Bluetooth Automation using Host Card Emulation

A Dissertation submitted in the partial fulfillment for the award of

MASTER OF TECHNOLOGY

IN

SOFTWARE ENGINEERING

By

Amit Mani Tewari
Roll no. 2k12/SWT/03

Under the Essential Guidance of

Mrs. DivyaSikha Setia
Assistant Professor

Department of Computer Engineering

Department of Computer Engineering
Delhi Technological University
New Delhi
2012-2015

DECLARATION

I hereby declare that the thesis entitled “**Bluetooth Automation using Host Card Emulation”.** Which is being submitted to the **Delhi Technological University**, in partial fulfillment of the requirements for the award of degree of **Master of Technology in Software Engineering** is an authentic work carried out by me. The material contained in this thesis has not been submitted to any university or institution for the award of any degree.

Amit Mani Tewari
Department of Computer Engineering
Delhi Technological University,
Delhi.

CERTIFICATE

DELHI TECHNOLOGICAL UNIVERSITY

(Govt. of National Capital Territory of Delhi)

BAWANA ROAD, DELHI-110042

Date: ____________________

This is to certify that the thesis entitled **"Bluetooth Automation using Host Card Emulation"**submitted by **Amit Mani Tewari(Roll Number: 2K12/SWT/03),** in partial fulfillment of the requirements for the award of degree of Master of Technology in Software Engineering, is an authentic work carried out by her under my guidance. The content embodied in this thesis has not been submitted by her earlier to any institution or organization for any degree or diploma to the best of my knowledge and belief.

Project Guide

Mrs. DivyaSikha Setia

Assistant Professor

Department of Computer Engineering

Delhi Technological University, Delhi-110042

ACKNOWLEDGEMENT

I take this opportunity to express my deepest gratitude and appreciation to all those who have helped me directly or indirectly towards the successful completion of this thesis.

Foremost, I would like to express my sincere gratitude to my mentor guide **Mrs. DivyaSikha Setia**, **Astt.Professor, Department of Computer Engineering**, **Delhi Technological University, Delhi** whose benevolent guidance, constant support, encouragement and valuable suggestions throughout the course of my work helped me successfully complete this thesis. Without her continuous support and interest, this thesis would not have been the same as presented here.

Besides my guide, I would like to thank the entire teaching and non-teaching staff in the Department of Computer Science, DTU for all their help during my course of work.

AMIT TEWARI

ABSTRACT

A health management system will be efficient if medical records of patient maintained smartly so that it can be accessed faster, easily available and portable. As with technology advancement there are revolutionary changes in area of handheld devices and wireless communication, this advancement in technology can be utilized to develop efficient and smart health care system. Smart cards can be used to store health data securely and it can be embedded in handheld device i.e. mobile phone, so that it is easily portable. Wireless technology i.e. Bluetooth, NFC can be used transfer health records between devices; this will avoid dependency on permanent and fixed storage and avoid traditional cumbersome paper reports.

Major Goal of this thesis is to use wireless communication to HEALTHCARD to make it more efficient.

- Bluetooth automation has been used for communicating health card devices i.e. patient handled device (mobile) with smart card device to transfer patient Health records.
- Bluetooth communication has been automated to involve minimal user interaction i.e. No PIN exchange or confirmation, No Authentication or Authorization popup is shown still able to SEND and GET data over Bluetooth.
- Secure Health Card has been used to Maintain Patient Health record on handheld devices.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter-1

INTRODUCTION

1.1 General

With advancement in technology, Medical field has been organized to have advance database which stores medical information efficiently, still an issue of Infrastructure to access database should be there. To overcome this and make medical system still efficient, idea of using portable setup to access medical records has been purposed.

To use this system main requirements were:

- Having efficient storage device which is portable
- Efficient communication System, using which data can be accessed properly without much user involvement.

To store records efficiently HEALTH CARD can be used with will store large size data and are secured. To address accessibility issue's advance wireless techniques can be utilized i.e.NFC, Bluetooth. Wireless communication should be such that data can be accessed easily still security should be ensured.

Although NFC communication is easy to use and secure still main drawback of it is its low bandwidth due to which only small amount of data can be transferred using NFC.
Bluetooth is another technology which is having better range then NFC (up to 100mtrs) and having high bandwidth of transferring data. Blue low energy (BT 4.0) is advance BT version where user don't have to initiate pairing procedure and its range is better the NFC but problem with BT low energy that is designed with purpose of transferring small chunks of data so that Battery power consumption should be minimum. As in health care system sizeable amount of health records needs to be transferred hence BT low energy also not perfectly fits into purpose.

1.2Health Care Elements

Smart Health Cards-This can be used efficiently to store Medical records of large size and having wireless communication support capability (i.e. NFC, Bluetooth etc) to communicate with remote device.

Host card emulation (HCE)- is the software architecture that provides exact virtual representation of various electronic identity (access, transit and banking) cards using only software. Prior to the HCE architecture, NFC transactions were mainly carried out using secure elements, which were integrated into carrier-issued SIM cards.HCE enables mobile applications running on supported operating systems with the ability to offer payment card and access card solutions independently of third parties while leveraging cryptographic processes traditionally used by hardware-based secure elements without the need for a physical secure element. This technology enables the merchants to offer payment cards solutions more easily through mobile closed-loop contactless payment solutions, offers real-time distribution of payment cards and, more tactically, allows for an easy deployment scenario that does not require changes t**o** the software inside payment terminals**.**

Bluetooth-Wireless LANs can be designed in many different ways depending on their applications; one kind is a wireless *ad-hoc* network. An ad-hoc network is a peer-to-peer network set up temporarily to meet some immediate need. In contrast to the majority of radio systems in use today, it doesn’t have any centralized server. However, the Bluetooth system is the first commercial ad hoc radio system envisioned to be used on a large scale and widely available to the public.

1.3Problem Proposed solution

To develop an efficient Health Care System where Medical information can be maintained smartly without requiring tedious Infrastructure and Data can be accessed in real time and easily portable without having resources and technology to access bigger databases. This thesis purposes use of classical Bluetooth automation as a wireless communication medium to access data between portable Health card devices.

This work AIMS in providing Bluetooth automation support for communicating Hand held device with Smart card to exchange information at the same time ensuring no user interaction is done i.e. PIN exchange or confirmation, No Authentication or Authorization popup is shown still able to SEND and GET data over Bluetooth.

Smart Card are used to store Health records securely, these smart card supports wireless communication techniques i.e. HCE. Host Card Emulation technique helps in transferring details wirelessly and securely which are used for further communication Over Bluetooth.

Chapter-2

LITERATURE SURVEY

2.1 BLUETOOTH RADIO SYSTEM ARCHITECTURE

Bluetooth devices operate at 2.4GHz, in the globally available, license-free, ISM band. That is the bandwidth reserved for general use by Industrial, Scientific and Medical applications worldwide. Since this radio band is free to be used by any radio transmitter as long as it satisfies the regulations, the intensity and the nature of interference can't be predicted. Therefore, the interference immunity is very important issue for Bluetooth. Generally, interference immunity can be obtained by interference suppression or avoidance. Suppression can be obtained by coding or direct-sequence spreading, but the dynamic range of interfering signals in ad hoc networks can be huge, so practically attained coding and processing gains are usually inadequate. Avoidance in frequency is more practical. Since ISM band provides about 80MHz of bandwidth and all radio systems are band limited, there is a high probability that a part of the spectrum can be found without a strong interference.

Considering all this, FH-CDMA (Frequency Hopping - Code Division Multiple Access) technique has been chosen to implement the multiple access scheme for the Bluetooth. It combines a number of properties, which make it the best choice for an ad hoc radio system. It fulfills the spreading requirements set in the ISM band, i.e. on average the signal can be spread over a large frequency range, but instantaneously only a small part of the bandwidth is occupied, avoiding most of potential interference. It also doesn't require neither strict time synchronization (like TDMA), nor coordinated power control (like DS-CDMA). In the 2.45GHz ISM band, a set of 79 hop carriers has been defined, at 1MHz spacing. A nominal hop dwell time is 625 us. Full-duplex communication is achieved by applying time-division duplex (TDD), and since transmission and reception take place at different time slots, they also take place at different hop carriers. A large number of pseudo-random hopping sequences have been defined, and the particular sequence is determined by the unit that controls the FH channel. That unit is usually called the *master* and it also defines timing parameters during the certain session. All other devices involved in the session, the *slaves*, have to adjust their spreading sequences and clocks to the master's.

An FH Bluetooth channel is associated with the piconet. As mentioned earlier, the master unit defines the piconet channel by providing the hop sequence and the hop phase. All other units participating in the piconet are slaves. However, since the Bluetooth is based on peer communications, the master/slave role is only attributed to a unit for the duration of the piconet. When the piconet is cancelled, the master and slaves roles are canceled too. In addition to defining the piconet, the master also controls the traffic on the piconet and takes care of access control. The time slots are alternatively used for master and slaves transmission. In order to prevent collisions on the channel due to multiple slave transmissions, the master applies a polling technique, for each slave-to-master slot the master decides which slave is allowed to transmit.

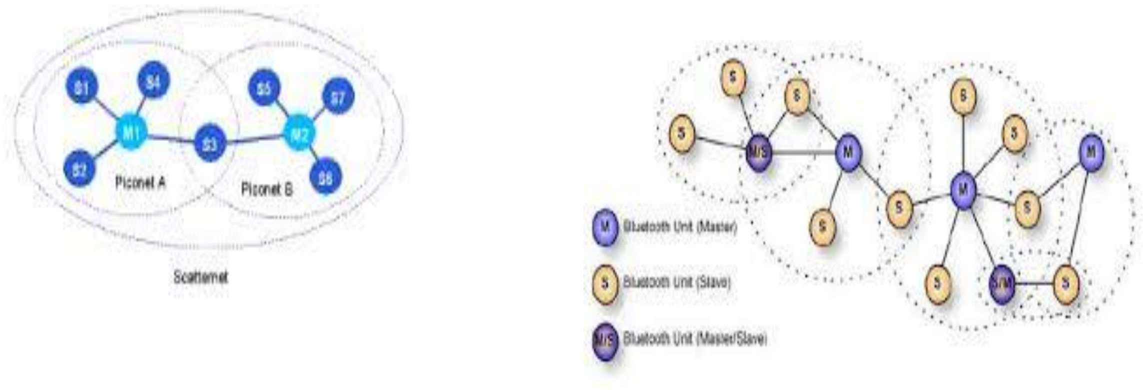


FIGURE1: Bluetooth Scatter-net[4]

2.2BLUETOOTH EVOLUTION

Bluetooth v1.0 and v1.0B

Versions 1.0 and 1.0B had many problems, and manufacturers had difficulty making their products interoperable. Versions 1.0 and 1.0B also included mandatory Bluetooth hardware device address (BD_ADDR) transmission in the Connecting process (rendering anonymity impossible at the protocol level), which was a major setback for certain services.

Bluetooth v1.1

- Ratified as IEEE Standard 802.15.1-2002
- Many errors found in the 1.0B specifications were fixed.
- Added possibility of non-encrypted channels.
- Received Signal Strength Indicator (RSSI).

Bluetooth v1.2

- This version is backward compatible with 1.1 and the major enhancements include the following:
- Faster Connection and Discovery
- Adaptive frequency-hopping spread spectrum *(AFH)*, which improves resistance to radio frequency interference by avoiding the use of crowded frequencies in the hopping sequence.
- Higher transmission speeds in practice, up to 721 kbit/s, than in v1.1.
- Extended Synchronous Connections (eSCO), which improve voice quality of audio links by allowing retransmissions of corrupted packets, and may optionally increase audio latency to provide better concurrent data transfer.
- Host Controller Interface (HCI) operation with three-wire UART.
- Ratified as IEEE Standard 802.15.1-2005
- Introduced Flow Control and Retransmission Modes for L2CAP.

Bluetooth v2.0 + EDR

This version of the Bluetooth Core Specification was released in 2004 and is backward compatible with the previous version 1.2. The main difference is the introduction of an Enhanced Data Rate (EDR) for faster data transfer. The nominal rate of EDR is about 3 Mbit/s, although the practical data transfer rate is 2.1 Mbit/s. EDR uses a combination of GFSK and Phase Shift Keyingmodulation (PSK) with two variants, π/4-DQPSK and 8DPSK. EDR can provide a lower power consumption through a reduced duty cycle.
The specification is published as "Bluetooth v2.0 + EDR" which implies that EDR is an optional feature. Aside from EDR, there are other minor improvements to the 2.0 specification, and products may claim compliance to "Bluetooth v2.0″ without supporting the higher data rate.

Bluetooth v2.1 + EDR

Bluetooth Core Specification Version 2.1 + EDR is fully backward compatible with 1.2, and was adopted by the Bluetooth SIG on July 26, 2007.
The headline feature of 2.1 is secure simple pairing (SSP): this improves the pairing experience for Bluetooth devices, while increasing the use and strength of security.
2.1 allows various other improvements, including "Extended inquiry response" (EIR),

Bluetooth v3.0 + HS

Version 3.0 + HS of the Bluetooth Core Specification was adopted by the Bluetooth SIG on April 21, 2009. Bluetooth 3.0+HS provide theoretical data transfer speeds of up to 24 Mbit/s**,**though not over the Bluetooth link itself. Instead, the Bluetooth link is used for negotiation and establishment, and the high data rate traffic is carried over a collocated 802.11 link.
The main new feature is AMP (Alternate MAC/PHY), the addition of 802.11 as a high speed transport. The High-Speed part of the specification is not mandatory, and hence only devices sporting the "+HS" will actually support the Bluetooth over 802.11 high-speed data transfer. A Bluetooth 3.0 device without the "+HS" suffix will not support High Speed, and needs to only support a feature introduced in Core Specification Version 3.0 or earlier Core Specification.

Bluetooth 4.0 (BLE) Bluetooth low energy

- This version's development
- Ease of implementation and multi-vendor interoperability
- Ultra-low peak, average and idle mode power consumption
- Low cost of integration
- Power handling
- Resistance to interference

Bluetooth 4.1 (Still to be commercialized)

1. Coexistence

Bluetooth and 4G (LTE) famously don’t get on: their signals interfere degrading one another’s performance and draining battery life. Bluetooth 4.1 eliminates this by coordinating its radio with 4G automatically so there is no overlap and both can perform at their maximum potential. Given most phones will come with 4G next year this is a vital improvement.

2. Smart connectivity

Rather than carry a fixed timeout period, Bluetooth 4.1 will allow manufacturers to specify the reconnection timeout intervals for their devices. This means devices can better manage their power and that of the device they are paired to by automatically powering up and down based on a bespoke power plan.

3. Improved Data Transfer

Bluetooth 4.1 devices can act as both hub and end point simultaneously. This is hugely significant because it allows the host device to be cut out of the equation and for peripherals to communicate independently.For example, whereas previously a smartwatch would need to talk to your phone to get data from a heart monitor, now the smartwatch and heart monitor can talk directly saving your phone’s battery and then upload their compiled results directly to your phone.

2.3BLUETOOTH STACK

The Bluetooth protocol stack is defined as a series of layers, though there are some features which cross several layers. A Bluetooth device can be made up of two parts: a host implementing the higher layers of the protocol stack, and a module implementing the lower layers. This separation of the layers can be useful for several reasons. For example, hosts such as PCs have spare capacity to handle higher layers, allowing the Bluetooth device to have less memory and a less powerful processor, which leads to cost reduction. Also, the host device can sleep and be awoken by an incoming Bluetooth connection. Of course, an interface is needed between the higher and lower layers, and for that purpose the Bluetooth defines the Host Controller Interface (HCI). But for some small and simple systems, it is still possible to have all layers of the protocol stack run on one processor. An example of such a system is a headset.

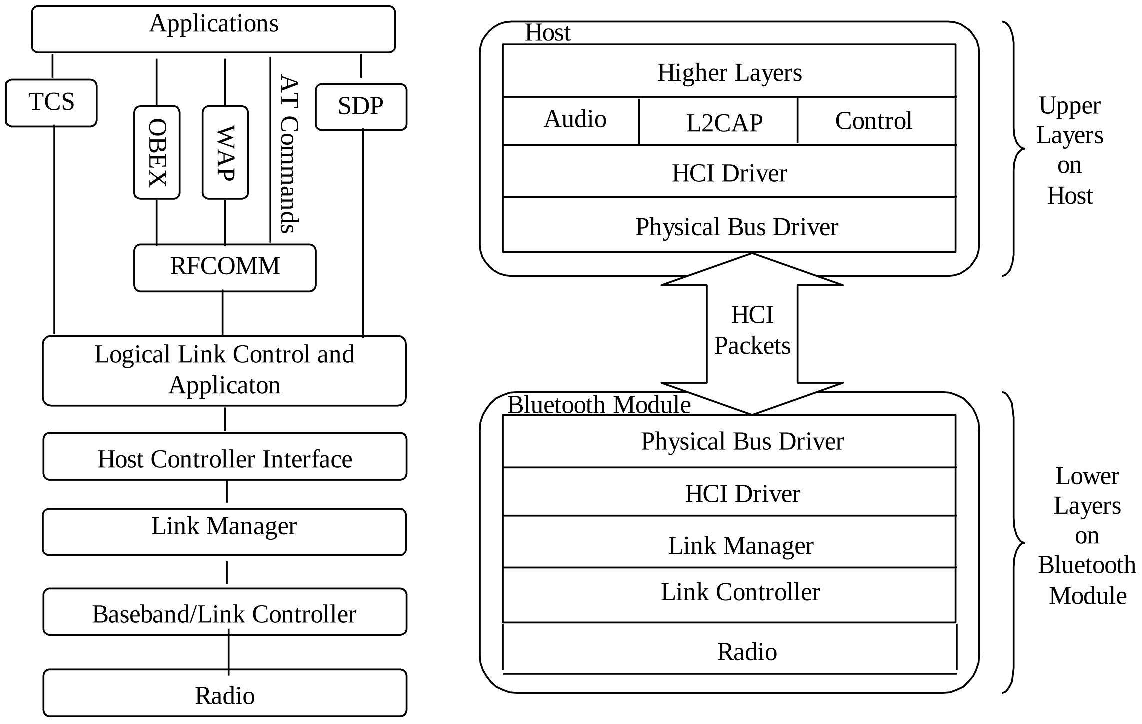


Figure 2: Bluetooth Protocol Stack [4]

BLUETOOTHCONTROLLER:

Baseband- There are two basic types of physical links that can be established between a master and a slave:

- Synchronous Connection Oriented (SCO)
- Asynchronous Connection-Less (ACL)

The Link Controller - The link control layer is responsible for managing device discoverability, establishing connections and maintaining them. In Bluetooth, three elements have been defined to support connection establishment: scan, page and inquiry.

