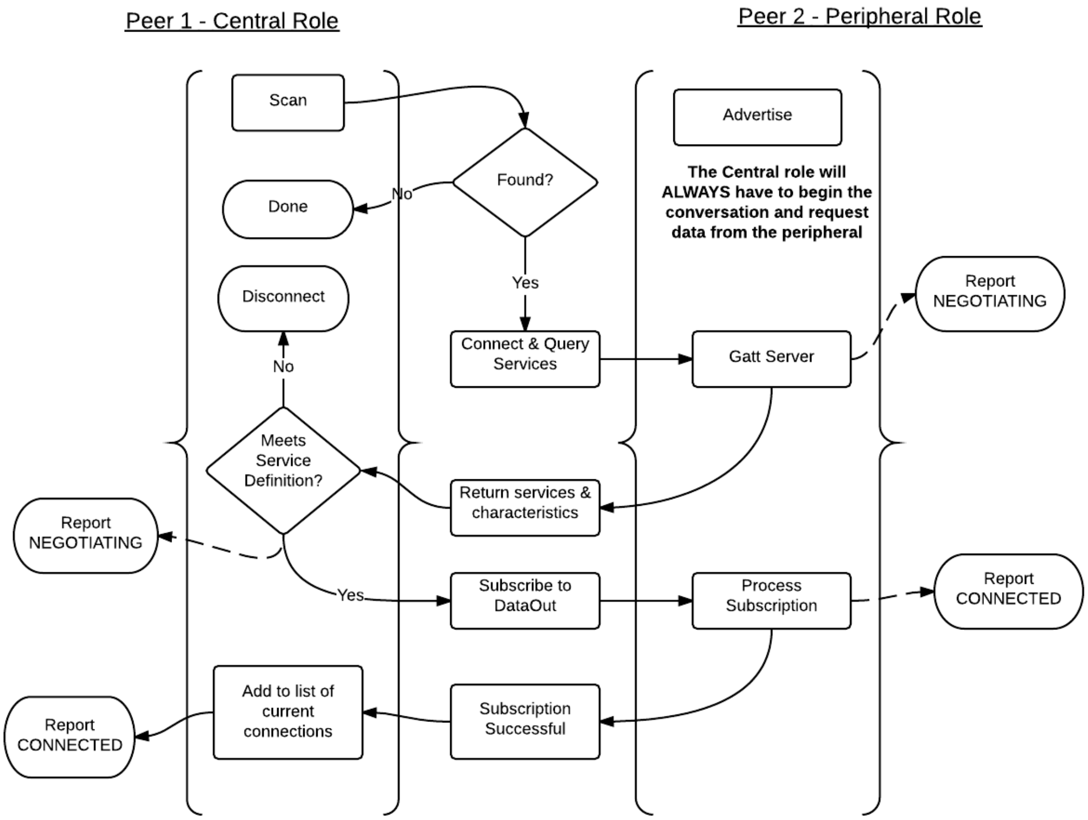


Figure 3.2 - SimpBLE Handshake

3.3.2 Preparing a Message

SimpBLE creates packets of byte size 20 to send to the device's Bluetooth stack. The packet size is derived from the maximum per-packet payload allowed in Android versions before 5.0. The calling application submits a message in a byte array (or a helper object) to BleMessenger along with a target peer's identifier. BleMessenger instantiates a BleMessage object using these bytes and adds it to the appropriate BlePeer. The BleMessage object then creates a set of BlePackets, each BlePacket holding a byte array of data ready for transmission to the recipient. The layout of the byte array for packets used by SimpBLE is provided in Figure 3.3.

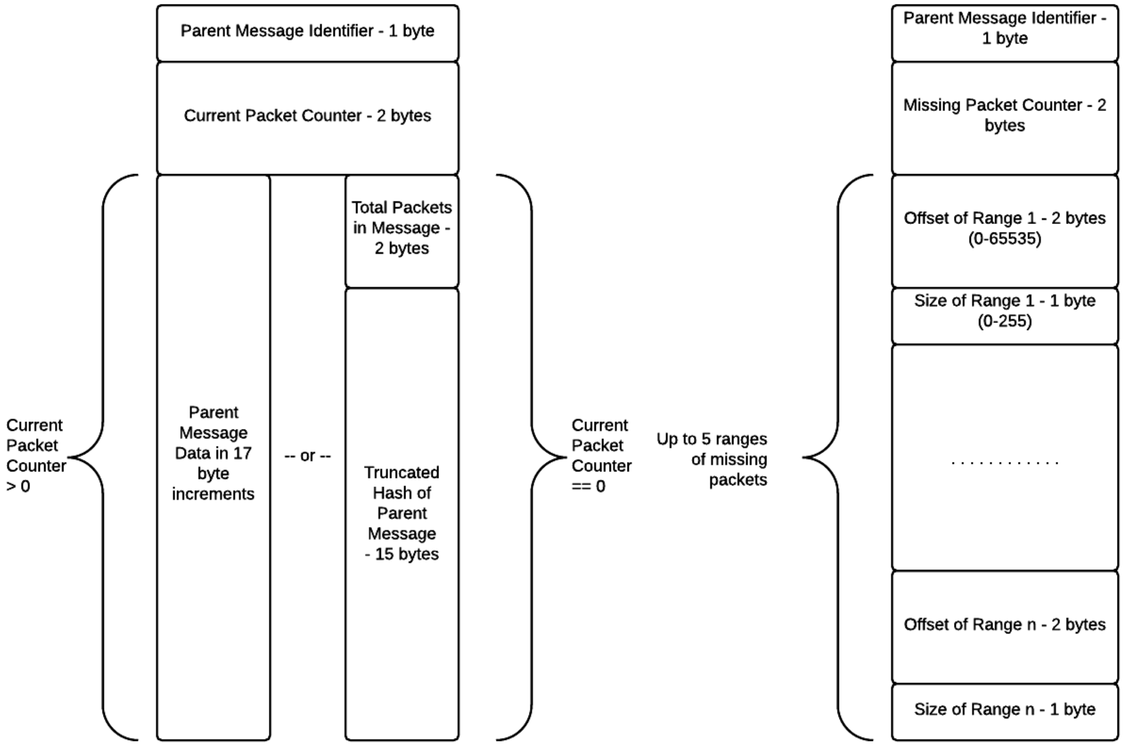

Parent Message Identifier - 1 byte
Current Packet Counter - 2 bytes
Current Packet Counter > 0
Parent Message Data in 17 byte increments
-- or --
Total Packets in Message - 2 bytes
Truncated Hash of Parent Message - 15 bytes
Current Packet Counter == 0
Up to 5 ranges of missing packets
Parent Message Identifier - 1 byte
Missing Packet Counter - 2 bytes
Offset of Range 1 - 2 bytes (0-65535)
Size of Range 1 - 1 byte (0-255)
. . . .
Offset of Range n - 2 bytes
Size of Range n - 1 byte

Figure 3.3 - SimpBLE Data and Ack Packets

In each payload packet: SimpBLE uses 1 byte to identify the message for a connection session (max 256 messages per connection session), 2 bytes to identify the current packet (max 65,536 packets), and 17 bytes for the message payload itself. Since 2 bytes are used for the packet counter, there are a maximum of 65,535 packets per message (packet 0 is used as the header packet).   At 17 bytes of payload a packet, SimpBLE allows for messages of a maximum size of 1,114,095 bytes.

3.3.3 Transmitting Packets

SimpBLE identifies the target connected peer using the Bluetooth address and references the appropriate BlePeer object. The BlePeer object references its own collection of BleMessage objects that have not been sent, and provides a BleMessage object to BleMessenger. Using transport mechanisms as noted below, BleMessenger gathers the pending BlePackets from the provided BleMessage and loops over each, sending to the target device via the GATT Characteristic as noted below.

Transmitting from the Central Role. The only way for a Central to transmit data to a peripheral is by updating the value of a GATT Server's Write Characteristic, noted in Chart 1a as **DATA_CtoS.** Since the Bluetooth specification allows a Central to be connected to multiple Peripheral peers, the Bluetooth address of the target peripheral is necessary to direct each message to the appropriate recipient. BleMessenger directs the BleCentral object to deliver each packet to the **DATA_CtoS** Characteristic. There is a 50ms delay added in between each WRITE operation, as developers have experienced problems overwhelming the Android 4.3 Bluetooth stack outgoing data buffer, leading to Characteristics not being properly sent. BleCentral then calls the underlying android.bluetooth API to deliver the packet to the target Characteristic on the GATT Server.