The Link Manager - The host drives a Bluetooth device through Host Controller Interface (HCI) commands, but it is the link manager that translates those commands into operations at the baseband level. Its main functions are to control piconet management (establishing and destruction of the links and role change), link configuration, and security and QoS functions.

BLUETOOTH HOST:

Logical Link Control and Adaptation Protocol (L2CAP) - Logical Link Control and Adaptation Protocol takes data from higher layers of the Bluetooth stack and from applications and sends them over the lower layers of the stack.. The major functions of the L2CAP are:

- Multiplexing between different higher layer protocols to allow several higher layer links to share a single ACL connection.
- Segmentation and reassembly to allow transfer of larger packets than lower layers support.

RFCOMM - RFCOMM is a simple, reliable transport protocol that provides emulation of the serial cable line settings and status of an RS-232 serial port. It provides connections to multiple devices by relying on L2CAP to handle multiplexing over single connection.

The Service Discovery Protocol - One of the most important members of the Bluetooth protocol stack is Service Discovery Protocol (SDP). It provides a means for an SDP client to access information about services offered by SDP servers. An SDP server is any Bluetooth device which offers services to other Bluetooth devices.

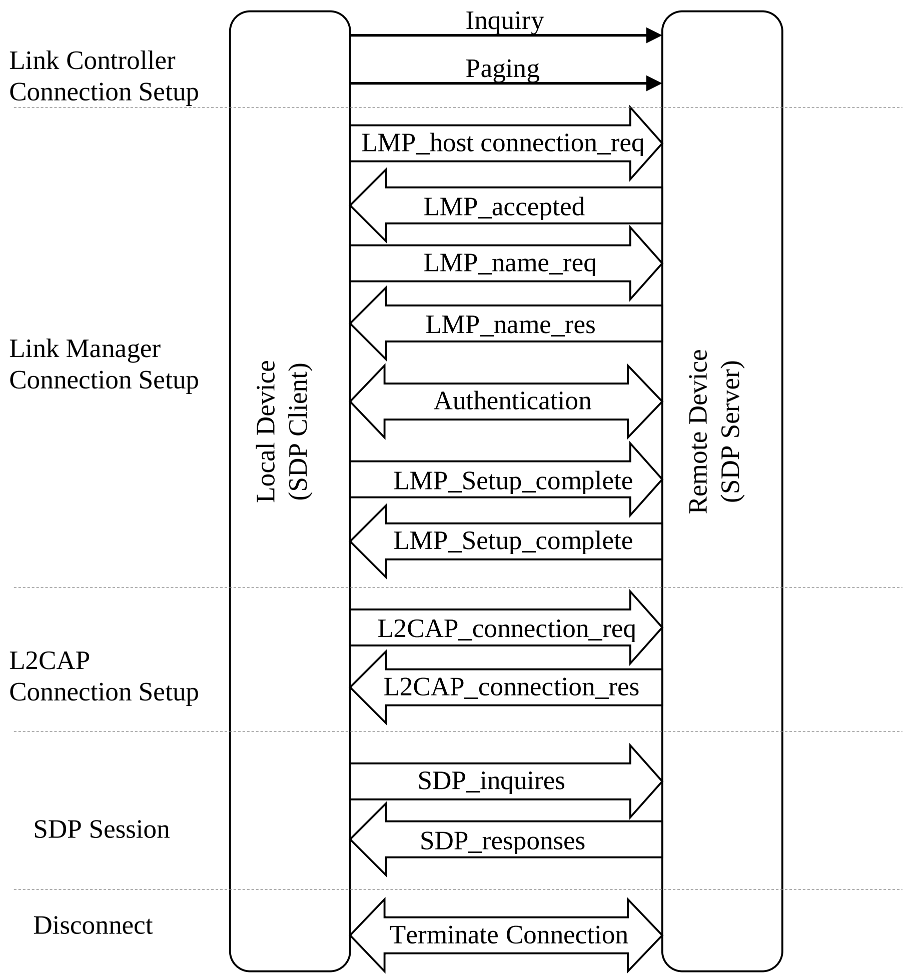


Figure 3: Stages in setting up an SDP session[6]

2.4 PROTOCOLS AND PROFILES

BLUETOOTH PROTOCOLS:

As mentioned at the begin, one of the most important characteristics of the Bluetooth specification is that it should allow devices from lots of different manufacturers to work with one another. For that reason, Bluetooth is designed in such a way to allow many different protocols to be run on top of it. Some of these protocols are:

The Wireless Access Protocol (WAP)

Object Exchange Protocol (OBEX)

The Telephony Control Protocol

Audio / Video Distribution Transport Protocol (AVDTP).

BLUETOOTH PROFILES

Profiles are associated with applications. The profiles specify which protocol elements are mandatory in certain applications. This concept prevents devices with little memory and processing power implementing the entire Bluetooth stack when they only require a small fraction of it. Simple devices like a headset or mouse can thus be implemented with a strongly reduced protocol stack.

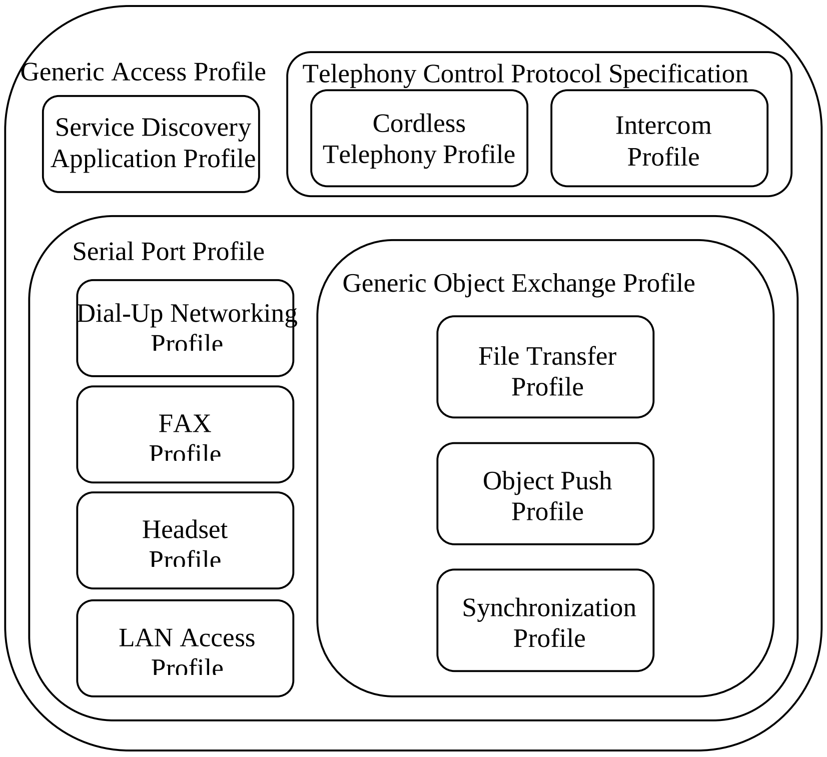


Figure 4: Bluetooth profiles [6]

Bluetooth Profile Example:

GAP -Generic Access Profile

SPP- Serial Port Profile)

DUN-Dial-up Networking,

FTP- File Transfer Profile

OPP-Object PushProfile

A2DP-Advanced Audio Distribution Profile

HFP-Hands Free Profile

MAP-Message Access Profile

2.5 CLASSICBLUETOOTH SECURITY

Trusted Device: The device has been previously authenticated, a link key is stored and the device is marked as "trusted" in the Device Database.

Untrusted Device: The device has been previously authenticated, a link key is stored but the device is not marked as "trusted" in the Device Database

Unknown Device: No security information is available for this device.

SECURITY MODES:

Mode 1: **No Security**

Only security at this level is by the nature of the connection: data-hopping and short-distance.All Bluetooth devices employ “data-hopping”, which entails skipping around the radio band up to 1600 times per second, at 1MHz intervals (79 different frequencies).Most connections are less than 10 meters, so there is a limit as to eavesdropping possibilities.

Mode 2: Service Level Security

Trusted devices have unrestricted access to all services, fixed relationship to other devices. UnTrusted devices generally have no permanent relationship and services that it has access to are limited. Unfortunately, all services on a device are given the same security policy, other than application layer add-ons.

Services can have one of 3 security levels:

Level 3: Requires Authentication and Authorization. PIN number must be entered.

Level 2: Authentication only, fixed PIN ok.

Level 1: Open to all devices, the default level.

Mode3: Link Level Security

Security is implemented by symmetric keys in a challenge-response system.

Security implementations in Bluetooth units are all the same, and are publicly available: Critical ingredients: PIN, BD_ADDR, RAND, Link and Encryption Keys.

Chapter-3

DATATRANSFER OVER BLUETOOTH

INTRODUCTION

As already explained in section Chapter 2 above there are multiple Bluetooth profile’s designed for specific target and to achieve a specific functionality. Each profile has its own set of rules and specification that has to be followed by all profile implementers to achieve that functionality. This ensures interoperability between different operators. Based on BT SIG supported profiles, below set of profiles can be used for data transfer. Each profile has specific data types or format which it can transfer; some operates on bits, while other on files and PIMS object etc.

Protocols followed by each of below profiles are different. Some profiles are designed to operate on large size of data while others can transfer fix size of bytes only.

Each profile’s has its own set of authentication or encryption technique. Each of these profiles has been discussed in details in below section.

- OBJECT PUSH PROFILE
- FILE TRANSFER PROFILE
- SERIAL PORT PROFILE
- BT LOW ENRGY (BT 4.0)

3.1OBJECT PUSH PROFILE

Introduction

The Object Push Profile (OPP) defines the requirements for the protocols and procedures that shall be used by the applications providing the Object Push usage model. This profile makes use of the Generic Object Exchange Profile (GOEP) [10], to define the interoperability requirements for the protocols needed by applications. Common devices using these models are notebook PCs, PDAs, and mobile phones.

The scenarios covered by this profile are the following:

• Use of a Bluetooth device to push an object to the inbox of another Bluetooth device. The object can for example be a business card or an appointment.

• Use of a Bluetooth device to pull a business card from another Bluetooth device.

• Use of a Bluetooth device to exchange business cards with another Bluetooth device. Exchange is defined as a push of a business card followed by a pull of a business card.

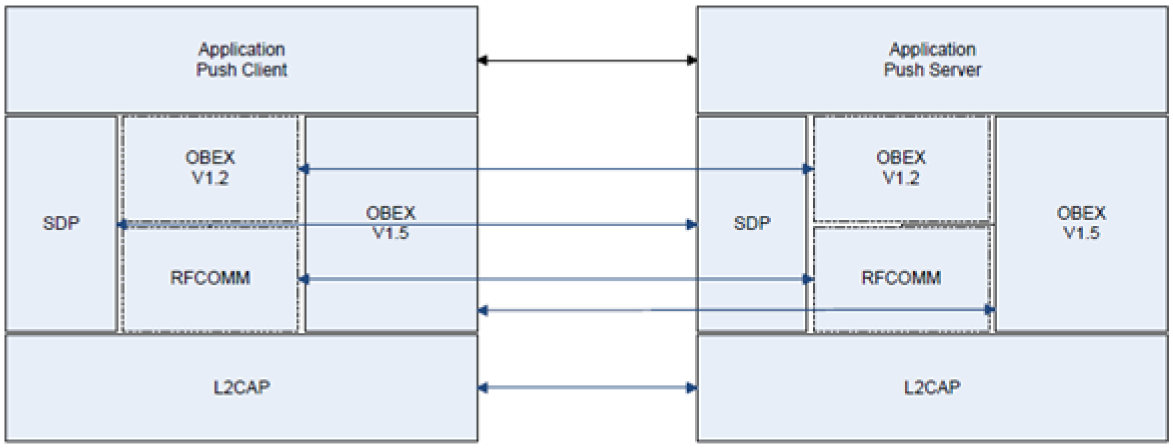


FIGURE 5: OBEX Protocol Model[8]

Configurations and Roles

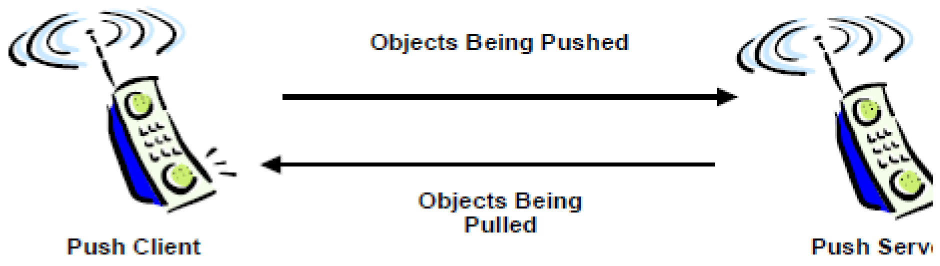


FIGURE-6: OPP PUSH and PULL OPERATIONS[8]

The following roles are defined for this profile:

Push Server – This device provides an object exchange server. In addition to the interoperability requirements defined in this profile, the Push Server shall comply with the interoperability requirements for the server of the GOEP if not defined in the contrary.

Push Client – This device pushes and pulls objects to and from the Push Server. In addition to the interoperability requirements defined in this profile, the Push client shall also comply with the interoperability requirements for the client of the GOEP, if not defined to the contrary.

Profile Scenarios:

The scenarios covered by this profile are:

• Use of a Push Client to push an object to a Push Server. The object can, for example, be a business card or an appointment.

• Use of a Push Client to pull a business card from a Push Server.

• Use of a Push Client to exchange business cards with a Push Server.

The push operation described in this profile pushes objects from the Push Client to the inbox of the Push Server.

3.2 File Transfer Profile

Introduction

This profile defines the requirements for the protocols and procedures that shall be used by the applications providing the file transfer usage model. The file transfer usage model makes use of the underlying Generic Object Exchange Profile (GOEP) to define the interoperability requirements for the protocols needed by applications. Typical scenarios covered by this profile involving a Bluetooth device browsing , transferring and manipulating objects on/with another Bluetooth device.

Roles/Configurations

The following roles are defined for this profile:

Server – The Server device is the target remote Bluetooth device that provides an object exchange server and folder browsing capability using the OBEX Folder Listing format. In addition to the interoperability requirements defined in this profile, the Server must comply with the interoperability requirements for the Server of the GOEP if not defined in the contrary.

Client – The Client device initiates the operation, which pushes and pulls objects to and from the Server. In addition to the interoperability requirements defined in this profile, the Client must also comply with the interoperability requirements for the Client of the GOEP if not defined in the contrary.

Profile Scenarios

The scenarios covered by this profile are the following:

Usage of the Client to browse the object store of the Server. Clients are required to pull and understand Folder Listing Objects. Servers are required to respond to requests for Folder Listing Objects. Servers are required to have a root folder. Servers are not required to have a folder hierarchy below the root folder.

Usage of the Client to transfer objects to and from the Server. The transfer of objects includes folders and files. Clients must support the ability to push or pull files from the Server. Clients are not required to push or pull folders. Servers are required to support file push, pull, or both. Servers are allowed to have read-only folders and files, which means they can restrict object pushes. Thus, Servers are not required to support folder push or pull.

Usage of the Client to create folders and delete objects *(folders and files)* on the Server. Clients are not required to support folder/file deletion or folder creation. Servers are allowed to support read-only folders and files, which means they can restrict folder/file deletion and creation. A device adhering to this profile must support Client capability, Server capability or both.

File Transfer Applications provide the following functions**.**

- Navigate Folders
- Pull Object
- Push Object
- Delete Object
- Create Folder
-

When the user selects the Select Server function, an inquiry procedure will be performed to produce a list of available devices in the vicinity.

3.3SERIAL PORT PROFILE

Introduction

The Serial Port Profile defines the protocols and procedures that shall be used by devices using Bluetooth for RS232 (or similar) serial cable emulation. The scenario covered by this profile deals with legacy applications using Bluetooth as a cable replacement, through a virtual serial port abstraction.