Transmitting from the Peripheral Role. A Peripheral role has many options to transmit data to its Central peer. SimpBLE uses the Notify attribute, noted in Chart 1a as **DATA_StoC**. The Indicate attribute was also an option for transfer, however that proved too slow as the Central must acknowledge receipt of each packet for an Indicate.

While the Bluetooth 4.0 specification allows a single Central peer to be connected to multiple Peripheral peers, a Peripheral can only be connected to a single Central. Future versions of Bluetooth have indicated the capability of allowing a Peripheral to host multiple Central peers, so BleMessenger uses an index to look up the connected peer, although as only a single peer can be connected this strategy is currently unnecessary.

BleMessenger checks to ensure that the Client device is subscribed to the **DATA_StoC** Characteristic, and then directs the BlePeripheral object to update this Characteristic with the byte value of each BlePacket for the message. The android.bluetooth layer then delivers each packet to the subscribed Client device.

3.3.4 Message Acknowledgment

Figure 3.4 illustrates the message acknowledgment process described in this section After a Central completes sending a message to the Peripheral, BleMessenger follws up by directing BleCentral to issue a Read request to the Peripheral's **DATA_ACK** GATT characteristic. BlePeripheral relays the calling Bluetooth address and Characteristic called to BleMessenger, which upon seeing **DATA_ACK** was called looks up the BlePeer via the Bluetooth address and identifies any incoming BleMessages for that peer, which it then queries for any missing BlePackets. An ACK packet detailing which BlePackets are missing will be returned. If any are missing, BleMessenger directs

BleMessage to requeue these and will re-send at the next send event.   If none are missing, BleMessenger directs BlePeer to mark that particular message as sent.

After a Peripheral completes sending a message to the Central, the Central initiates the acknowledgment action (as the Central must initiate any connection event). Because this action is initiated immediately after a successful receipt, the BleMessage object can be referenced directly. BleMessenger directs BleCentral to issue a Write request in the format of an ACK packet to the Peripheral's **DATA_ACK** GATT Characteristic indicating the message was fully received. If the message was not fully received, then a cleanup routine will identify missing packets and follow up with an ACK packet which instead will indicate which packets were not received.

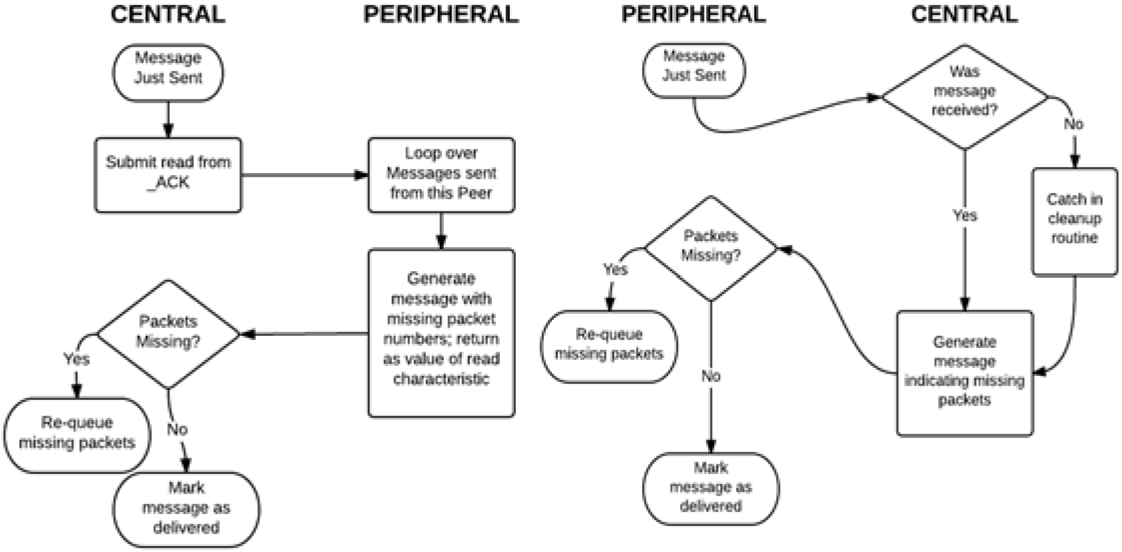


Figure 3.4- Message Acknowledgment

3.3.5 Message Receipt

Regardless of whether the packet stream is incoming via **DATA_StoC** or **DATA_CtoS**, BleMessenger reads the incoming bytes into a BlePacket and uses the Bluetooth address to identify the sending peer's corresponding BlePeer object, using the first byte of the packet to identify to which BleMessage that BlePacket belongs. Based on the first BlePacket received for the parent message, a BleMessage knows how many packets to expect. Once all BlePackets have been received, BleMessage reconstructs the original message and BleMessenger delivers this message to the calling application.

3.4 Testing and Performance

A custom rudimentary benchmarking application was used to test connection distance and measure transfer speeds between a Moto E 4G LTE (Android 5.0.2) and a Nexus 5 (Android 4.4.4). Basic transport mechanisms were implemented using CoreBluetooth on an iPad Mini (iOS 8.3) to demonstrate that iOS and Android can indeed communicate over BLE. Initially the iOS LightBlue App [32] was used for Service and Characteristic exploration between the iPad Mini and the Moto E.

BLE advertises raw transfer rates of 1Mbps, with a maximum theoretical application throughput of 236.7 kbps [33], although testing in literature has only seen as much as 58 kbps using embedded devices [34]. The testing for this paper consistently yielded just under 10 kbps, significantly slower but enough to transfer a 100kB file in just over a minute [Figure A.2]. Proof of concept testing between the Moto E and the iPad

Mini resulted in similar throughput results. This testing was made possible by adapting code from a previous project [27].

Note that due to the asymmetric nature of BLE transfer, these results were achieved with the device in GAP Peripheral mode sending a payload to the device in GAP Central mode. For transfers from a Central to Peripheral, throughput of just over 2kbps was achieved. This throughput, while not suitable for large payloads, is suitable for a small amount of data such as the transfer of encryption key information.

Chapter 4 – SimpBLE Public Interface

The goal of SimpBLE is to abstract out the cumbersome details of message transfer over Bluetooth Low Energy. As such a calling application will only interface with the BleMessenger class. BleMessenger exposes a short set of public methods and properties, and will relay information back to the calling application via several methods from a single registered callback.

4.1 OS Resources

Because of how Android allocates handles to its resources, some of the lower level details must still be addressed by the calling application. To implement BLE actions, the constructor for SimpBLE needs a pointer to the following Android system level resources: BluetoothManager, BluetoothAdapter, and Context..

```
BluetoothManager btMgr = (BluetoothManager)
        this.getSystemService(Context.BLUETOOTH_SERVICE);

BluetoothAdapter btAdptr = btMgr.getAdapter();

Context ctx = this;
```

To instantiate the BleMessage class, the application passes in handles to the BluetoothManager, BluetoothAdapter, the application’s context, and finally a callback to handle BleMessenger events.

```
bleMessenger = new BleMessenger(btMgr, btAdptr, this, bleStatusCallback);
```

For an Android application to perform Bluetooth operations and initiate device discovery, the following permissions must also be set in the AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
```

The AndroidManifest.xml file must specify a minimum API level of 18. Scanning is more optimized in API levels 21 and above, however API levels 18 through 20 offer scanning that proved to be functional and robust enough during testing to merit compatibility.

```
<uses-sdk android:minSdkVersion="18" android:targetSdkVersion="21" />
```

The application developer can decide whether or not to take certain actions depending on whether the device supports Peripheral (GATT Server) mode or not.

```
if (bleMessenger.SupportsAdvertising) {
        // perform actions based on whether or not this device can advertise
}
```

4.2 Connection Management

To look for peers, the application calls the ScanForPeers method, passing in the duration to scan in milliseconds. SimpBLE will auto-connect to any detected device that advertises the Service definition specified in Section 3.1. Whenever a device is connected to or its connection status updated, the callback peerConnectionStatus will be executed (detailed further down).

```
bleMessenger.ScanForPeers(MillisecondsToScan);
```

To be found by another device which is actively scanning, the application makes a simple call to start advertising.