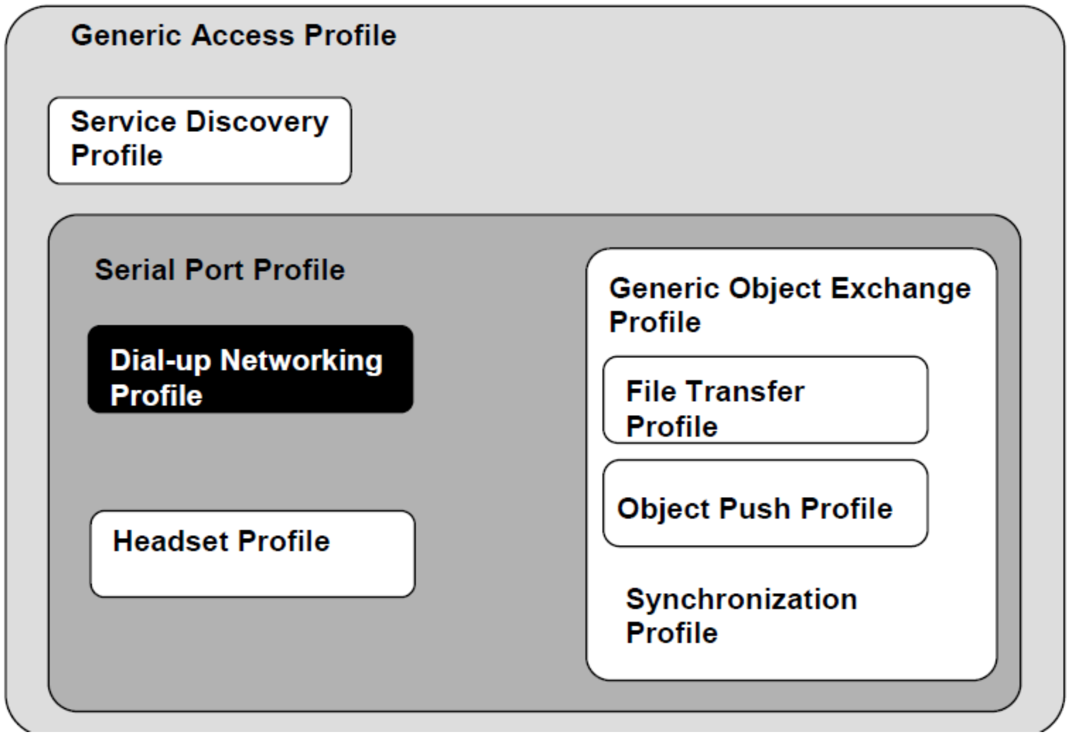


FIGURE 7: SPP PROFILE STRUCTURE [8]

Profile Stack:

Figure Below shows the protocols and entities used in this profile:

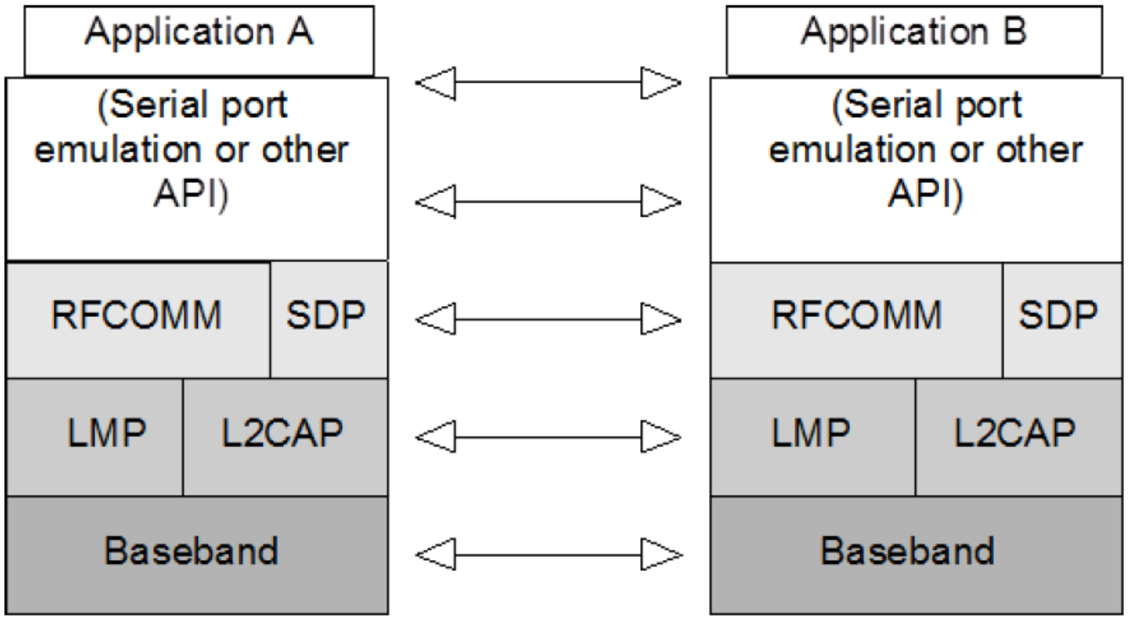


FIGURE 8: SPP PROFILE STACK [8]

Roles and Configuration

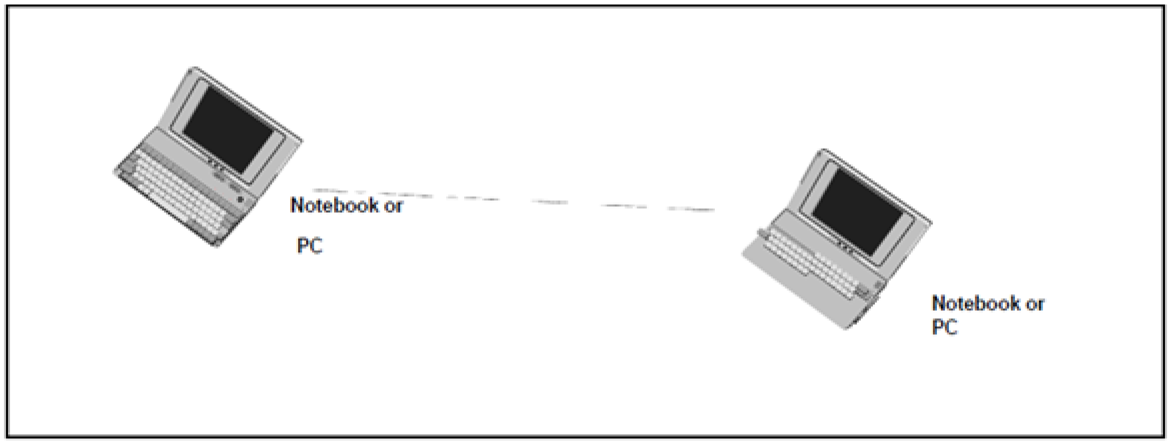


FIGURE 9: SPP PROFILE ROLE [8]

Device A (DevA) – This is the device that takes initiative to form a connection to another device.
Device B (DevB) – This is the device that waits for another device to take initiative to connect.

Profile Fundamental:

For the execution of this profile, use of security features such as authorization, authentication and encryption is optional. Support for authentication and encryption is mandatory, such that the device can take part in the corresponding procedures if requested from a peer device. If use of security features is desired, the two devices are paired during the connection establishment phase Bonding is not explicitly used in this profile, and thus, support for bonding is optional.
Link establishment is initiated by DevA. Service discovery procedures have to be performed to set up an emulated serial cable connection. There are no fixed master slave roles.

3.4 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is designed to provide significantly lower power consumption. This allows Android apps to communicate with BLE devices that have low power requirements, such as proximity sensors, heart rate monitors, fitness devices, and so on
BTLE devices will go into sleep mode and wake only for connection attempts or events. As a result, the software developer needs to understand a few of the basic concepts in BTLE.

Generic Attribute Profile (GATT)—The GATT profile is a general specification for sending and receiving short pieces of data known as "attributes" over a BLE link. All current Low Energy application profiles are based on GATT.

Attribute Protocol (ATT)—GATT is built on top of the Attribute Protocol (ATT). This is also referred to as GATT/ATT. ATT is optimized to run on BLE devices. To this end, it uses as few bytes as possible. Each attribute is uniquely identified by a Universally Unique Identifier (UUID), which is a standardized 128-bit format for a string ID used to uniquely identify information. The *attributes* transported by ATT are formatted as *characteristics* and *services*.
Characteristic—A characteristic contains a single value and 0-n descriptors that describe the characteristic's value. A characteristic can be thought of as a type, analogous to a class.
Descriptor—Descriptors are defined attributes that describe a characteristic value. For example, a descriptor might specify a human-readable description, an acceptable range for a characteristic's value, or a unit of measure that is specific to a characteristic's value.
Service—A service is a collection of characteristics. For example, you could have a service called "Heart Rate Monitor" that includes characteristics such as "heart rate measurement.

GATT Operations

These are all commands a client can use to discover information about the server.

- Discover UUIDs for all primary services. This operation can be used to determine if device supports Device Information Service, for example.

- Discover all characteristics for a given service. For example, some heart rate monitors also include a body sensor location characteristic.
- Read and write descriptors for a particular characteristic. One of the most common descriptors used is the Client Characteristic Configuration Descriptor. This allows the client to set the notifications to *indicate* or *notify* for a particular characteristic. If the client sets the Notification Enabled bit, the server sends a value to the client whenever the information becomes available. Similarly, setting the Indications Enabled bit will also enable the server to send notifications when data is available, but the indicate mode also requires a response from the client.

- Read and write to a characteristic. Using the example of the heart rate monitor, the client would be reading the heart rate measurement characteristic. Or, a client might write to a characteristic while upgrading the firmware of the remote device.

3.5BLE USE CASES

Blood pressure profile. Proximity Profile, Heart Rate Profile etc.Each profile is designed with fix set of characteristics having unique UUID.

3.6 BLE LIMITATION

As BLE is designed to keep connection active for very small time .so only small brust of dataCan be transferred using BLE in form of characteristics and descriptors as ATT protocol designed.

Chapter-4

HOST-CARD EMULATION

4.1 HCE

Smartphones have already used as mobile payments. Most of the modern mobile devices are equipped with NFC module, and by using such devices, it is possible to get rid of carrying heavy, metal keys, pass-cards, etc. People often forget keys at home and they are relatively small and easy to lose. Instead of carrying all these keys, we present an NFC- enabled Access Control and Management System, which by the help of mobile devices, NFC technology and HCE mode, introduced in Android 4.4, makes possible for people to use only one single key. ISO 7816-4 smart card standard is used for emulation a smart card and the data exchange between the mobile device and NFC-reader.

4.2 Using Secure Element for Card Emulation

When NFC card emulation is provided using a secure element, the card to be emulated is provisioned into the secure element on the device through an Android application. Then, when the user holds the device over an NFC terminal, the NFC controller in the device routes all data from the reader directly to the secure element. The secure element itself performs the communication with the NFC terminal, and no Android application is involved in the transaction at all. After the transaction is complete, an Android application can query the secure element directly for the transaction status and notify the user.

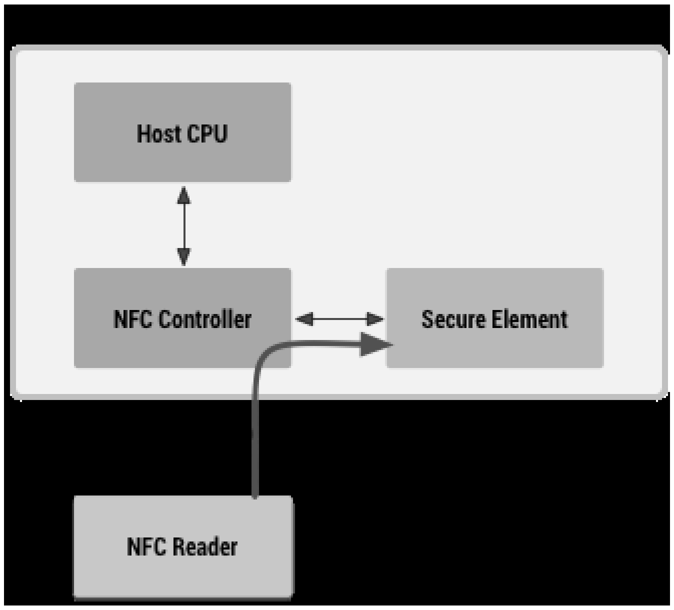


FIGURE 10: NFC card emulation with a secure element [9]

4.3 Using Host-based Card Emulation

When an NFC card is emulated using host-based card emulation, the data is routed to the host CPU on which Android applications are running directly, instead of routing the NFC protocol frames to a secure element.

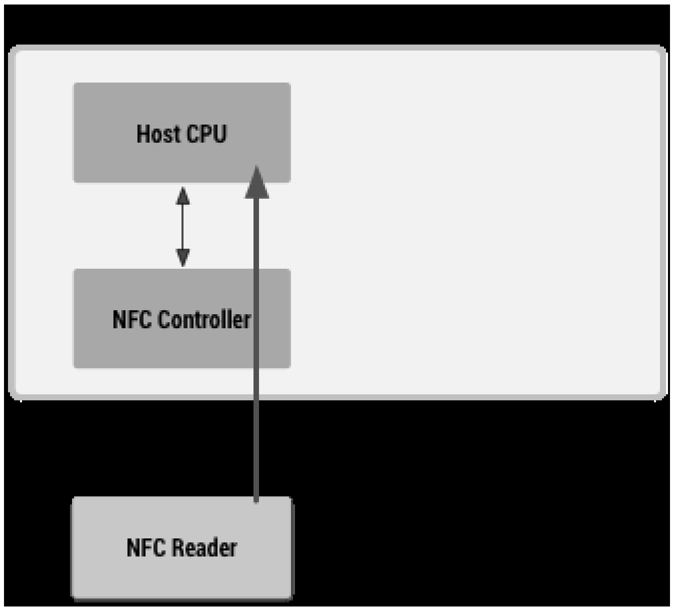


FIGURE 11: NFC card emulation without a secure element [9]

4.4 HCE Services

Android Service components (known as "HCE services") serves as the base for the HCE Architecture in Android. A service can run in the background without any user interface, this is one of the key features of a service. This is a natural fit for many HCE applications like loyalty or transit cards, with which the user shouldn't need to launch the app to use it. Instead, tapping the device against the NFC reader starts the correct service (if not already running) and executes the transaction in the background. One can also launch additional UI (such as user notifications) from your service if that makes sense.

When the user taps a device to an NFC reader, the Android system needs to know which HCE service the NFC reader actually wants to talk to. This is where the ISO/IEC 7816-4 specification comes in: it defines a way to select applications, centred around an Application ID (AID). An AID consists of up to 16 bytes. If you are emulating cards for an existing NFC reader infrastructure, the AIDs that those readers are looking for are typically well-known and publicly registered (for example, the AIDs of payment networks such as Visa and MasterCard).If one want to deploy new reader infrastructure for your own application, you will need to register your own AID(s). The registration procedure for AIDs is defined in the ISO/IEC 7816-5 specification. Google recommends registering an AID as per 7816-5 if you are deploying a HCE application for Android, as it will avoid collisions with other applications.

4.5 HCE Services support

To emulate an NFC card using host-based card emulation, a Service component that handles the NFC transactions needs to be created. The application can check whether a device supports HCE by checking for the *FEATURE_NFC_HOST_CARD_EMULATION* feature. The <*uses-feature*>tag in the manifest of application should be used to declare that the app uses the HCE feature, and whether it is required for the app to function or not.

Chapter-5

SYSTEM DESIGN

5.1MAJOR GOALS FOR BLUETOOTH AUTOMATION

1. Devices should not be scanned.
2. No Pairing Process popup should and user should be able to connect without showing any popup.
3. No Authorization popup should occur while receiving file and user should be able to receive file directly.

5.2 BT TRANSFER TECHNIQUES APPLICABILITY

As discussed above, Bluetooth has its own set of profiles support to achieve specific functionality. In case of File or Data Transfer PROFILES designed are OPP (OBJECT PUSH PROFILE) and FTP (FILE TRANSFER PROFILE) both of these uses OBEX (object exchange protocol) for transferring data.

OPP: Before we should initiate data transfer over, Bluetooth devices must be scanned and pairing should be completed. This pairing process will generate the link key using which pairing keys will be generated. Also once the pairing is complete OBEX protocol has its own authorization mechanism which ensures that proper user confirmation should be taken before accepting a file transfer. It requires user involvement and data can be transferred in fixed packet size mutually agreed upon.

FTP: In case of FTP as well before data transfer is initiated**,** Bluetooth devices must be scanned and pairing should be completed. This pairing process will generate the link key using which pairing keys will be generated. Also once the pairing is complete OBEX protocol has its own authorization mechanism which ensures that proper user confirmation should be taken before accepting a file transfer. It requires user involvement and data can be transferred in fixed packet size mutually agreed upon.

BLE: Using Bluetooth Low Energy user involvement in pairing process can be avoided but with BLUETOOTH Low energy design only small chunks of data can be transferred and support of file transfer is not there.

Hence any of above method for FILE TRANSFER is not suitable for efficient health card communication.

5.3 BLUETOOTH AUTOMATION ARCHITECTURE

1. All communication in Bluetooth is inherently packet based but RFCOMM emulates a serial cable where bytes can be sent individually. This means that the application sitting on top of RFCOMM has to detect packet frames. As we can see in BT protocol stack OBEX (Protocol for file transfer) is sitting on top of RFCOMM. OBEX protocol has its own authorization mechanism, this will prompt user whether to accept file or not.

   To avoid authorization we have two purposed solution

- Changes should be done in BT stack at OBEX layer to avoid authorization
- Avoid using OBEX and Application should directly interact with RFCOMM.

   As changes in BT stack is not possible as solution will not be universal so in our case we have directly interacted with RFCOMM Layer.

2. Bluetooth services generally require either encryption or authentication and as such require pairing before they let a remote device connect. During pairing, the two devices establish a relationship by creating a shared secret known as a *link key*. If both devices store the same link key, they are said to be *paired* or *bonded*. A device that wants to communicate only with a bonded device can cryptographically authenticate the identity of the other device, ensuring it is the same device it previously paired with. Once a link key is generated, an authenticated Asynchronous Connection-Less (ACL) link between the devices may be encrypted to protect exchanged data.
   To authenticate remote Bluetooth device link needs to be generated and pairing should be performed which will involve BT DEVICES SCAN and user user involvement on pairing popup.

To avoid authentication at RFCOMM layer insecure RFCOMM sockets can be created.

API **CreateInsecureRfcommSocket.**

This will create Insecure BT Link between two BT devices and link keys will not be generated.

SocketcreateInsecureRfcommSocketToServiceRecord(UUID);

3. As user interaction has to be avoided for BT automation hence BT search for devices is not allowed. To overcome this BT MAC address of devices is exchanged over HCE (Host CARD Emulation) and insecure RFCOMM socket is created over exchanged MAC.

5.4RFCOMM:

The RFCOMM protocol emulates the serial cable line settings and status of an RS-232 serial port and is used for providing serial data transfer. RFCOMM connects to the lower layers of the *Bluetooth* protocol stack through the L2CAP layer.

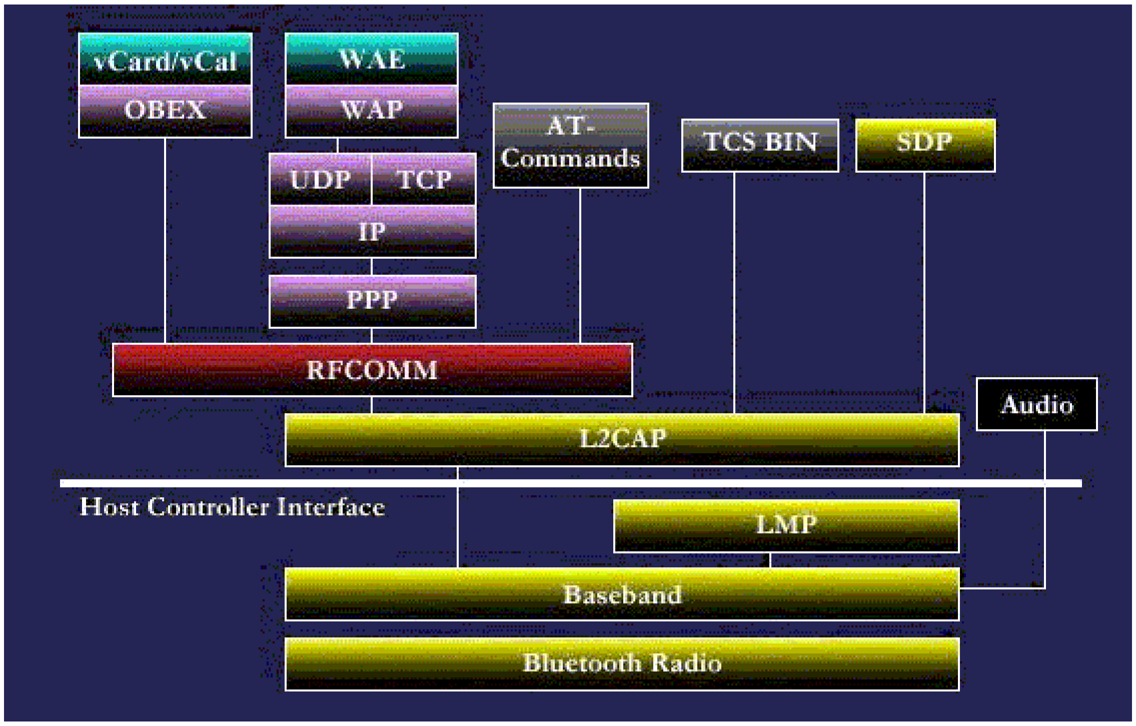


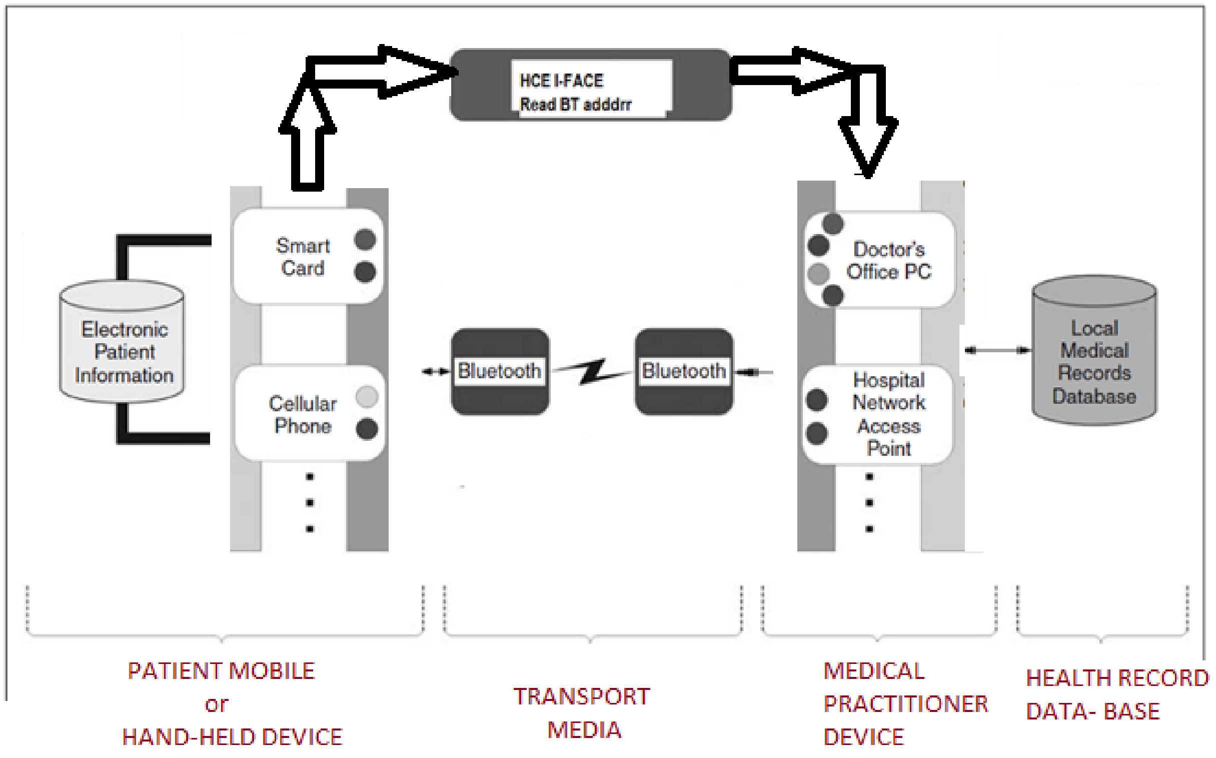
FIGURE 12: RFCOMM IN BLUETOOH STACK [5]

5.5 DESIGN:

Health care system design involves two major modules.

a) Smart Card supporting HCE for Storing data efficiently and provide required security to data

b) Wireless communication over Bluetooth which should be automated i.e. no user interaction involved.



HCE I-FACE
Read BT adddrr
Electronic Patient Information
Smart Card
Cellular Phone
Bluetooth
Bluetooth
Doctor's Office PC
Hospital Network Access Point
Local Medical Records Database
PATIENT MOBILE or HAND-HELD DEVICE
TRANSPORT MEDIA
MEDICAL PRACTITIONER DEVICE
HEALTH RECORD DATA- BASE

FIGURE 13: HEALTH_CARE SYSTEM ARCHITECTURE [1]

5.6APPLICATION ARCHITECTURE

BT Automation App is based on **CLIENT- SERVER** communication over **BT RFCOMM** socket. One device will act as **CLIENT** while other will act as **SERVER**. These devices will communicate using insecure RFCOMM sockets is created using RFCOMM LAYER API's.

Application will be installed on both communication devices, and have three different thread running.

- MAIN ACTIVITY
- CLIENT THREAD
- SERVER THREAD

CLIENT THREAD:

BT device either Patient Health Card or Doctor Handheld device both can act as Client. Any device which will initiate transfer of data will be in CLIENT ROLE.

SERVER THREAD:

BT device either Patient Health Card or Doctor Handheld device both can act as Server. Any device which will accept transferred data will be in SERVER ROLE.

ENCRYPTION/DECRYPTION

Both communicating devices needs to have BT Automation application installed and Client and Server thread always running on both. Device which needs to transfer data will act as CLIENT and will initiate RFCOMM socket connection. At Client side data will be read from FILE and will write to socket using **WRITE** API. At other end Server thread will read data using **READ** API. Client (Doctor Device) will communicate with Server Device (Patient) device using RFCOMM socket. Client will encrypt data using below bytes sequence at one end, while server thread will decrypt data using same sequence.

Ist BYTE: To decide whether communication is for data to be sent or command will be sent to Patient device
IST BYTE= 0*0-> Data
IST BYTE=0*21->command

2nd **BYTE:** HEADER MSB **,**Fixed value already known at server end.
3rd **BYTE:**HEADER LSB**,** Fixed value already known at server end.

4th **to** 7th **BYTE:** Size of Data to be transferred.

8th **to** 23rd **BYTE**: Message Digest to match image using MD5 algorithm.

Next Chunk of bytes is actual data of size **4192** bytes in each **CHUNK.**

5.7 USE CASES

Use case 1: "Medical Practitioner needs to Transfer Report"
In this case APPLICATION at doctor device will act as client. BT MAC address of patient device will exchanged using HCE Tap of both devices. On press of SEND button on App, FILE browser will open and doctor can select desired REPORT. Socket connection will be established with Patient device which will act as SERVER and Report will be send to Patient. App on Patient side after receiving report will save it to desired path.

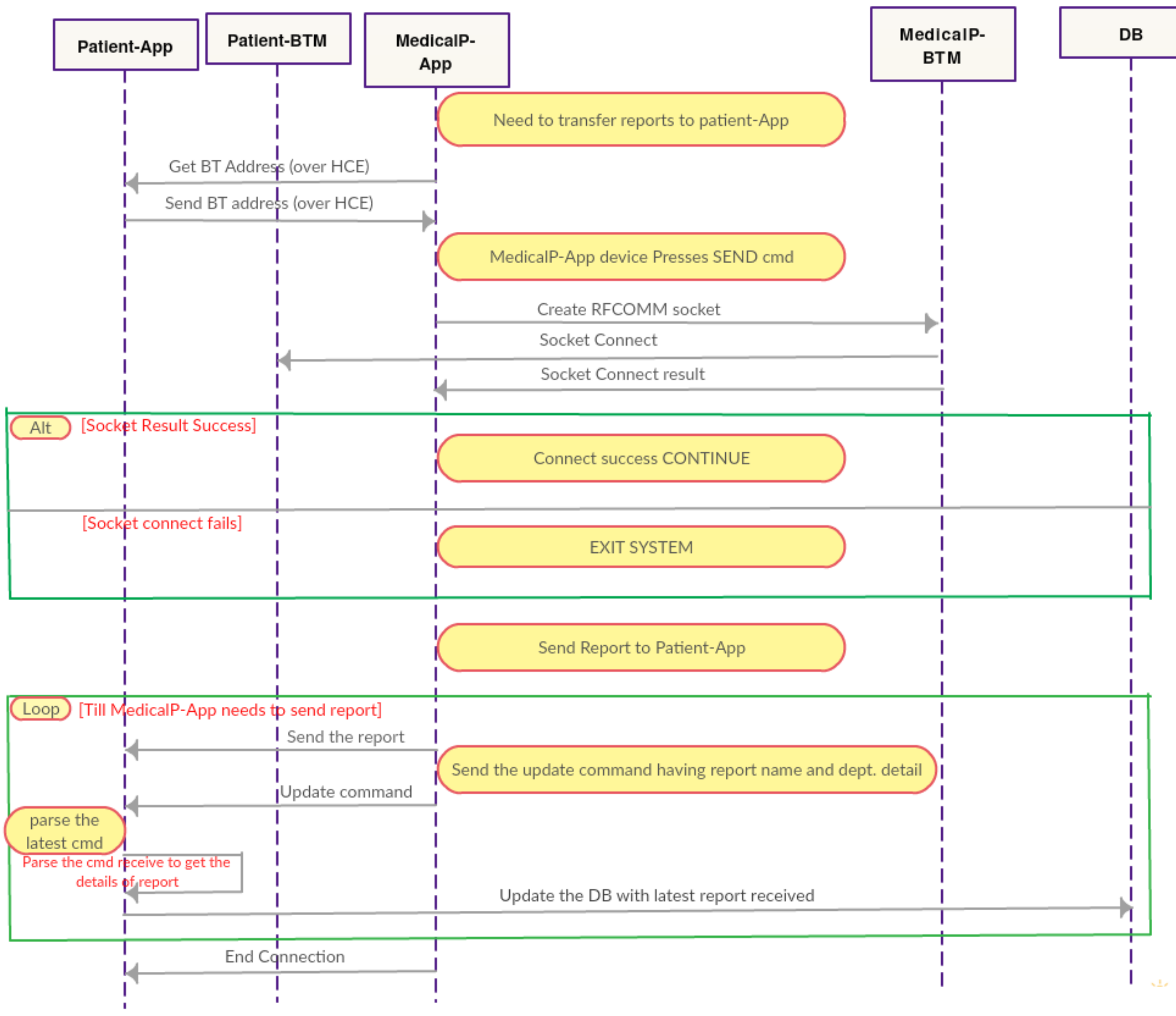
Use case 2:**"Medical Practitioner needs to get report from Patient Device"**
In this case APPLICATION at doctor device will act as Client as well as Sever. Patient device will be working in both role of Client and Server. On HCE tap doctor device will get BT MAC of patient. Doctor will enter full path of report on Patient device in TEXT filed provided on Application interface and press GET button and will act as Client. Socket's connection will be established and Patient device will receive command. On receiving command, patient device will send medical report from path provided and it will now act as client will establish socket connection to doctor device will act as SERVER.

Use case 3: "Patient needs to Transfer Report to Medical Practitioner"
In this case APPLICATION at patient device will act as client. BT MAC address of medical practitioner device will exchanged using HCE Tap of both devices. On press of SEND button on App, FILE browser will open and doctor can select desired REPORT. Socket connection will be established with Patient device which will act as SERVER and Report will be send to Patient. App on medical practitoner side after receiving report will save it to desired path.

Chapter-6

SEQUENCE DIAGRAM

=



Patient-App
Patient-BTM
MedicalP-App
MedicalP-BTM
DB
Need to transfer reports to patient-App
Get BT Address (over HCE)
Send BT address (over HCE)
MedicalP-App device Presses SEND cmd
Create RFCOMM socket
Socket Connect
Socket Connect result
Alt
[Socket Result Success]
Connect success CONTINUE
[Socket connect fails]
EXIT SYSTEM
Send Report to Patient-App
Loop
[Till MedicalP-App needs to send report]
Send the report
Send the update command having report name and dept. detail
Update command
parse the latest cmd
Parse the cmd receive to get the details of report
Update the DB with latest report received
End Connection

FIGURE 14: SEQUENCE DIAGRAM
MEDICAL P. SENDS REPORT

Sequence diagram for scenario where doctor send medical record to patient device at application level (user level).
On Tap of Doctor and patient device, Bluetooth address of Patient device is over HCE to Doctor device. automated pairing over Bluetooth will be done between Doctor and Patient device, which requires no user intervention. Doctor can transfer required report to patient device.