```
bleMessenger.StartAdvertising();
```

When the application no longer wants to be visible to nearby devices, it makes a similar call to stop advertising.

```
bleMessenger.StopAdvertising();
```

SimpBLE makes assumptions about certain advertising parameters such as power and duty cycle, and sets up the advertising functionality per the specifications in Section 3.1. Any connections made to the device are reported through the function peerConnectionStatus in the BleStatusCallback, regardless of which device initiated the connection.

```
public void peerConnectionStatus(String remoteAddress, int ConnectionStatus) {
      if (ConnectionStatus == BleMessenger.CONNECTION_NEGOTIATING) {
            // negotiating connection with remoteAddress
      } else if (ConnectionStatus == BleMessenger.CONNECTION_CONNECTED) {
            // connected to remoteAddress
      } else if (ConnectionStatus == BleMessenger.CONNECTION_DISCONNECTED) {
            // disconnected from remoteAddress
      }

}
```

Note that connection status of CONNECTION_CONNECTED refers to a connection at the SimpBLE abstraction level. The devices are in fact connected at the GATT level when SimpBLE reports CONNECTION_NEGOTIATING however the ability to send and receive data has not yet been finalized.

The calling application should maintain a collection of Bluetooth addresses passed in from the remoteAddress parameter in peerConnectionStatus , and can direct BleMessenger to disconnect from any of these.

```
bleMessenger.disconnectPeer(BluetoothAddress);
```

Once the connection has been terminated, the callback peerConnectionStatus is called with a ConnectionStatus of CONNECTION_DISCONNECTED.

4.3 Sending a Message

When a device is connected, SimpBLE reports the connection back to the calling application along with the Bluetooth address of the remote device as a way for both SimpBLE and the calling application to keep track of peers. The calling application will use this address as an index with which to reference a message recipient, allowing a message to be sent to a single peer while connected to multiple devices.

The calling application calls the AddMessage method, passing in the content of the outgoing message to BleMessenger as either an array of bytes or an instantiation of the abstract BleApplicationMessage helper class. The AddMessage method returns a truncated SHA-1 digest of the message bytes so the calling application can uniquely identify the message by content.

```
bleMessenger.AddMessage(BluetoothAddress, ApplicationMessage);

// or

bleMessenger.AddMessage(BluetoothAddress, RawByteArray);
```

When a message is queued and a peer is connected, BleMessenger periodically checks for any messages that are pending send. After sending the message, BleMessenger attempts to determine if a message was successfully delivered. If so, it executes the BleStatusCallback function messageDelivered.

```
public void messageDelivered(String remoteAddress, String payloadDigest, int
       parentMessageId) {
       // message identified by payloadDigest was delivered to remoteAddress
}
```

4.4 Receiving a Message

When a message is received in its entirety, the handleReceivedMessage method from the callback BleStatusCallback is executed. If the application developer has been using a custom class derived from the provided BleApplicationMessage helper class, the details of the message can be easily rebuilt.

```
public void handleReceivedMessage(String remoteAddress, int parentMessageId,
       boolean messageIntact, byte[] MessageBytes) {

       ApplicationMessage incomingMsg = new ApplicationMessage();

       String incomingDigest = incomingMsg.SetRawBytes(MessageBytes);

       incomingMsg.BuildMessageDetails();
```

```
    // assuming as single byte is used for the message type
    int messageType = incomingMsg.MessageType & 0xFF;

    // who’s supposed to get it, who sent it, what does the message contain
    byte[] recipientFingerprint = incomingMsg.RecipientFingerprint;
    byte[] senderFingerprint = incomingMsg.SenderFingerprint;
    byte[] messagePayload = incomingMsg.MessagePayload;
    byte[] messageHash = incomingMsg.BuildMessageMIC();

}
```

4.5 Usage By a Calling Application

Earlier the use case of a two friends exchanging a photo between an Android and iOS device was presented. An application that implements this with SimpBLE could be as simple as loading the image into a raw byte array, two devices connecting, and one device sending the byte array to the other.

If the information is more sensitive than (for instance) a photo suitable for social media, an application may choose to offer authentication and encryption. In this case a pairing mechanism will need to be implemented as well as a layer of encryption. Sample implementations for these layers are detailed in Chapter 5.

Chapter 5 – Secure Message Passing Application

In Chapter 2 the use case of one friend needing to send the other a non-sensitive message via peer to peer transport between heterogeneous platforms was posited. For situations like these an application may not implement security, however for more sensitive information authentication and encryption may be required. This chapter presents a sample application to illustrate how a developer may use SimpBLE to implement a custom security layer.

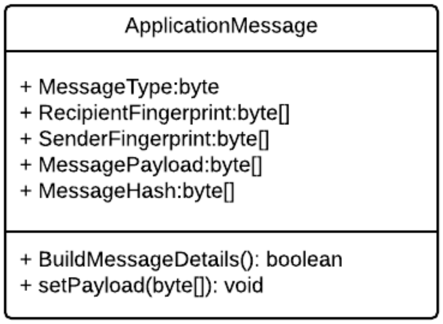
5.1 Authentication and Encryption

This particular application relies on Public/Private key authentication and encryption, as well as symmetric key encryption. When a message is prepared for sending, the plaintext of the original message is encrypted with symmetric key encryption and the ciphertext sent to the recipient. The symmetric key itself is then encrypted with the recipient’s public key and sent to the recipient in a separate message.

5.1.1 Key Exchange

The application requires that an RSA key pair is generated and stored on each device. This implementation on Android uses the Android Keystore Provider, which identifies this application via a random UUID written to the installation directory of the app.

For an installation, the application will generate an **Identity** message of type ApplicationMessage (an instantiation of the abstract BleApplicationMessage helper class presented in chapter 4) with the fields detailed below.



ApplicationMessage

+ MessageType:byte
+ RecipientFingerprint:byte[]
+ SenderFingerprint:byte[]
+ MessagePayload:byte[]
+ MessageHash:byte[]

+ BuildMessageDetails(): boolean
+ setPayload(byte[]): void

Figure 5.1 - Application Message Class

Message Type is used to indicate the type of message being sent. In this case the meessage type is 0x01, indicating an identity message.

SenderFingerprint is used to identify the sender of the message and is always the SHA-1 digest of the sender's Public Key (20 bytes).

RecipientFingerprint is used to identify the recipient of the message and is generally the SHA-1 digest of the recipient's Public Key. In the case of an Identity message no specific recipient is necessary, thus this field is the byte 0x00 repeated 20 times.

Payload is the data destined for the message recipient. In this case, the payload is the sender's Public Key.

To begin the identity exchange process, each user starts the app on their phones. User A advertises connectability and User B scans for User A's phone. The two phones connect over BLE and each user is notified the app is ready to exchange data. Each user pushes a button to queue an identity message and passes it to SimpBLE, which relays the message to the other device.

Upon receipt of an identity message, a screen pops up on each device, displaying a hex representation of the other party's Public Key along with a hex representation of a SHA1 fingerprint of the Public Key (Figure 5.3).   The users confirm each others' fingerprints by voice using the first 4 to 8 hex octets. Each user enters in a friendly name to identify the other person (Figure 5.2). These values are saved into the application's SQLite database.

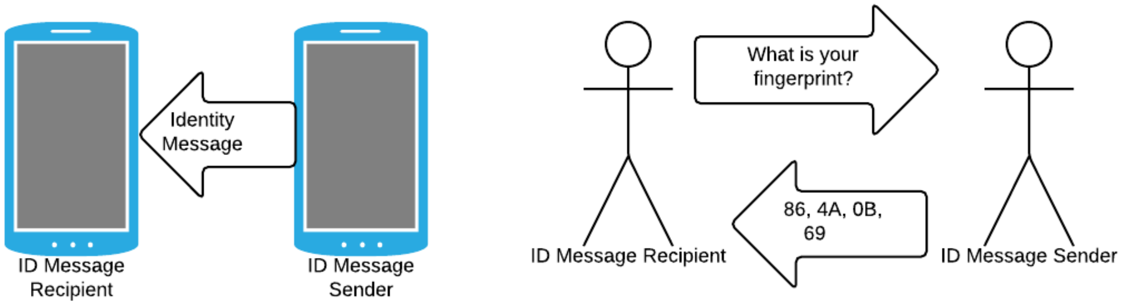
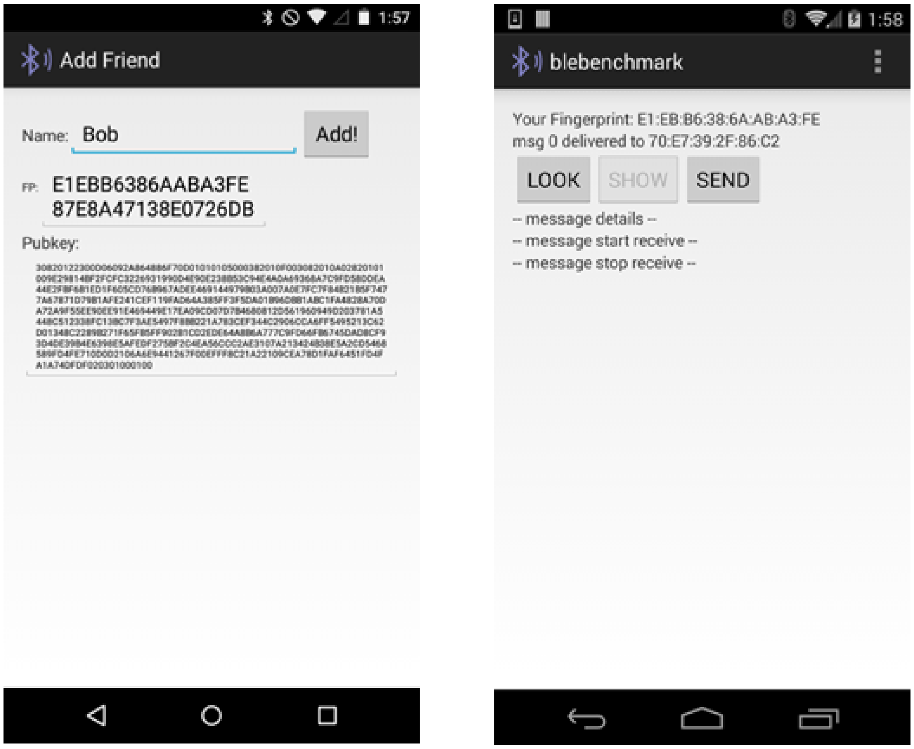


Figure 5.2 – Identity Exchange, Manual Authentication



Name: Bob

FP: E1EBB6386AABA3FE
87E8A47138E0726DB

Pubkey:

```
30820122300D06092A864886F70D01010105000382010F003082010A02820101
009E298148F2FCFC3226931990D4E90E238853C94E4A0A69368A7C9FD58DDEA
44E2F8F681ED1F605CD76B967ADEE469144979803A007A0E7FC7F84821B5F747
7A67871D79B1AFE241CEF119FAD64A385FF3F5DA01B96D8B1ABC1FA4828A70D
A72A9F55EE90EE91E469449E17EA09CD07D7B4680812D5619609490203781A5
448C51233BFC13BC7F3AE5497F8B8221A783CEF344C2906CCA6FF5495213C62
D01348C22898271F65FB5FF902B1C02EDE64A886A777C9FD66F86745DAD8CF9
304DE39B4E6398E5AFEDF2758F2C4EA56CCC2AE3107A213424B38E5A2CD5468
589FD4FE710D002106A6E9441267F00EFFF8C21A22109CEA78D1FAF6451FD4F
A1A74DFDF020301000100
```

Your Fingerprint: E1:EB:B6:38:6A:AB:A3:FE
msg 0 delivered to 70:E7:39:2F:86:C2

-- message details --
-- message start receive --
-- message stop receive --

Moto E, Receiving ID

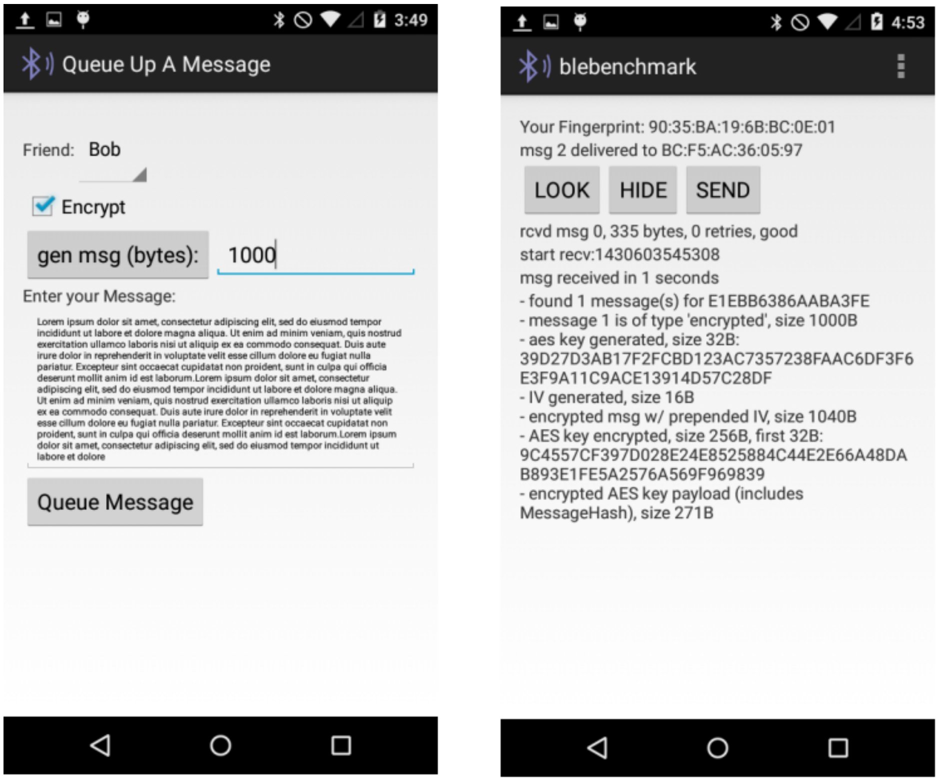
Nexus 5, Sending ID

Figure 5.3 - Identity Exchange, Implemented on Android

Alternatively the public keys and fingerprints could be exchanged beforehand – for instance via secure email - and copy/pasted directly into the Add Friend dialog. The friends have now authenticated to each other and can trust that the friendly name stored in the application corresponds to their friend's device.

5.1.2 Encrypted Message Transfer

Once the application has a friend's Public Key and Fingerprint stored in its database, a user can author a message and specify that friend as a recipient.



Friend: Bob

☑ Encrypt

gen msg (bytes): 1000

Enter your Message:

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore

```
Your Fingerprint: 90:35:BA:19:6B:BC:0E:01
msg 2 delivered to BC:F5:AC:36:05:97
rcvd msg 0, 335 bytes, 0 retries, good
start recv:1430603545308
msg received in 1 seconds
- found 1 message(s) for E1EBB6386AABA3FE
- message 1 is of type 'encrypted', size 1000B
- aes key generated, size 32B:
39D27D3AB17F2FCBD123AC7357238FAAC6DF3F6
E3F9A11C9ACE13914D57C28DF
- IV generated, size 16B
- encrypted msg w/ prepended IV, size 1040B
- AES key encrypted, size 256B, first 32B:
9C4557CF397D028E24E8525884C44E2E66A48DA
B893E1FE5A2576A569F969839
- encrypted AES key payload (includes
MessageHash), size 271B
```

Moto E, Generating a Random 1kB message

Moto E, Encrypting Payload and AES Key

Figure 5.4 - Queue Message and Encrypt for Sending

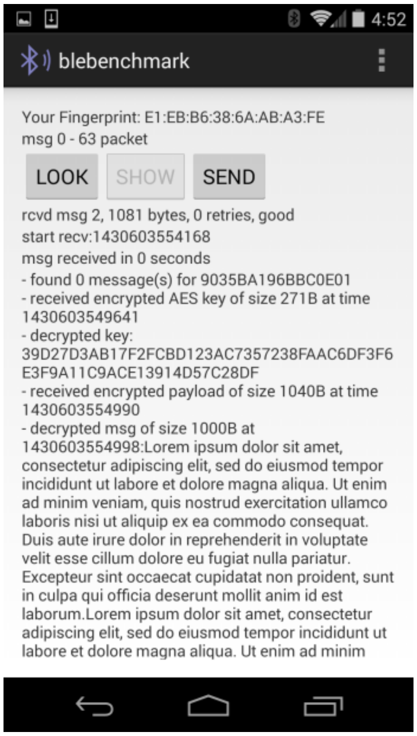
Queueing a message. To require encryption for a message, the user checks a box when authoring a message. The body of the message is stored in the SQLite database as plain text. When the peers reconnect and identify each other, the database is queried for any messages queued for the connected peer. If the message found for the connected peer requires encryption, the system will encrypt the original message and create an additional message providing the encryption key used to encrypt the original message.

Encrypting and sending the message. The system will generate a one-time random 256 bit AES key and a one-time random 128 bit Initialization Vector (IV). The message plaintext, AES key, and IV are provided to a simple implementation of the javax.crypto library [35] to generate the message ciphertext. The IV is then prepended to the encrypted ciphertext to create an encrypted payload which is sent to the message recipient.

Encrypting and sending the key. The app then takes the AES key used to generate the ciphertext and RSA encrypts it by using the Public Key stored for the message recipient. This encrypted key is then appended to a SHA-1 digest of the original message and sent to the message recipient.

Receiving and decrypting the key. The message recipient decrypts the key message payload using the Private Key corresponding to the Public Key that was used to authenticate with the message sender, and stores that decrypted symmetric key in a Map indexed by the digest of the plaintext message.

Receiving and decrypting the message. The message recipient stores the AES-encrypted payload in another Map indexed by the hash of the plaintext message. The receiving application then uses the digest of the received message to look up the previously decrypted AES key. If it exists, then the AES key is pulled and used to decrypt the payload, which is then displayed in plain text to the user.