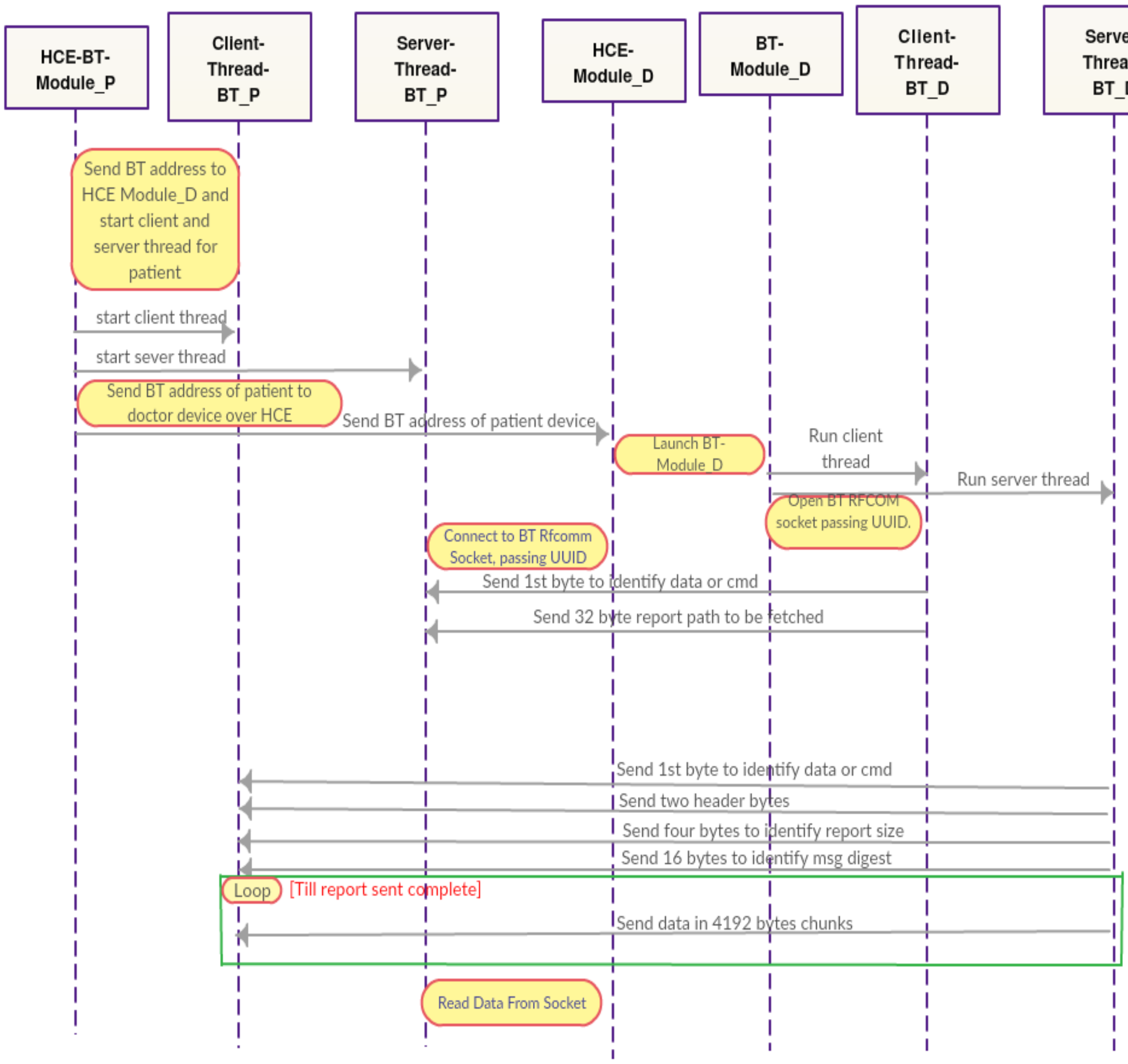

HCE-BT-Module_P
Client-Thread-BT_P
Server-Thread-BT_P
HCE-Module_D
BT-Module_D
Client-Thread-BT_D
Server-Thread-BT_D
Send BT address to HCE Module_D and start client and server thread for patient
start client thread
start sever thread
Send BT address of patient to doctor device over HCE
Send BT address of patient device
Launch BT-Module_D
Run client thread
Run server thread
Open BT RFCOM socket passing UUID.
Connect to BT Rfcomm Socket, passing UUID
Send 1st byte to identify data or cmd
Send 32 byte report path to be fetched
Send 1st byte to identify data or cmd
Send two header bytes
Send four bytes to identify report size
Send 16 bytes to identify msg digest
Loop
[Till report sent complete]
Send data in 4192 bytes chunks
Read Data From Socket

FIGURE 15:SEQUENCE DIAGRAM:
SEND OPERATION AT PROTCOL

Doctor and patient devices will communicate over Bluetooth RFCOMM socket. Two threads (client and Server) will run at both side. Client thread will send data and Server thread will listen for data. Bytes sequence as shown in above diagram will be transferred from client(Patient device) side which will be decrypted at server side (Doctor) side and report will be received.

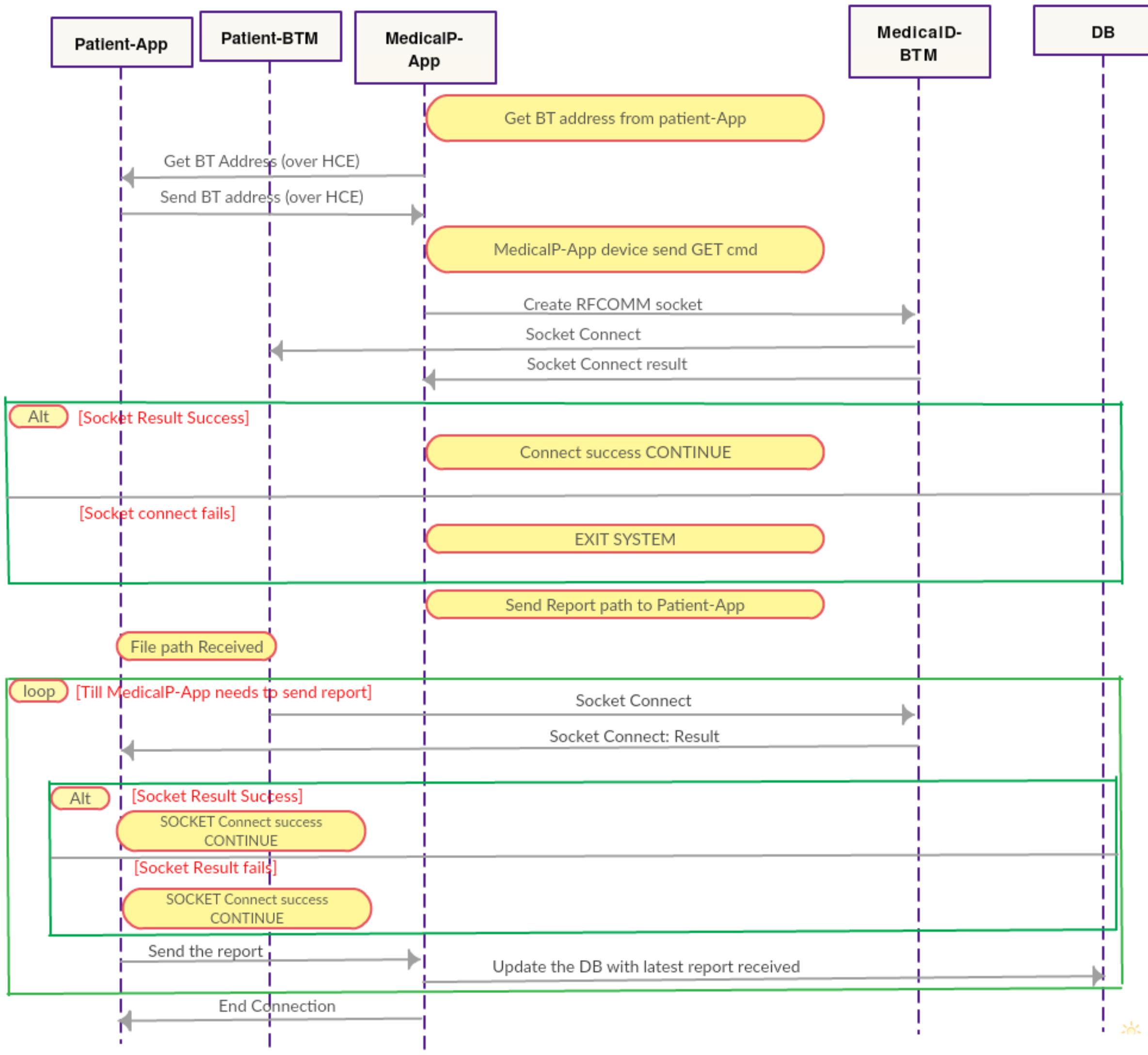


FIGURE 16: SEQUENCE DIAGRAM
MEDICAL P. GETS REPORT

On Tap of Doctor and patient device, Bluetooth address of Patient device is over HCE to Doctor device. Automated pairing over Bluetooth will be done between Doctor and Patient device, which requires no user intervention. Doctor will send path of report to Patient device over Bluetooth, from Patient device, desired report will transfer back to doctor device.

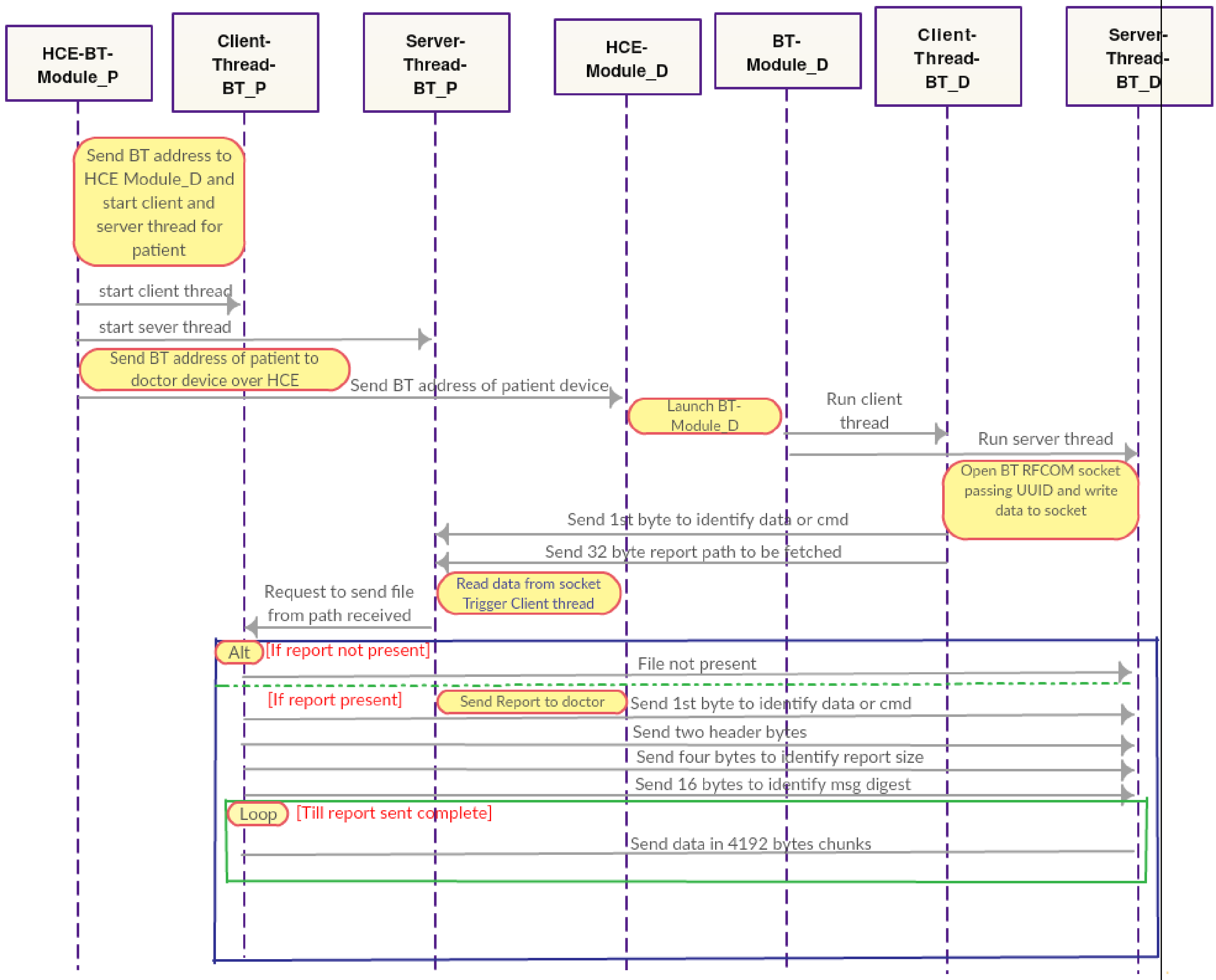

HCE-BT-Module_P
Client-Thread-BT_P
Server-Thread-BT_P
HCE-Module_D
BT-Module_D
Client-Thread-BT_D
Server-Thread-BT_D
Send BT address to HCE Module_D and start client and server thread for patient
start client thread
start sever thread
Send BT address of patient to doctor device over HCE
Send BT address of patient device
Launch BT-Module_D
Run client thread
Run server thread
Open BT RFCOM socket passing UUID and write data to socket
Send 1st byte to identify data or cmd
Send 32 byte report path to be fetched
Read data from socket Trigger Client thread
Request to send file from path received
Alt
[If report not present]
File not present
[If report present]
Send Report to doctor
Send 1st byte to identify data or cmd
Send two header bytes
Send four bytes to identify report size
Send 16 bytes to identify msg digest
Loop
[Till report sent complete]
Send data in 4192 bytes chunks

FIGURE 17::SEQUENCE DIAGRAM:
GET REPORT PROTOCOL LEVEL

Doctor and patient devices will communicate Over Bluetooth RFCOMM socket. Two threads (Client and Server) will run at both side. Client thread will send data and Server thread will listen for data. . First, sequence of bytes as shown above will be transfer from Doctor to patient device to transfer command and then from Patient to Doctor device to transfer Report

Chapter-7

AUTOMATION APP USER INERACTION

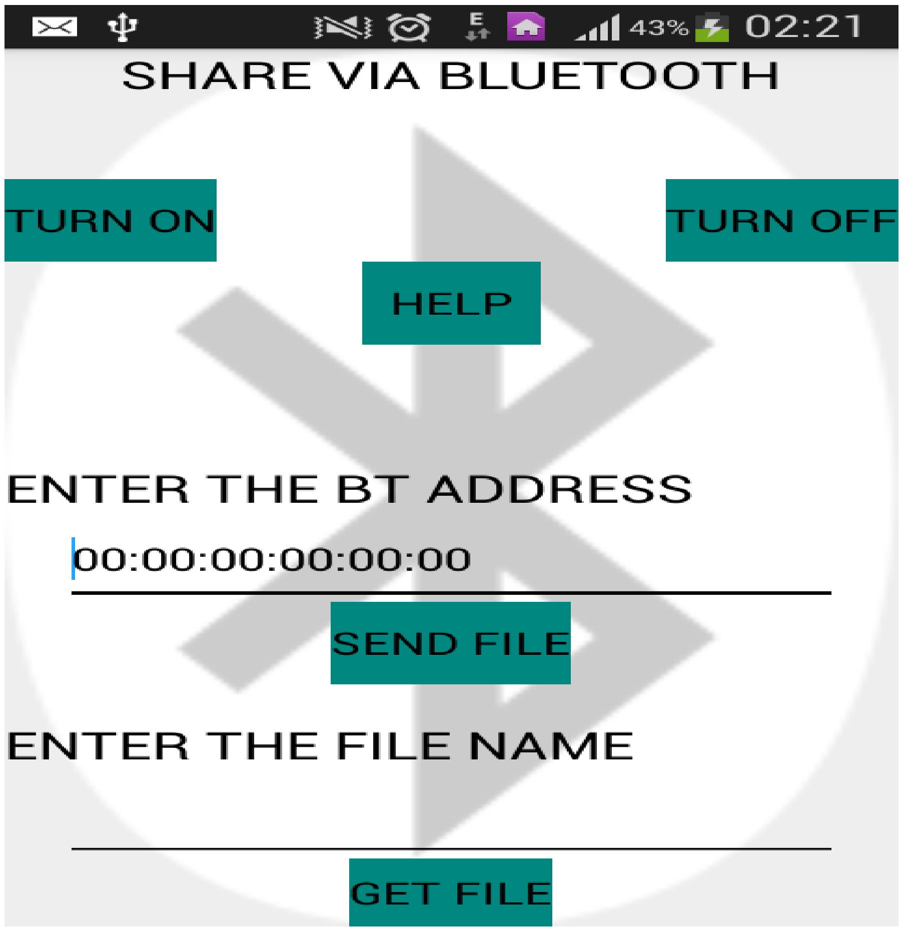


FIGURE18: APPLICATION MAIN PAGE

Medical Practitioner has option to Bluetooth TURN ON, Bluetooth TURN OFF. A Help Button will be provided to guide user. Medical Practitioner has to enter Bluetooth MAC address of patient device. On Pressing SEND button File Browser open to select medical REPORT, Which needs to be transferred to patient device. Once this App is integrated with HCE module BT address of patient device will be received through HCE on tap of devices and report will be shared over Bluetooth.