```
Your Fingerprint: E1:EB:B6:38:6A:AB:A3:FE
msg 0 - 63 packet
rcvd msg 2, 1081 bytes, 0 retries, good
start recv:1430603554168
msg received in 0 seconds
- found 0 message(s) for 9035BA196BBC0E01
- received encrypted AES key of size 271B at time
1430603549641
- decrypted key:
39D27D3AB17F2FCBD123AC7357238FAAC6DF3F6
E3F9A11C9ACE13914D57C28DF
- received encrypted payload of size 1040B at time
1430603554990
- decrypted msg of size 1000B at
1430603554998:Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco
laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt
in culpa qui officia deserunt mollit anim id est
laborum.Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim
```

Figure 5.5 - Receipt and Decryption of AES Key and Original Message

5.1.3 SQLite Data Objects

To support static data between application sessions should the application be terminated before a message can be sent, the SQLite library is used (available on Android and iOS). As this is a fairly trivial system for demonstration, only two tables are needed whose relationships are noted in Figure 5.6.

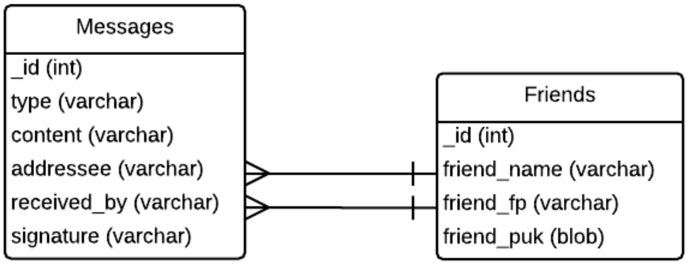


Figure 5.6 - ER Diagram, Messages and Friends

Friends. The Friends table stores a peer's Public Key, the Public Key's Fingerprint as a hex string, and a friendly name for a user's Friend. The _id field is unique per row and is used to simplify row-level operations.

Messages. The Messages table stores incoming and outgoing messages. The message type is stored to indicate whether or not this message will be encrypted. The message content is stored in plaintext, while the friendly name of the recipient allows the system to look up recipient details from the Friends table. A hex representation of the recipient's fingerprint is stored (in case this message passes through an intermediary), and a SHA-1 signature of the message's payload provides unique message identification.

5.2 Summary of Message Passing Application

The authentication mechanism presented in this implementation has several advantages over the traditional Bluetooth pairing mechanism provided by common mobile phone operating systems.

- **The app does not cede control.** Since the application manages the pairing and does not have to cede control to the operating system's pairing mechanism, the user is never forced to leave the application to interact with a separate dialog.
- **The authentication step can happen beforehand.** In the case that two users have already authenticated each others' Public Keys, the users can add each these Public Keys manually into the application with no loss in trust.
- **The keys can be moved to a different device.** While the sample application does not have a mechanism to move keys to a different device, this functionality is supported by virtue of the key store being separate from the authentication mechanism itself.

Along with flexibility in security comes some vulnerability. Because the key data is not maintained and protected at the OS level, the possibility of surrendering this key to a malicious actor may be more likely. Also note that this implementation does not ensure that the connected peer actually owns the private key for the recipient's public key. This shortcoming is mitigated somewhat in that regardless of who gets the message, only the holder of the private key can decrypt the message. While this may be a problem for highly secure messages, this enables other devices to act as middle men to help deliver a message securely to its ultimate destination.

The overhead added by this encryption strategy in terms of computation complexity is negligible. On the Moto E 4G LTE device (Android 5.0.2) used for testing, AES encryption of a 100KB message required 8ms to complete. Decryption of the

100KB payload on the receiving device (Nexus 5, Android 4.4.4) required 14ms to complete. RSA encryption of the 32 byte key requires around 2ms to complete, while RSA decryption of the 32 byte key averaged around 80ms.

In terms of message complexity, the primary overhead is due to the addition of a separate key message. This overhead is mitigated somewhat due to the small size of the key message itself (just over 256 bytes assuming a 2048 bit RSA key), allowing this message to be transferred in just over a second. Also note that AES encryption does not increase message size [36], so the increase in size of transferred messages is negligible.

Chapter 6 – Distribution of Secret Shares

A novel use case for an environment rich in wireless peer to peer connectivity is that of a mobile threshold secret sharing scheme. A threshold secret sharing scheme is a way of splitting up a message such that the original message cannot be reconstructed without a pre-determined minimum number of shares; Adi Shamir presented the first implementation of such a scheme [37]. Note that with Shamir's scheme, each share must be the size of the original message.

6.1 Motivating Scenarios

Group Trust. A parent has three children, each who have a smartphone. The parent wants each child to have access to money in case of an emergency but does not trust any child individually. The parent gives the oldest child a PIN operated ATM card, but not the PIN. Instead he uses a smartphone application that splits the PIN into 3 shares, where a minimum of 2 shares is needed to reconstruct the original PIN. He gives each child's device a single share, ensuring that at least 2 children must confer in order to rebuild the PIN.

Untrusted Couriers. A message sender needs to get a message to a recipient in an area without infrastructure and cannot deliver the message personally. The sender and recipient do not share any encryption keys, symmetric or otherwise. The sender uses a smartphone application to split up a short message into 5 secret shares such that any 3 are needed to rebuild it. The application transmits a single share each onto smartphones possessed by 5 separate message couriers. As long as only 2 of these couriers collude to attempt to rebuild the message and at least 3 of the couriers make it to the recipient and are able to relay their 3 shares, the message can be delivered securely.

Dead Drop.   If a message sender and recipient cannot meet in person, the sender drops a message off at a pre-determined location with the expectation that the recipient will retrieve the message. As recent as 2006, Russian intelligence agents observed agents for the British government using a dead drop to gather information from Russian informants; the dead drop was a wireless receiver and transmitter hidden in a fake rock [38]. The scenario below presents a mobile and distributed dead drop implementation that overcomes the vulnerability in having a dead drop at only a single location.

Alice needs to deliver a message to Bob, however Alice and Bob are both being tracked and cannot risk being seen together. Neither can communicate using infrastructure as the email accounts and internet access of both are being tracked. Alice loads an application on her smartphone and splits a message into three secret shares, requiring a threshold of two shares to reconstruct the message. Alice goes to a coffee shop and surreptitiously drops a share onto the shop’s peer to peer electronic message board. She then goes to a different part of the city and drops another share onto an untrusted courier’s phone inside a crowded shopping mall. On her way back to her apartment she stops by a sports bar and drops off a final share at the electronic message board at that location. A few hours later, Bob travels to an as-yet unvisited location where he knows the courier will be and picks up a share from that courier, then travels to the original coffee shop and retrieves a second share of the message. Bob then reconstructs the original message using the application on his phone.

Section 6.2 describes an Android implementation of such an application.

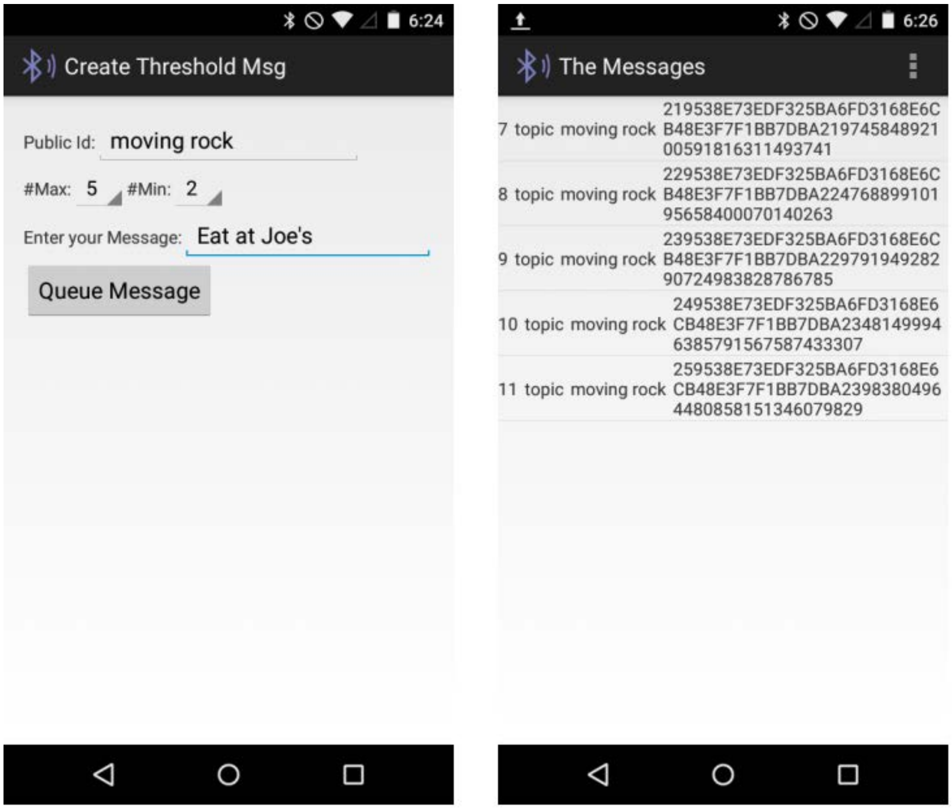
6.2 System Design

As with the encrypted message transfer from the previous section, the functionality described in this section is meant to show an application of a particular cryptographic technique using Bluetooth Low Energy as a peer to peer transport between wireless devices. The database and object design in this section are identical to the previous chapter.

6.2.1 Creating a Message

For this type of message the application need not identify a particular recipient as the final recipient is any peer who gains the minimum number of shares needed to rebuild the original message. The application prompts the user for the number of shares to generate as well as the threshold number of shares needed to recreate the message. The user also enters an identifying topic name for the message, which is used in place of the recipient name. The type of message is stored as “topic”.

The plaintext of the message is not stored in the database. Instead the application uses an adapted version of a Java implementation to generate secret shares [39], and stores each of these shares as an individual message. The content of each message is such that the first character indicates the number of shares required to rebuild the message (1-9). The second character indicates which share number a particular message is. The next 40 characters are a hexadecimal representation of a SHA-1 signature of the original plaintext, and the remaining characters are a hexadecimal representation of the share itself. Figure 6.1 details a message generated with 5 total shares, any 2 of which are required to rebuild the original message. The message was authored on a Moto E 4G LTE running Android 5.0.2.



Public Id: moving rock

#Max: 5 #Min: 2

Enter your Message: Eat at Joe's

```
7 topic moving rock 219538E73EDF325BA6FD3168E6C
                    B48E3F7F1BB7DBA219745848921
                    00591816311493741
8 topic moving rock 229538E73EDF325BA6FD3168E6C
                    B48E3F7F1BB7DBA224768899101
                    95658400070140263
9 topic moving rock 239538E73EDF325BA6FD3168E6C
                    B48E3F7F1BB7DBA229791949282
                    90724983828786785
10 topic moving rock 249538E73EDF325BA6FD3168E6
                     CB48E3F7F1BB7DBA2348149994
                     6385791567587433307
11 topic moving rock 259538E73EDF325BA6FD3168E6
                     CB48E3F7F1BB7DBA2398380496
                     4480858151346079829
```

Figure 6.1 - Creating a Shared Secret

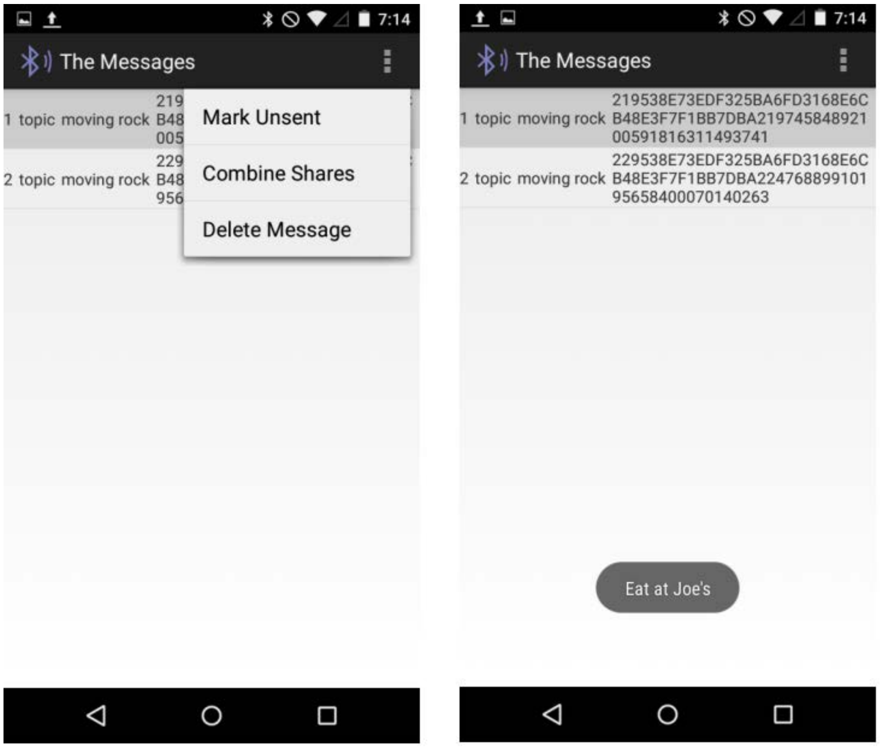
6.2.2 Distributing a Message

Upon connecting, the peers perform the identification handshake to exchange public keys, as detailed in Chapter 5. The sending application then checks to see if a share for this message has already been sent to the connected peer. If not, the originating application sends a share to that device and stores this peer's fingerprint to disallow subsequent deliveries of message shares to that peer. In the presented example, two shares were delivered, one each to a Galaxy Nexus (custom build of Android 5.1) and a Nexus 5 (Android 5.1)

6.2.3 Re-assembling a Message

The particular implementation tested for this paper assumes that any peer who has not already received a share is interested in receiving an unsent share. Upon receipt of a share message, the application stores the received message in the Messages table. The user can then view the received shares and pick any share for the secret message, then instruct the application to Combine Shares. If enough have been received then the original message is displayed on the device as shown in Figure 6.2.

To re-assemble the message, the application was removed and freshly installed onto the Moto E 4G LTE to create the appearance of a new peer. This step was necessary due to the number of devices required to test multiple peers in this simulation. This device then connected to each of the other two tested devices to gather the shares that were originally dropped off. The shares were reconstructed as shown in Figure 6.2.



1 topic moving rock 219 B48 005

2 topic moving rock 229 B48 956

1 topic moving rock 219538E73EDF325BA6FD3168E6C B48E3F7F1BB7DBA219745848921 00591816311493741

2 topic moving rock 229538E73EDF325BA6FD3168E6C B48E3F7F1BB7DBA224768899101 95658400070140263

Eat at Joe's

Figure 6.2 - Reassembling a Message

6.3 Future Work

Using Shamir's original scheme, each share is the size of the secret message itself. While Shamir's scheme is theoretically secure, computationally secure implementations have been presented which will allow for each share to be smaller than the original secret [40], allowing for larger messages to be created.

If social media location based apps become more prevalent and electronic peer to peer bulletin boards become popular, a share can be dropped in the open and cloaked by using popular topics, such as sports teams' names. The sending application can also use GPS to ensure that it doesn't drop off more than one share at a physical location.

Chapter 7 – Implications and Future Work

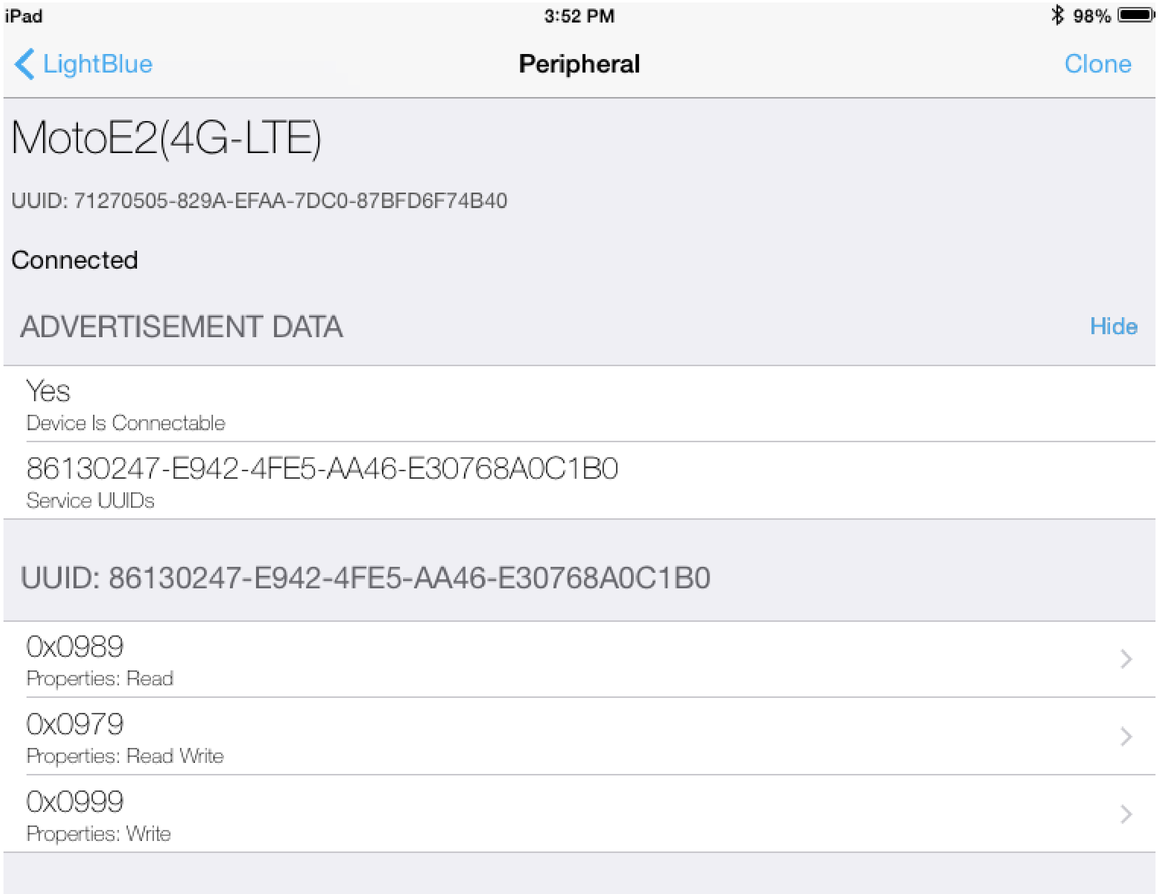
Enabling iOS and Android devices to communicate via an easy to implement peer to peer standard opens the doors for a wide variety of applications from mobile application developers, both in the private sector and in academia. To facilitate application development, a formal GATT Profile for arbitrary message transfer can be presented to the Bluetooth SIG and adopted as an implementable standard. All code for this project is available on GitHub at https://github.com/ludwigmace/blebenchmark [41] and can be used freely. The next most logical step is to re-implement the full SimpBLE framework on iOS and not just the rudimentary transport mechanisms used for testing.