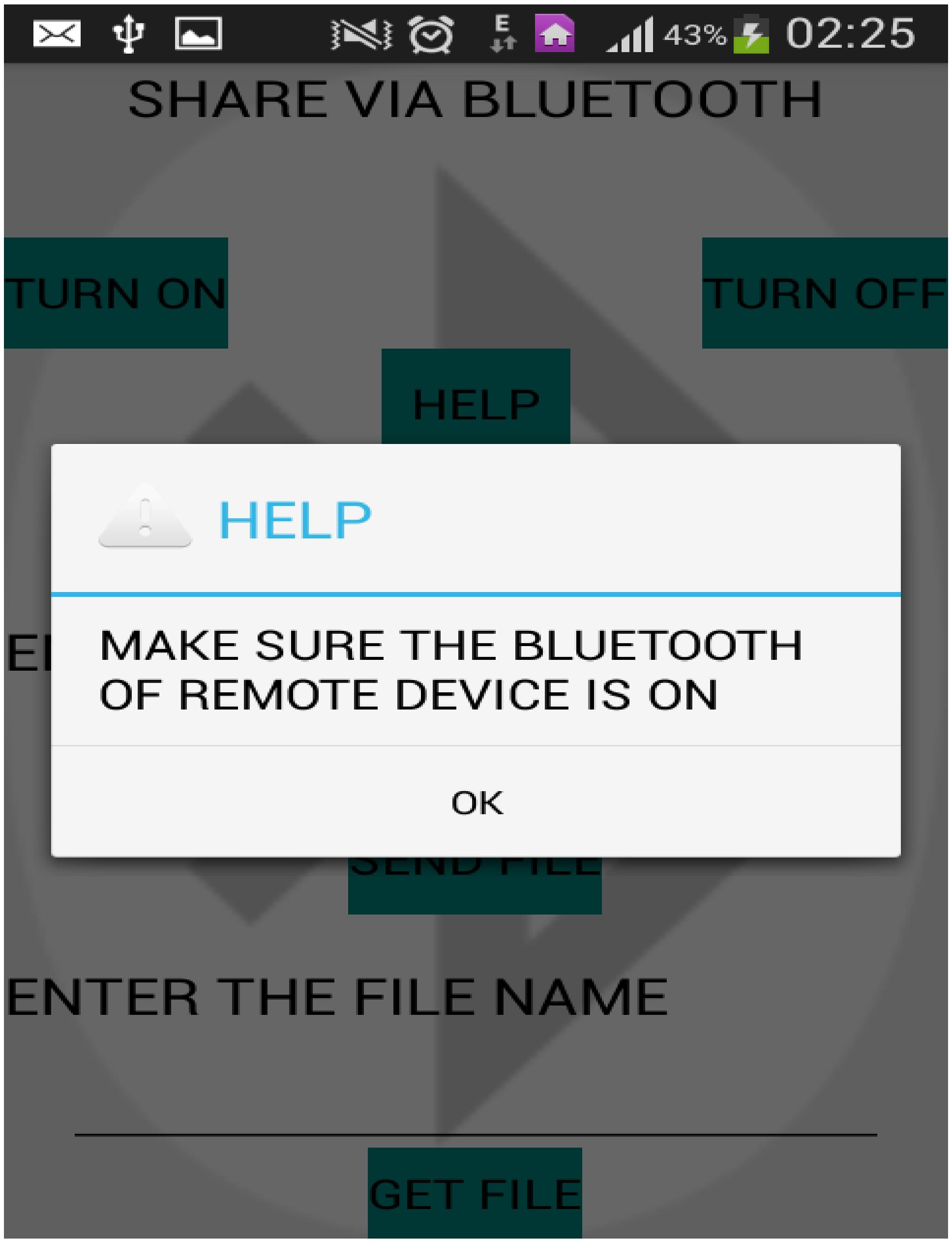


FIGURE 19: HELP BUTTON

HELP Button will guide the user about how to use Application

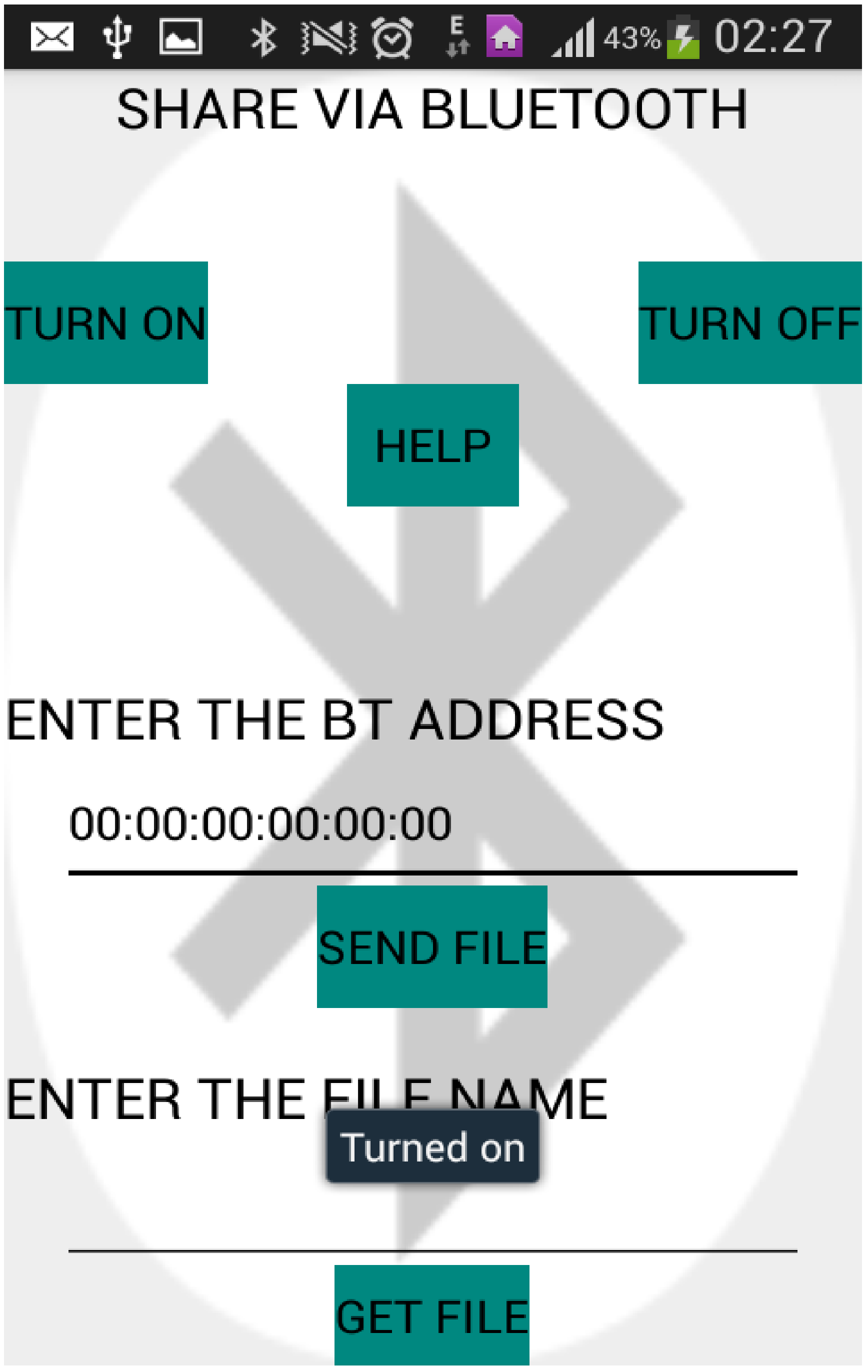


FIGURE20: BT ON BUTTON

It is provided to Turn ON Bluetooth of Medical Practitioner device before transferring reports to remote device.

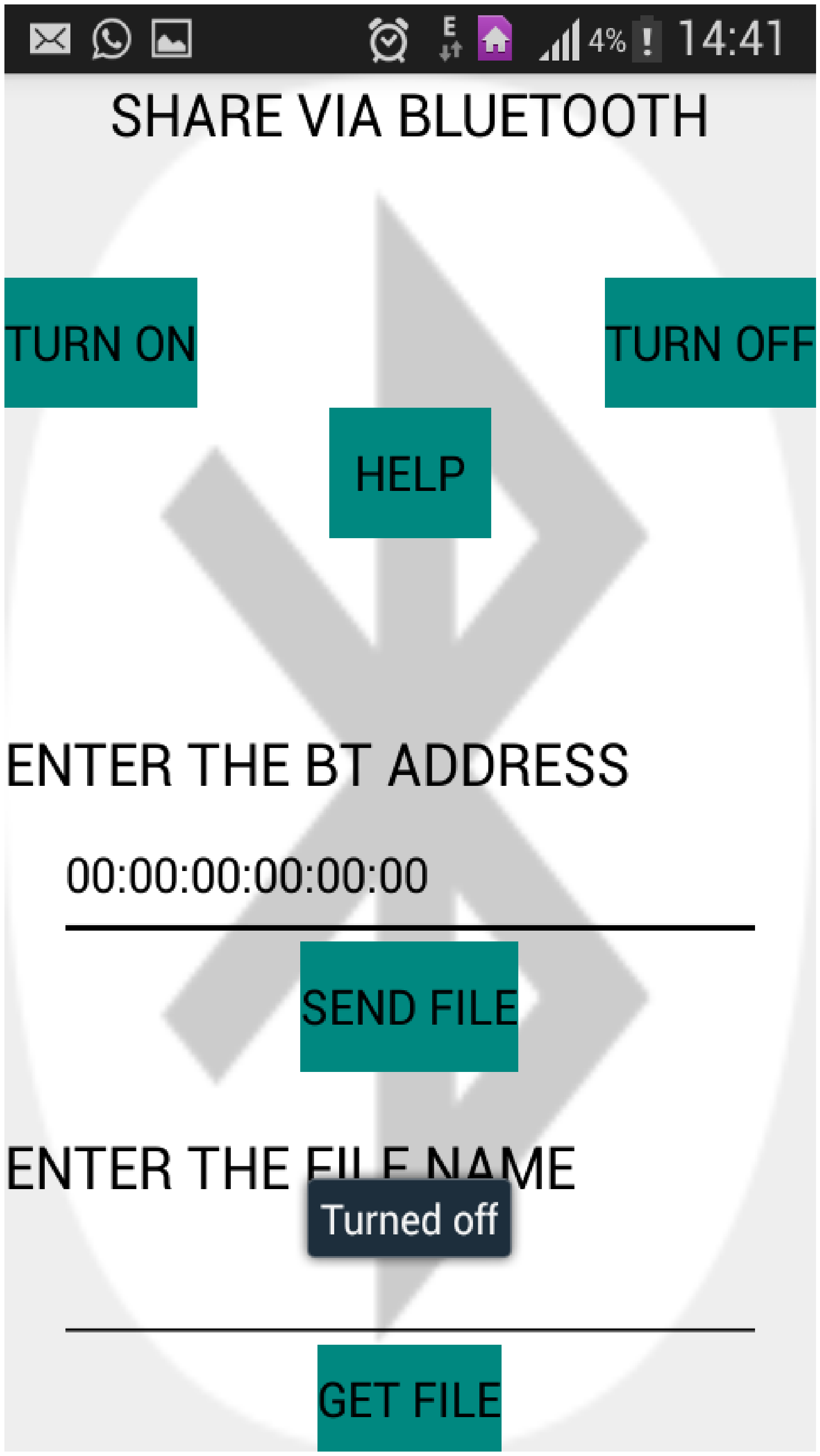


FIGURE21: BT OFF BUTTON

It is provided to Turn OFF Bluetooth of Medical Practitioner device in case it is not in use.

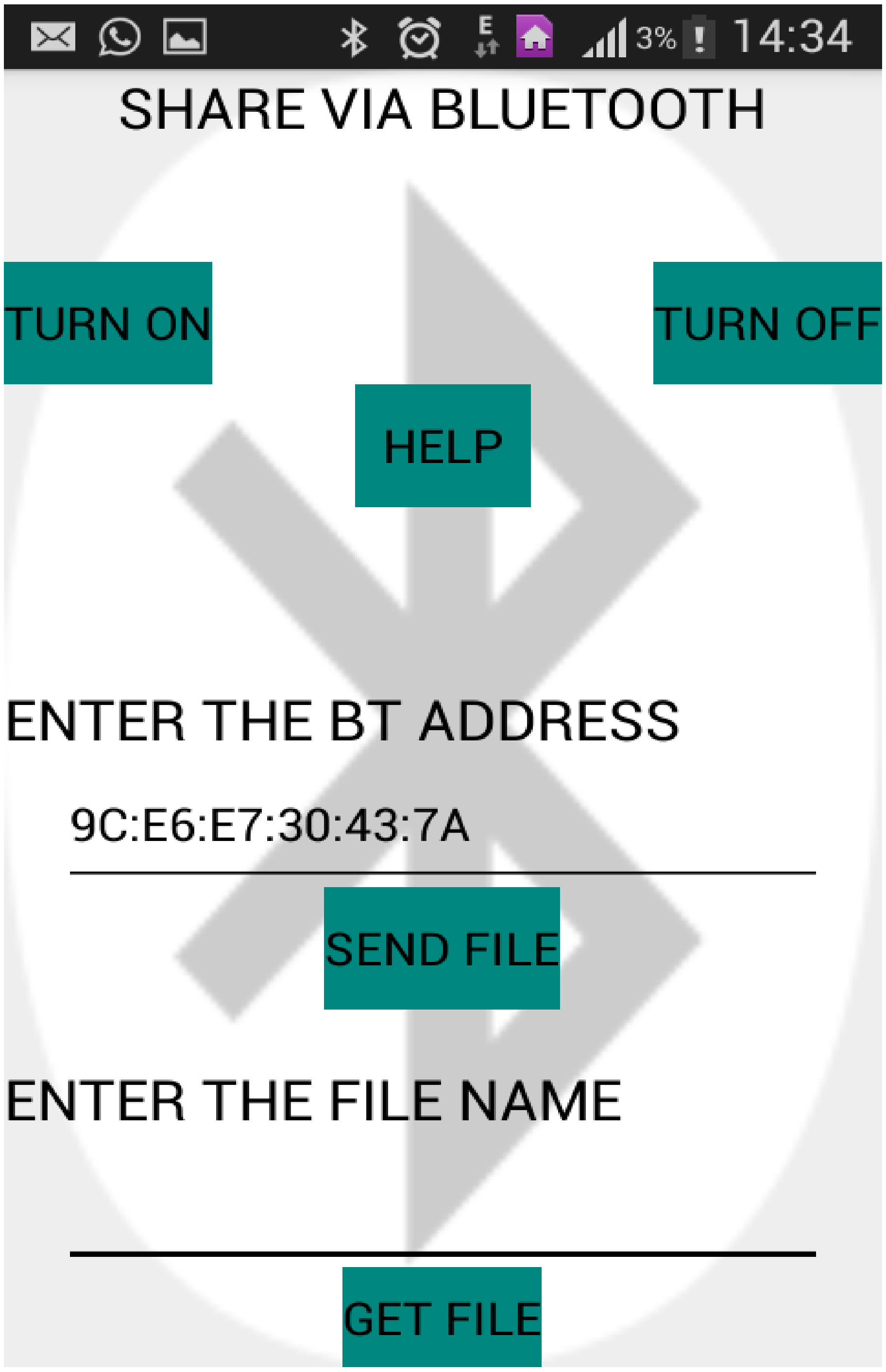


FIGURE 22: Text Field for Remote device BT MAC Address

Medical practitioner has to enter Bluetooth MAC address of patient Device where he wants to wants to Send or Get Reports from.

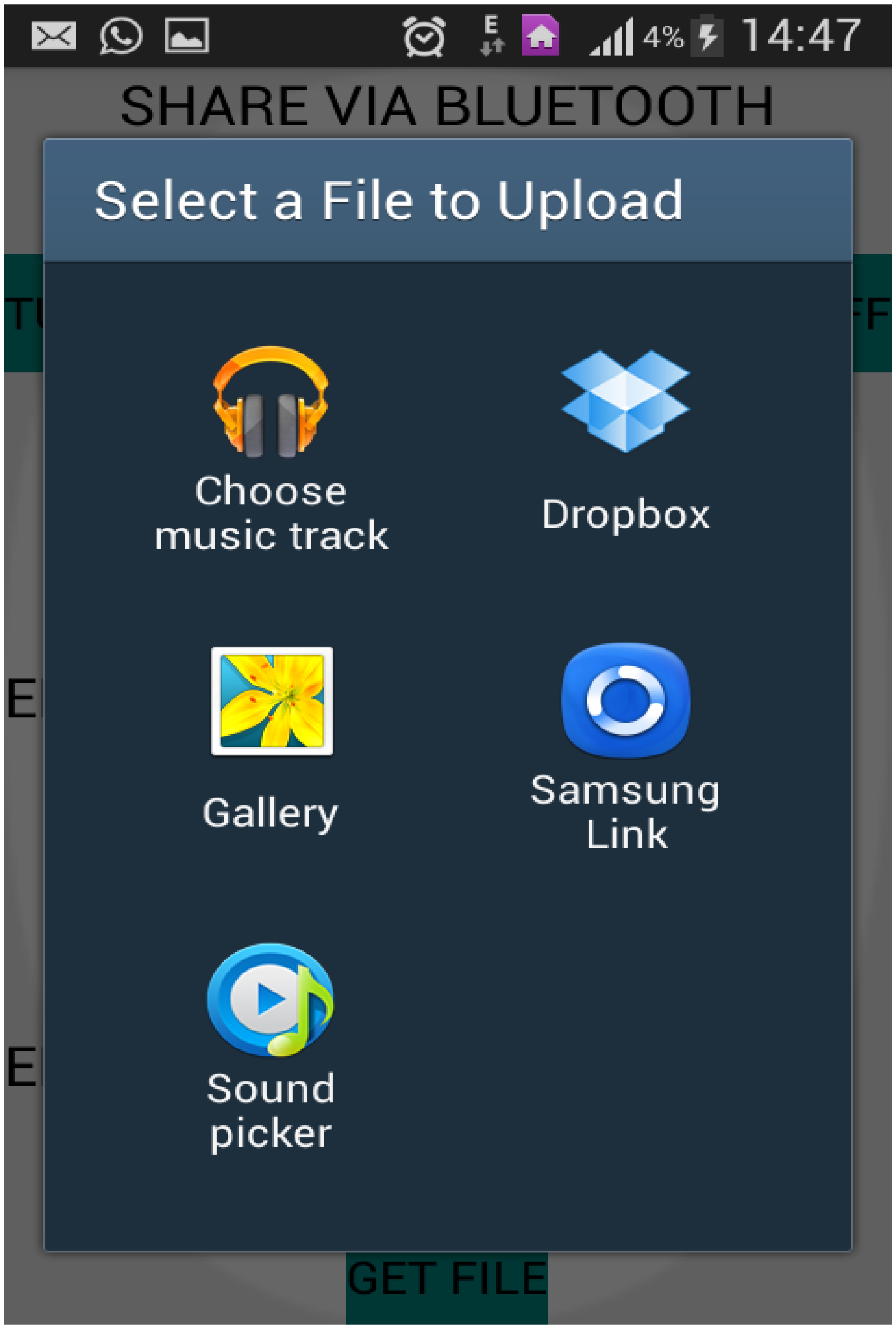


FIGURE 23:SEND Button to Send Reports

Medical practitioner has press SEND after entering MAC address of Patient Device.This will open a File Browser to select Report which medical practitioner wants totransfer to Patient device.