Without the requirement for OS level pairing and peer to peer capabilities between Android and iOS made possible, the modeling of message passing routines for Mobile Ad Hoc Networks (MANETs) can be simplified and greatly increase a potential testing population. Test subjects may also be more willing to install these apps as the battery drain from Bluetooth Low Energy is much less than from Classic Bluetooth.

The recently adopted specification for Bluetooth 4.2 increases the raw BLE packet sizes are from 27 bytes to 251 bytes. This mirrors the ability of iOS 8 and Android 5 to negotiate an MTU size larger than the maximum packet size specified in the Bluetooth 4.0 specification. With this change alone transfer speeds may become fast enough for much higher throughput and expand MANET capabilities.

This paper and implemented applications have demonstrated that Bluetooth Low Energy provides a suitable peer to peer transport method for small amounts of arbitrary data between modern smartphones. With the planned capabilities of Bluetooth Low Energy in future specifications and the rising ubiquity of consumer devices offering BLE capabilities, reliable and commonplace peer to peer networking is on the horizon.

Appendix A – Testing Applications



MotoE2(4G-LTE)

UUID: 71270505-829A-EFAA-7DC0-87BFD6F74B40

Connected

ADVERTISEMENT DATA

Yes
Device Is Connectable

86130247-E942-4FE5-AA46-E30768A0C1B0
Service UUIDs

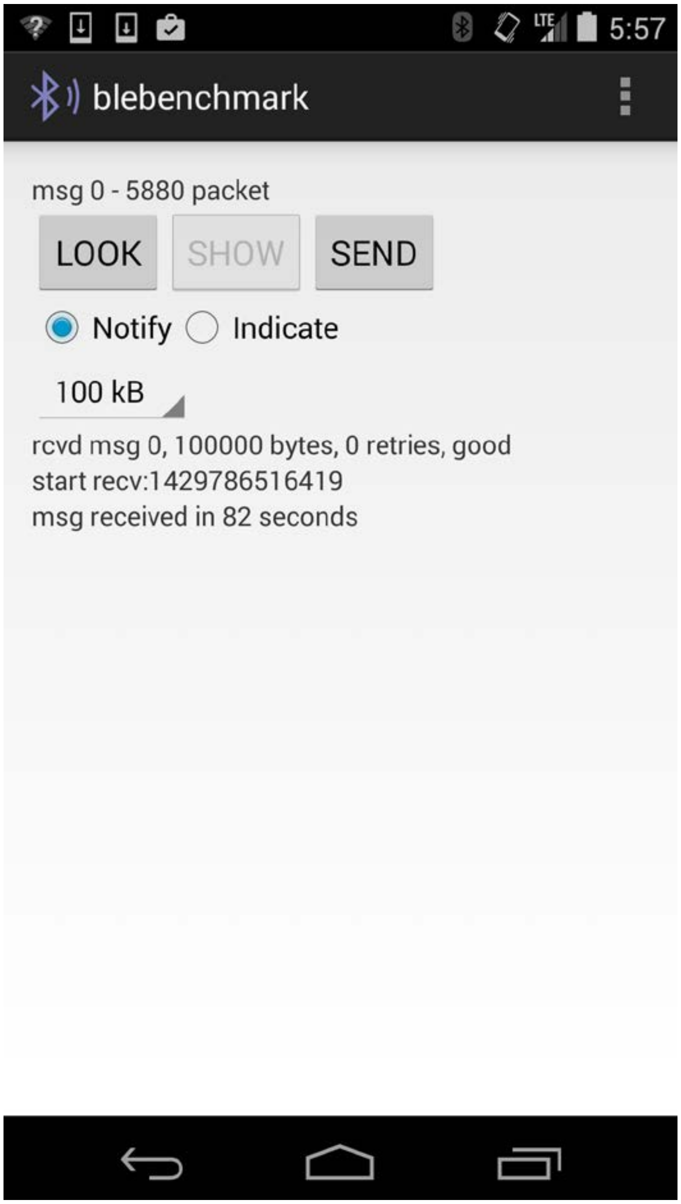
UUID: 86130247-E942-4FE5-AA46-E30768A0C1B0

0x0989
Properties: Read

0x0979
Properties: Read Write

0x0999
Properties: Write

Figure A.1 – Kevo Android Service and Characteristics



msg 0 - 5880 packet

● Notify ○ Indicate

100 kB

rcvd msg 0, 100000 bytes, 0 retries, good
start recv:1429786516419
msg received in 82 seconds

Figure A.2 - Benchmark App on Nexus 5 (4.4.4)

Bibliography

[1] Sasso, B. (2012). “FCC: hurricane Sandy took out 25 percent of cell towers.” The Hill, p. 10.

[2] Jurgensen, J. (2015). “Concert Crowds Flounder in Digital Dead Zones. WSJ.” Retrieved 4 May 2015, from http://www.wsj.com/articles/demand-for-wireless-signals-pressures-concert-promoters-1417722615

[3] Landau, S. (2013). “Making Sense from Snowden: What's Significant in the NSA Surveillance Revelations.” Security Privacy, IEEE, 11(4), 54-63.

[4] Cronan, B. (2014). “Hong Kong protestors use FireChat to text without cell service.” The Christian Science Monitor, 23.

[5] Rai, S. (2014). “Google introduces phone for emerging markets.” The New York Times, 11.

[6] Cooper, D. (2015). “Google's Android One program will set minimum standards for bargain-basement smartphones.” Engadget. Retrieved 4 May 2015, from http://www.engadget.com/2014/06/25/google-android-one/

[7] Android Connectivity APIs,. (2015). “Connectivity | Android Developers.” Retrieved 4 May 2015, from http://developer.android.com/guide/topics/connectivity/index.html

[8] iOS Developer Library,. (2015). “Multipeer Connectivity Framework Reference.” Retrieved 4 May 2015, from https://developer.apple.com/library/prerelease/ios/documentation/MultipeerConnectivity/Reference/MultipeerConnectivityFramework/index.html

[9] Apple iOS Support,. (2015). “iOS Supported Bluetooth profiles.” Retrieved 4 May 2015, from https://support.apple.com/en-us/HT204387

[10] International Data Corporation,. (2014). “Smartphone Outlook Remains Strong for 2014, Up 23.8%, Despite Slowing Growth in Mature Markets, According to IDC.” Retrieved from http://www.idc.com/getdoc.jsp?containerId=prUS25058714

[11] Milanesi, C. (2015). “Apple iOS leads US OS share for the first time since Q4 2012 - Global site - Kantar Worldpanel.” Kantarworldpanel.com. Retrieved 4 May 2015, from http://www.kantarworldpanel.com/global/News/Apple-iOS-leads-US-OS-share-for-the-first-time-since-Q4-2012

[12] Android Developer Dashboard, (2015). “Platform Versions.” Retrieved 4 May 2015, from http://developer.android.com/about/dashboards/index.html

[13] Ryan, M. (2013). “Bluetooth: With Low Energy Comes Low Security.” Presented as part of the 7th USENIX Workshop on Offensive Technologies. Berkeley, CA: USENIX. Retrieved from https://www.usenix.org/conference/woot13/workshop-program/presentation/Ryan

[14] Galeev, M. (2011). “Bluetooth 4.0: An introduction to Bluetooth Low Energy-Part II.” EE Times. Retrieved 4 May 2015, from http://www.eetimes.com/document.asp?doc_id=1278966

[15] Bluetooth SIG,. (2015). “Bluetooth Smart (Low Energy) Technology.” Retrieved 4 May 2015, from https://developer.bluetooth.org/TechnologyOverview/Pages/BLE.aspx

[16] Bluetooth SIG,. (2015). "Bluetooth Developer Portal - Heart Rate Profile.” Retrieved 4 May 2015, from https://developer.bluetooth.org/gatt/profiles/Pages/ProfileViewer.aspx?u=org.bluetooth.profile.heart_rate.xml

[17] Bluetooth SIG,. (2015). “GATT Specifications, Profiles.” Retrieved 4 May 2015, from https://developer.bluetooth.org/gatt/profiles/Pages/ProfilesHome.aspx

[18] Bluetooth SIG,. (2010). “Bluetooth Core Specification 4.0”, p. 216.

[19] OpenGarden,. (2015). “Firechat FAQ.” Retrieved 4 May 2015, from https://opengarden.com/faq#faq-general-F6

[20] Unikey. (2015). “Let There Be Android for Kevo.” Retrieved 4 May 2015, from http://www.unikey.com/blog/2015/04/26/let-there-be-android-for-kevo/

[21] Bluetooth SIG,. (2015). “Bluetooth Interoperability and Profiles.” Retrieved 4 May 2015, from https://developer.bluetooth.org/DevelopmentResources/Pages/Custom-Profile-Development.aspx

[22] Kwikset,. (2015). “Kevo Smart Lock - Battery Life.” Retrieved 4 May 2015, from http://www.kwikset.com/kevo/default.aspx#.VR7iuvkc7Gw

[23] Bluetooth SIG,. (2015). “The Low Energy Technology Behing Bluetooth Smart.” Retrieved 4 May 2015, from http://www.bluetooth.com/Pages/low-energy-tech-info.aspx

[24] Bronzi, W., Frank, R., Castignani, G., & Engel, T. (2014). “Bluetooth low energy for inter-vehicular communications.” In IEEE (pp. 215-221).

[25] International Data Corporation,. (2015). “Android and iOS Squeeze the Competition, Swelling to 96.3% of the Smartphone Operating System Market for Both 4Q14 and CY14, According to IDC.” Retrieved from http://www.idc.com/getdoc.jsp?containerId=prUS25450615

[26] Android Developer Preview, (2014). “Issue 1570 - android-developer-preview - BLE advertise mode not working.” Retrieved 4 May 2015, from https://code.google.com/p/android-developer-preview/issues/detail?id=1570#c52

[27] Sandoval, R., & Schoolfield, M. (2014). “SubContext.”

[28] “Marsh Deploys IBeacon System.” (2015). *MMR*, *32*(2), 30.

[29] Official Google Blog, (2015). “Android: Be together. Not the same.” Retrieved 4 May 2015, from http://googleblog.blogspot.com/2014/10/android-be-together-not-same.html

[30] Wright, J. (2007). “Dispelling Common Bluetooth Misconceptions.” SANS. Retrieved 4 May 2015, from http://www.sans.edu/research/security-laboratory/article/bluetooth

[31] Bluetooth Developer Portal,. (2015). “DescriptorViewer”. Retrieved 4 May 2015, from https://developer.bluetooth.org/gatt/descriptors/Pages/DescriptorViewer.aspx?u=org.bluetooth.descriptor.gatt.client_characteristic_configuration.xml

[32] Laiacano, A. (2015). “Testing Bluetooth Low Energy Devices.” Punch Through Design. Retrieved 22 April 2015, from http://blog.punchthrough.com/post/46285311872/testing-bluetooth-low-energy-devices

[33] Gomez, C., Demirkol, I., & Paradells, J. (2011). “Modeling the Maximum Throughput of Bluetooth Low Energy in an Error-Prone Link.” IEEE Communications Letters, 12(11), 1187-1189.

[34] Gomez, C., Oller, J., & Paradells, J. (2012). “Overview and evaluation of bluetooth low energy: an emerging low-power wireless technology.” Sensors (Basel, Switzerland), 12(9), 11734-11753.

[35] Android Developer Reference, (2015). “javax.crypto.” Retrieved 4 May 2015, from http://developer.android.com/reference/javax/crypto/package-summary.html

[36] Daemen, J., & Rijmen, V. (2002). “The design of Rijndael: AES--the advanced encryption standard.”

[37] Shamir, A. (1979). “How to Share a Secret.” Commun. ACM, 22(11), 612--613. Retrieved from http://doi.acm.org/10.1145/359168.359176

[38] Belton, C. (2012). “UK admits officials used fake rock to spy on Russia.” The Financial Times, 4.

[39] Tiemens, T. (2015). “Shamir's Secret Share in Java.” GitHub. Retrieved 4 May 2015, from https://github.com/timtiemens/secretshare

[40] Parakh, A., & Kak, S. (2011). “Space efficient secret sharing for implicit data security.” Information Sciences, 181(2), 335-341.

[41] Schoolfield, M. (2015).” Source code for SimpBLE and Benchmarking Application.” GitHub. Retrieved 4 May 2015, from https://github.com/ludwigmace/blebenchmark