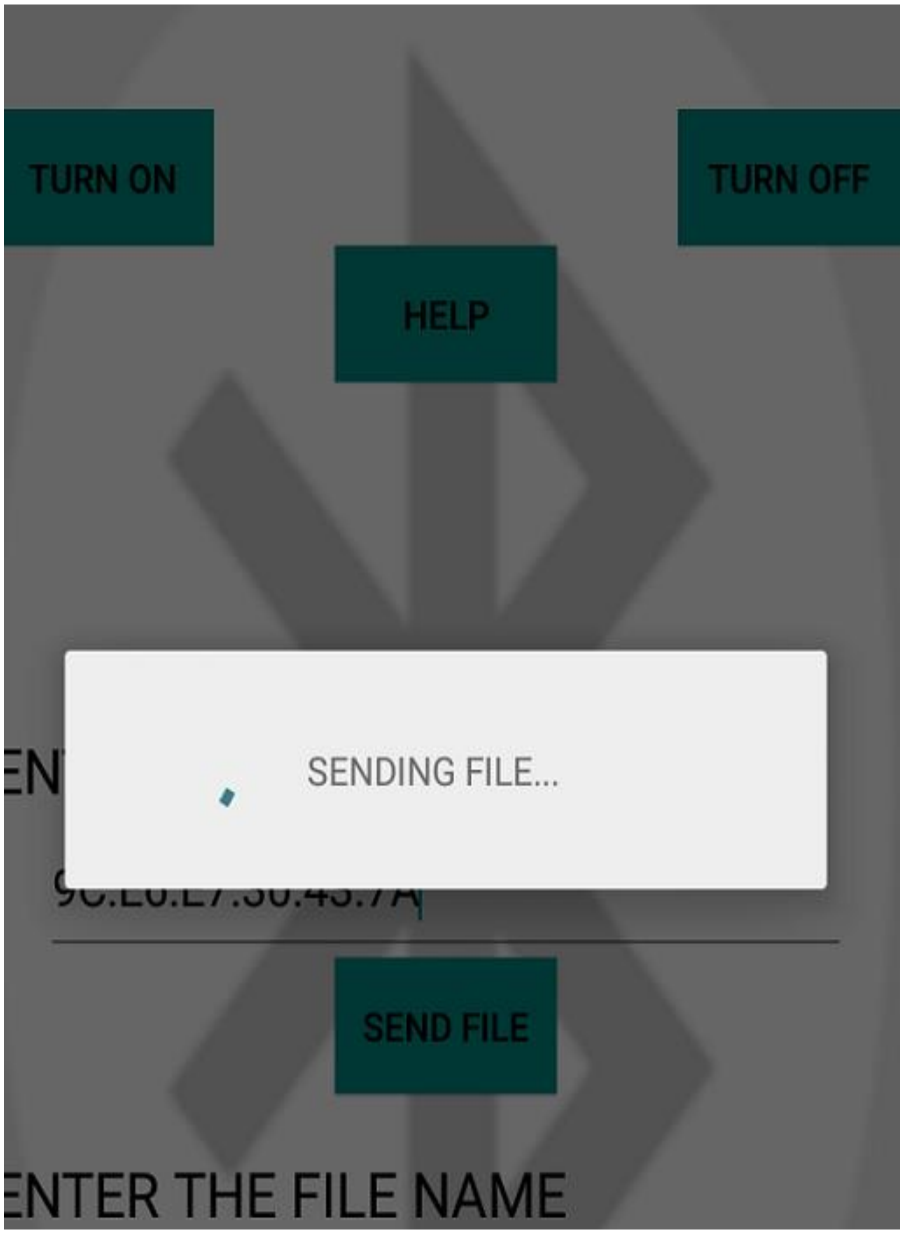


FIGURE 24: Sending Progress Bar

On Press of SEND Button Reports will be sent and progress bar will be shown.

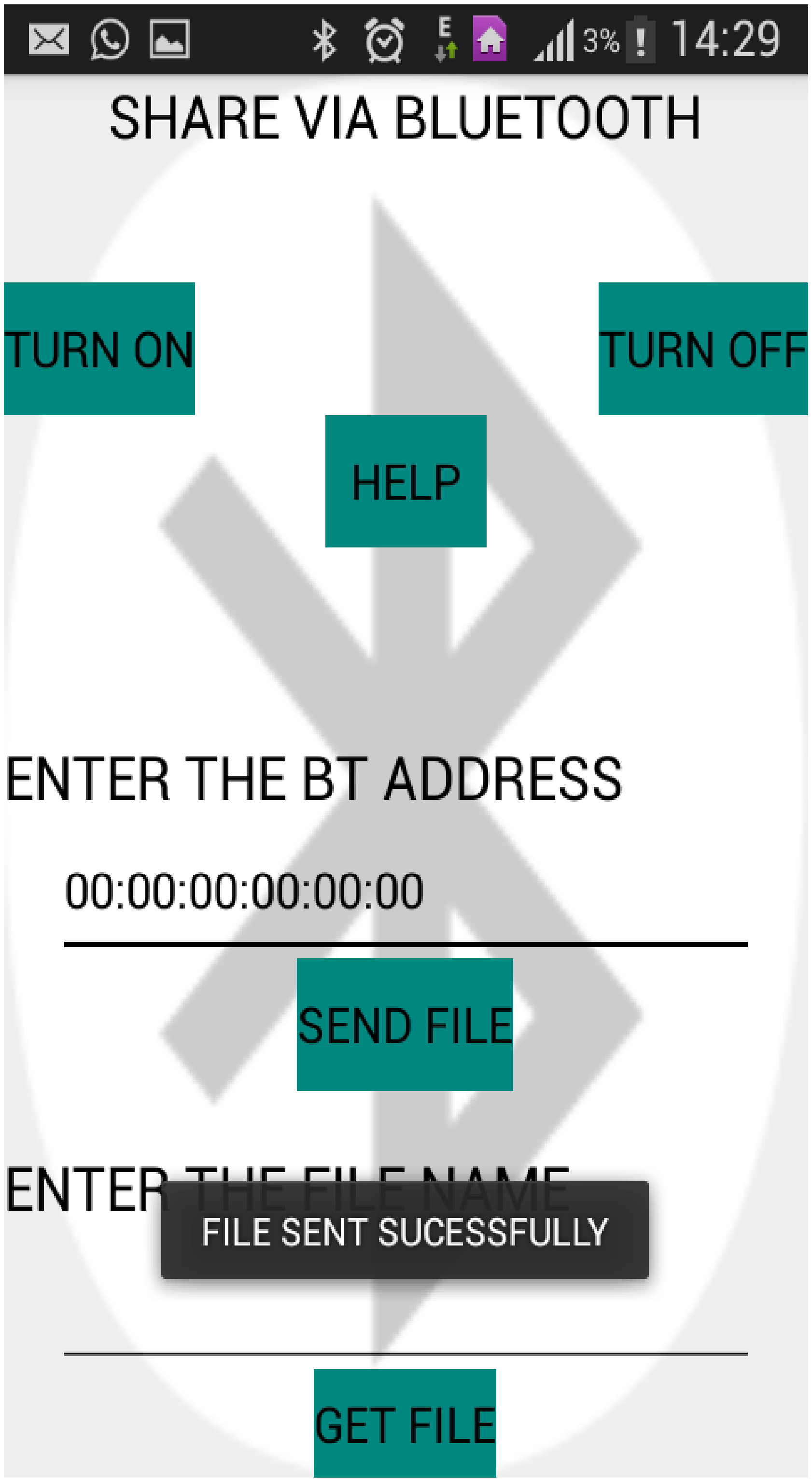


FIGURE 25: File Sent Success Toast

Once Reports are sent successfully SENT SUCCESS Toast will be shown.

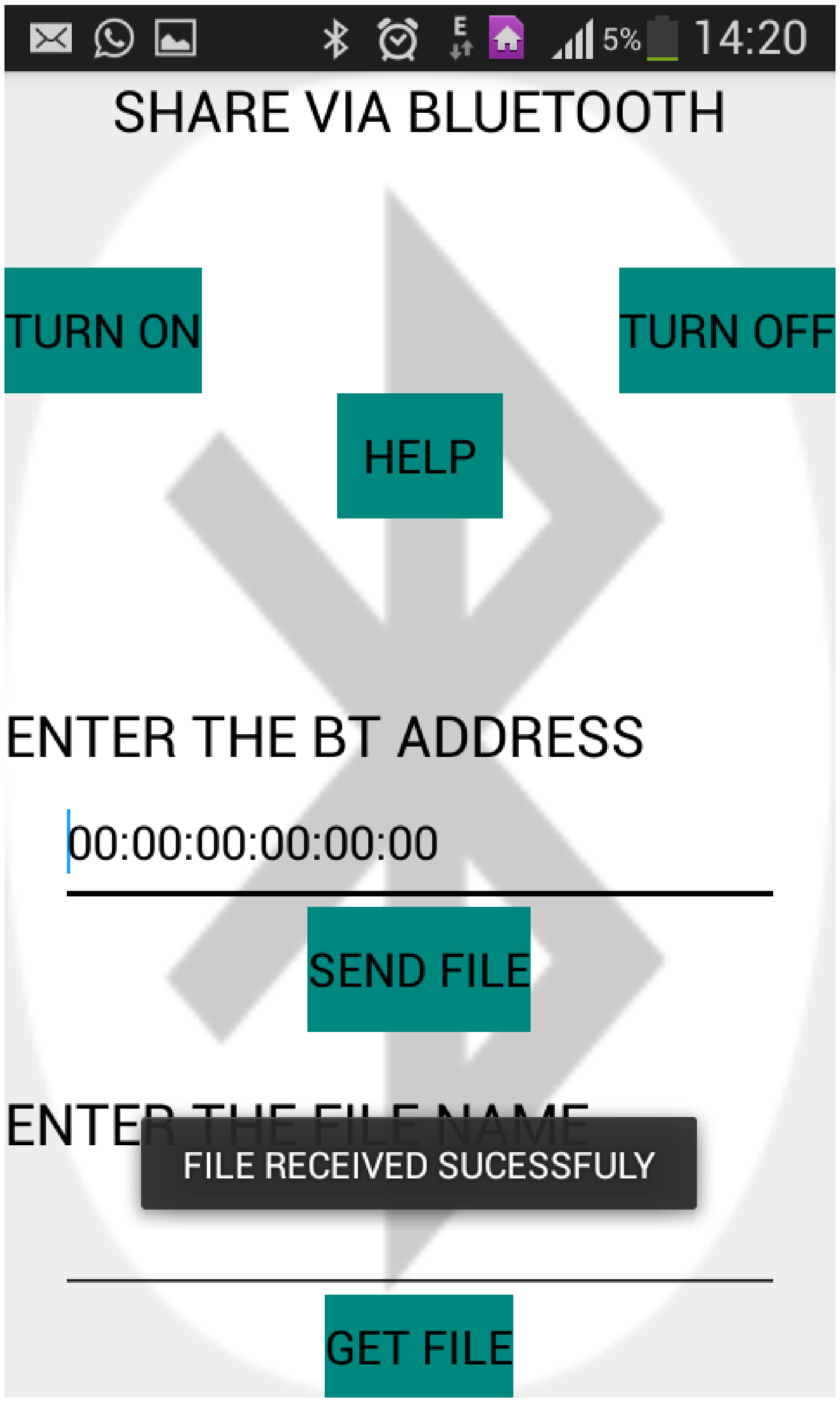


FIGURE 26: File Received Success Toast on receiving side

Once Reports are sent successfully received successfully toast will be shown on patient side.

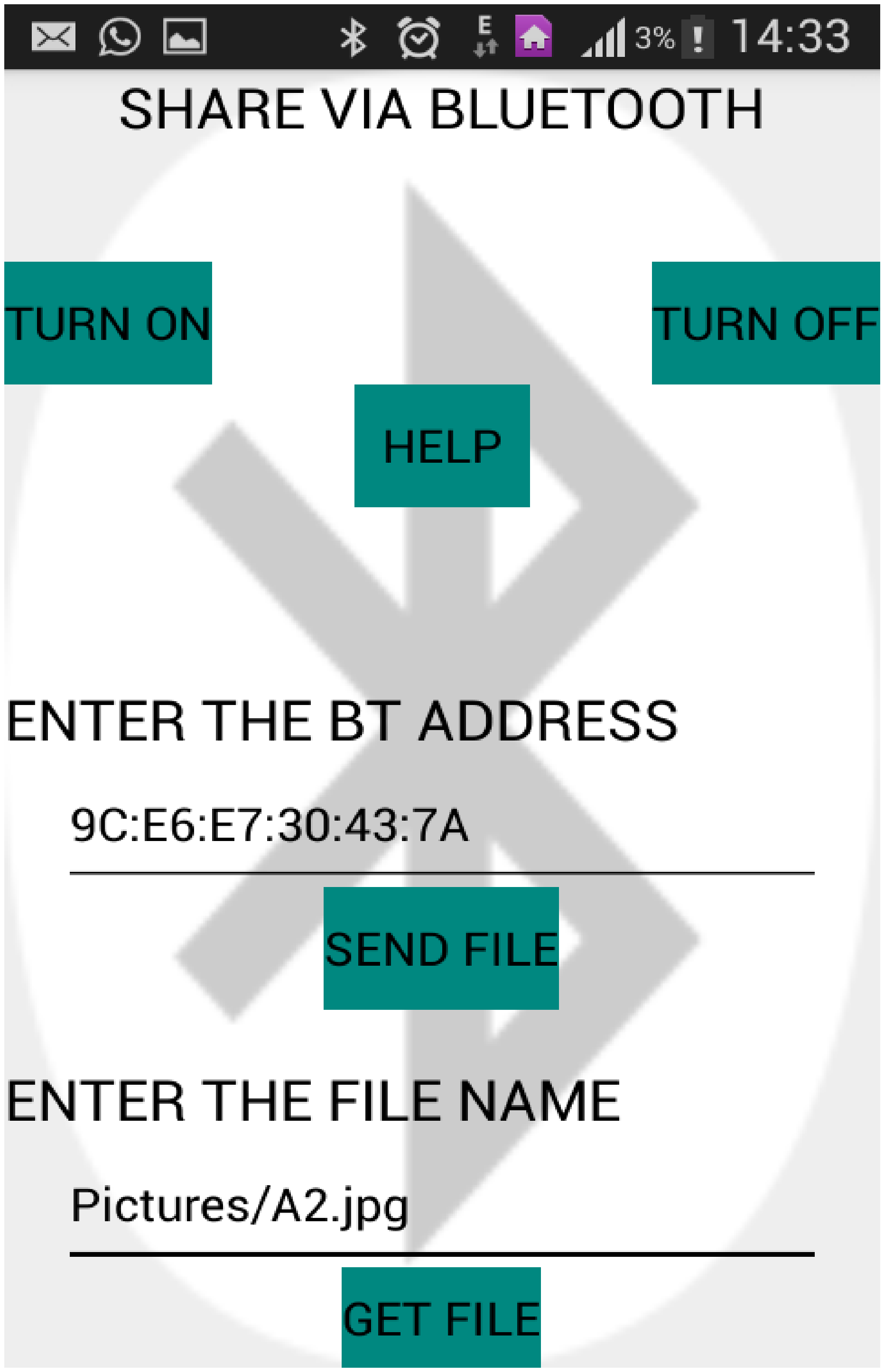


FIGURE 27: GET Reports Operation

Medical practitioner has to enter Bluetooth MAC address of patient device and Report's Full path on Patient device along with name of report to GET the Report from remote Patient device.

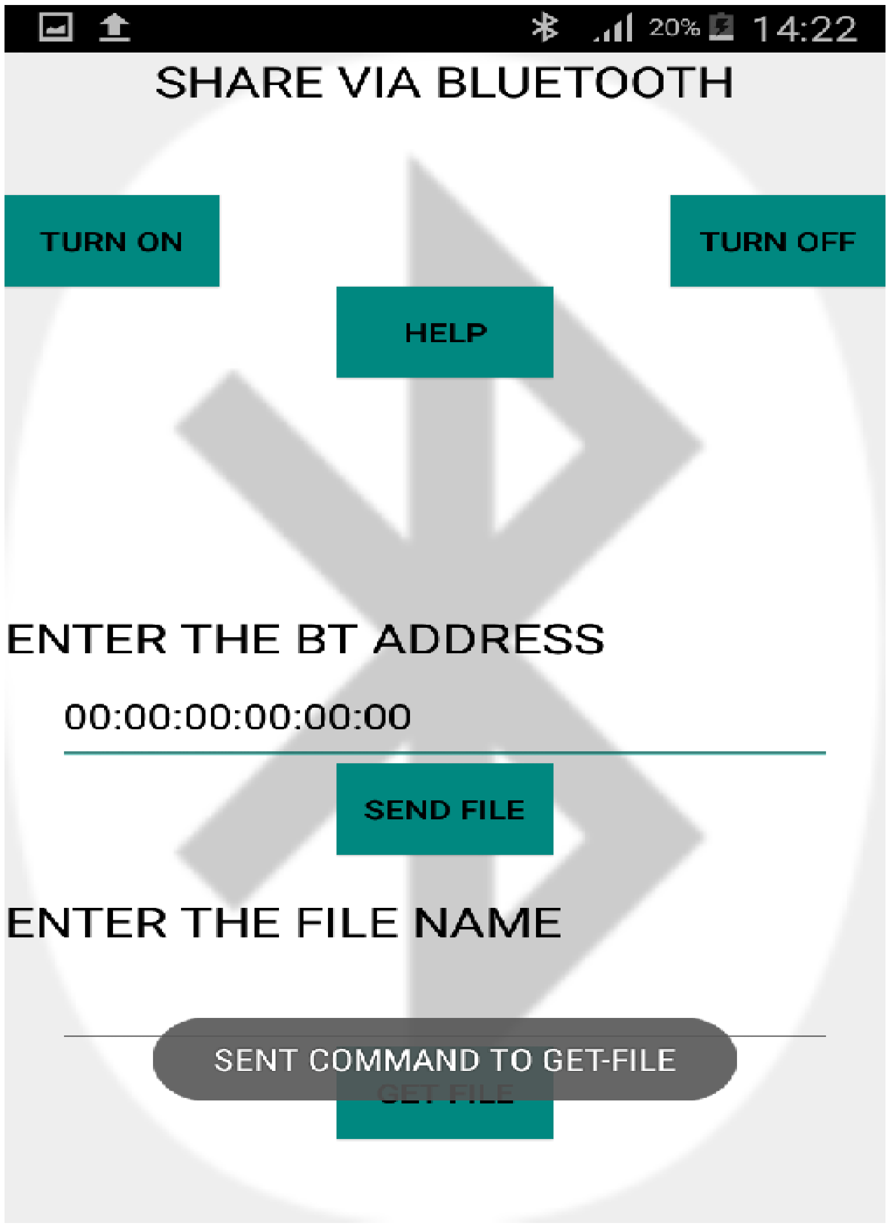


FIGURE 28: GET BUTTON PRESS

Once Medical practitioner Press GET Button after entering MAC address of remote device and full path of Report along with name a command will be sent to patient device which will indicate that patient has to transfer report to medical practitioner device from the path received from medical practitioner device which medical practitioner entered on Text field.

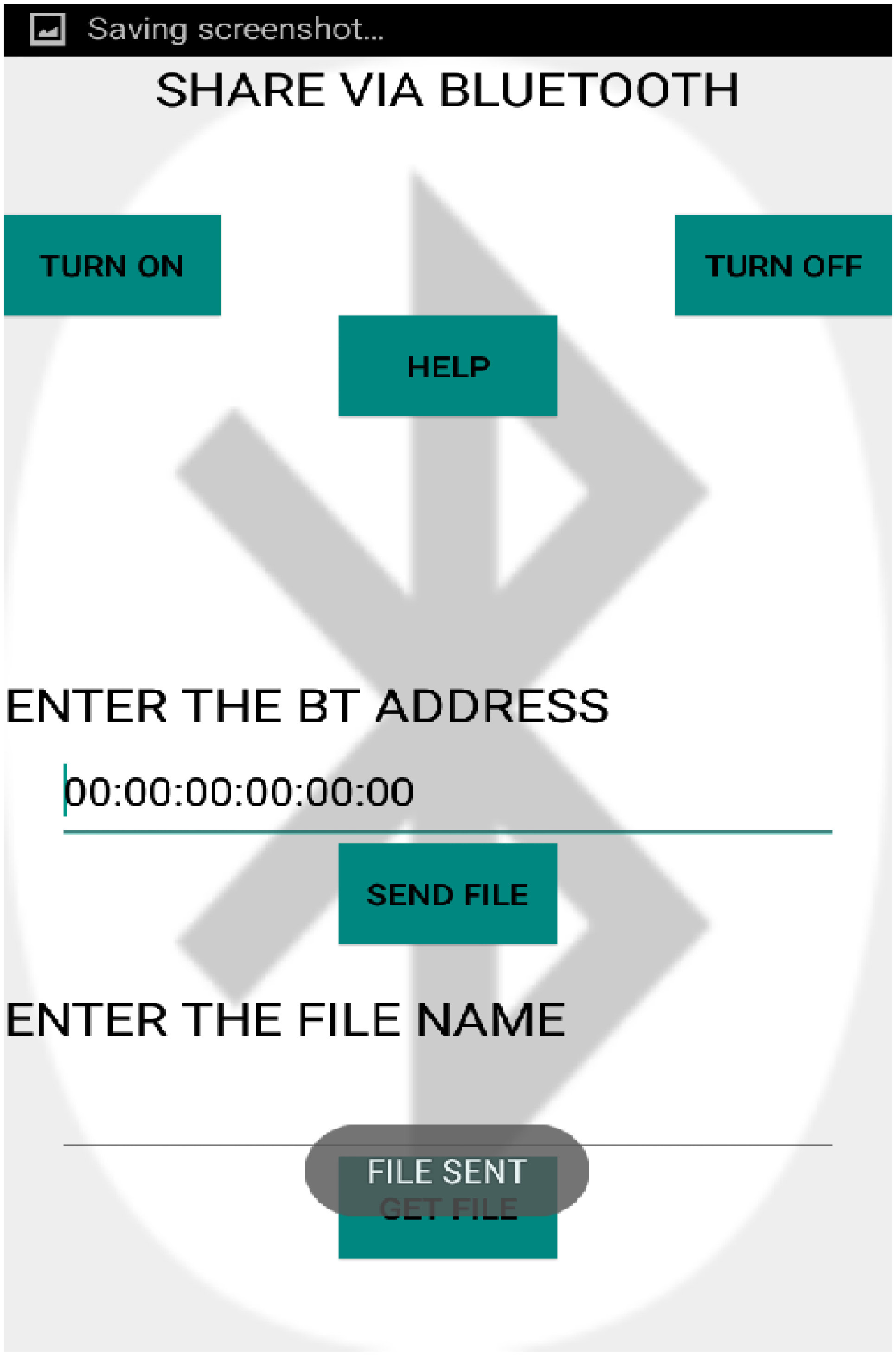


FIGURE 29:FILE GET SUCCESS

On Successful sent of Report from Patient device on GET Operation success Toast will be shown.

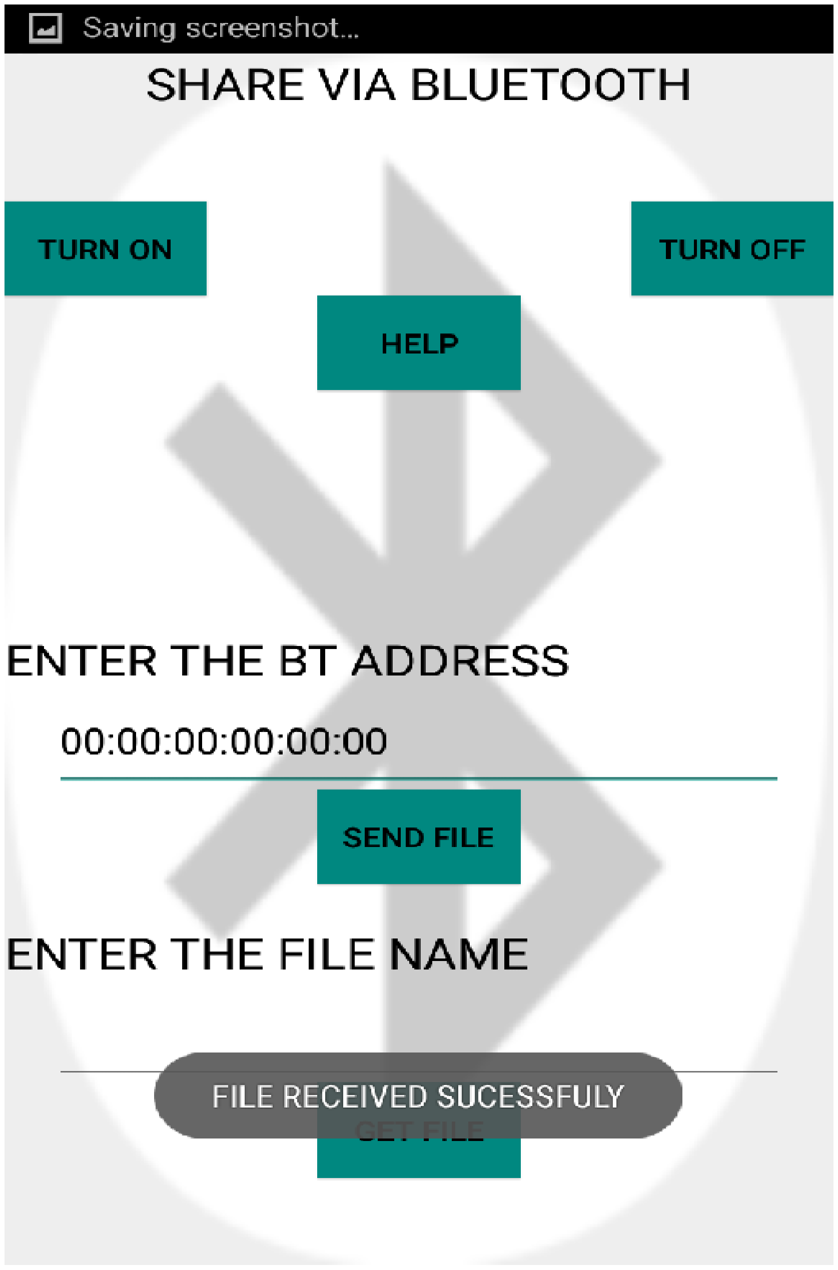


FIGURE 30: GET OPEARTION SUCCESS

On GET Operation Success toast of File received successfully will be shown on Medical practitioner device.

Chapter-8

CONCLUSION

Introduction

Millions of people suffer from medical conditions that should be made known to healthcare practitioners prior to treatment. Paramedics and emergency room doctors cannot provide optimal care without sufficient knowledge of a patient’s medical history. Lacking patient information such as allergies, current prescriptions, and pre-existing conditions, medical professionals are often forced to either delay treatment or rely on instincts

In this thesis work, Bluetooth Automation App has been developed which enables medical practitioner and Patient to exchange MEDICAL information wirelessly without any complexity and minimal user interaction. Health care system as developed in this thesis allows individuals to store personal medical information using portable electronic devices.

The Health Care System architecture is designed to allow an individual to carry personal electronic medical information on a wireless handheld device such as a cell phone, PDA, or enhanced smart card. Medical workers can obtain this information wirelessly using handheld devices, desktop computers, network access points, etc. In this way, patients become an active component of the medical information infrastructure, facilitating better delivery of medical care.

Results

Medical records of patient has been successfully transferred from Medical practitioner device to patient device using BT automation application and HCE also medical reports of patient from patient device has been successfully retrieved at doctor device using BT automation application.
These operations required no Human interference and BT pairing and Transfer done automatically with help of Host Card Emulation.

Chapter-9

FUTURE WORK

Health Care system involving Bluetooth automation has ensured that Health Data can be efficiently exchanged between medical practitioner and patient without complex infrastructure and minimal user interaction.

Medical information is highly confidential. All sensitive data must be secure from unauthorized Access. Initial prototypes include security features such as secure key-exchange and strong data encryption as well as multiple levels of access to information via password protection.

In Bluetooth automation RFCOMM communication has been done directly and standard OBEX protocol of data transfer has not been used. Also BT pairing procedure has been bypassed which generates LINK KEY which is used for encrypting data transferred over Bluetooth.
Hence work will be done in future to develop efficient encryption scheme to be used for transferring data over BT automation.

One of the primary vulnerabilities inherent in wireless communication is the threat of eavesdropping. Experts suggest that this is a significant danger, even if two devices are communicating only at short range. Bluetooth provides a limited number of low-level security measures to prevent eavesdropping, and more advanced security procedures (e.g., public keys, certificates) can be implemented within the Health care System to provide greater protection. To authenticate Patient and medical practitioner devices a mechanism will be developed to store predefined keys on both sides which will be used in authenticating devices.

Feasibility study of using BT Low energy in BT Health care system will be done as BT Low energy is inherently designed to avoid pairing process still it has its own security protocol SMP to ensure data transferred securely over air.

Chapter-10

SOURCE CODE

MainActivity.java: This file is the main application FILE which will have implementation of all APPLICATION INTERFACE and it will create CLIENT and SERVER Thread. It will have implementation of event handling of CLIENT and SEVER Thread.

```
package com.example.ronakb3.share;

import android.app.Activity;
import android.app.AlertDialog;
import android.app.ProgressDialog;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.content.Context;
import android.content.DialogInterface;
import android.content.Intent;
import android.database.Cursor;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.os.Handler;
import android.os.Message;
import android.provider.MediaStore;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.Toast;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.net.URISyntaxException;
import java.util.Random;

public class MainActivity extends Activity {
private static final String TAG = "BTPHOTO/MainActivity";
private Spinner deviceSpinner;
private ProgressDialog progressDialog;
    BluetoothAdapter BA = BluetoothAdapter.getDefaultAdapter();
    BluetoothDevice device;
    EditText edit;
    EditText name;
    String addr;
static String command;
    String FileName;
    String createfile;
```

```
boolean is_GET;
boolean is_Server;
private static final int FILE_SELECT_CODE = 0;

    @Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);


        MainApplication.clientHandler = new Handler() {
            @Override
public void handleMessage(Message message) {
Log.i(TAG, "Value of COMMAND,GETFLE are " + is_GET + is_Server);
switch (message.what) {
case MessageType.READY_FOR_DATA: {
if (is_GET) {
Log.i(TAG, "GET OPERATION: Preseed Need to SEND Command Here");
name = (EditText) findViewById(R.id.editFile);
command = name.getText().toString();
Log.i(TAG, "FILE NAME TO RECEIVE IS " + command);
                            Message messagedata = new Message();
                            messagedata.obj = command.getBytes();
                            MainApplication.serverThread.is_file_cmd =
true;
                            is_GET = false;
MainApplication.clientThread.incomingHandler.sendMessage(messagedata);
                        } else if
(MainApplication.clientThread.server) {  //use clientthread.server if
this does not work
Log.i(TAG, "GET OPERATION: FILE command recived from Remote Need to
SEND file");
Log.v(TAG, "FILE NAME RECIVED IN CLIENT THREAD IS" + FileName);
Log.v(TAG, "FILE NAME RECIVED IN CLIENT THREAD USING THread Var." +
MainApplication.clientThread.RecvdFile);
readFile(MainApplication.clientThread.RecvdFile);//Get The File From
Fixed Folder Here.
                            is_Server = false;
                        } else {
Log.i(TAG, "SEND OPERATION: Need to SEND NORMAL File");
                            Intent intent = new
Intent(Intent.ACTION_GET_CONTENT);
intent.setType("*/*");
intent.addCategory(Intent.CATEGORY_OPENABLE);

try {
startActivityForResult(
Intent.createChooser(intent, "Select a File to Upload"),
                                        FILE_SELECT_CODE);
                            } catch
(android.content.ActivityNotFoundException ex) {
```

```
                                // Potentially direct the user to the
Market with a Dialog
Log.d("asa", "asa");
                            }
                        }
break;

                    }

case MessageType.COULD_NOT_CONNECT: {
Toast.makeText(MainActivity.this, "COULD NOT CONNECT TO DEVICE",
Toast.LENGTH_SHORT).show();
break;
                    }

case MessageType.SENDING_DATA: {
progressDialog = new ProgressDialog(MainActivity.this);
progressDialog.setMessage("SENDING FILE...");
progressDialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);
progressDialog.show();
break;
                    }
case MessageType.SENDING_COMMAND: {
Toast.makeText(MainActivity.this, "SENT COMMAND TO GET-FILE",
Toast.LENGTH_SHORT).show();
break;
                    }
case MessageType.DATA_SENT_OK: {
if (progressDialog != null) {
progressDialog.dismiss();
progressDialog = null;
                        }

Toast.makeText(MainActivity.this, "FILE SENT SUCESSFULLY",
Toast.LENGTH_SHORT).show();
finish();
                        Intent i = new Intent(getApplicationContext(),
MainActivity.class);
startActivity(i);
break;
                    }

case MessageType.DIGEST_DID_NOT_MATCH: {

Toast.makeText(MainActivity.this, "FILE SENT",
Toast.LENGTH_SHORT).show();
finish();
                        Intent i = new Intent(getApplicationContext(),
MainActivity.class);
startActivity(i);
break;
                    }
```

```
case MessageType.FILE_NOT_PRESENT: {

Toast.makeText(MainActivity.this, "REQUESTED FILE NOT PRESENT",
Toast.LENGTH_SHORT).show();
finish();
                        Intent i = new Intent(getApplicationContext(),
MainActivity.class);
startActivity(i);
break;
                    }
                }
            }


        };
        MainApplication.serverHandler = new Handler() {
            @Override
public void handleMessage(Message message) {
switch (message.what) {
case MessageType.DATA_RECEIVED: {
if (progressDialog != null) {
progressDialog.dismiss();
progressDialog = null;
                        }

                        BitmapFactory.Options options = new
BitmapFactory.Options();
                        options.inSampleSize = 2;
                        Bitmap image =
BitmapFactory.decodeByteArray(((byte[]) message.obj), 0, ((byte[])
message.obj).length, options);
                        String root =
Environment.getExternalStorageDirectory().toString();
                        File myDir = new File(root + "/req_images");
myDir.mkdirs();
                        Random generator = new Random();
int n = 10000;
                        File file;
                        n = generator.nextInt(n);

if(MainApplication.serverThread.is_file_cmd) {
Log.v(TAG, "Parse the file name from PATH: " + command);
getfileName(command);//"Image-" + n + ".jpg";
Log.v(TAG, "File Name to Create  is: " + createfile);
file = new File(myDir, createfile);
Log.i(TAG, "" + file);

MainApplication.serverThread.is_file_cmd=false;
                        }
else {
                            String fname = "Image-" + n + ".jpg";
Log.v(TAG, "File Name to Create  is: " + fname);
file = new File(myDir, fname);
                        }
Log.i(TAG, "" + file);
if (file.exists())
file.delete();
try {
                            FileOutputStream out = new
FileOutputStream(file);
image.compress(Bitmap.CompressFormat.JPEG, 90, out);
out.flush();
out.close();
                        } catch (Exception e) {
e.printStackTrace();
                        }
Toast.makeText(MainActivity.this, "FILE RECEIVED SUCESSFULY",
Toast.LENGTH_SHORT).show();
finish();
                        Intent i = new
Intent(getApplicationContext(),MainActivity.class);
startActivity(i);

break;
                    }
     case MessageType.DIGEST_DID_NOT_MATCH:
                    {
Toast.makeText(MainActivity.this, "FILE RECEIVED",
Toast.LENGTH_SHORT).show();
break;
                    }

case MessageType.DATA_PROGRESS_UPDATE: {
                        // some kind of update
                        MainApplication.progressData = (ProgressData)
message.obj;
double pctRemaining = 100 - (((double)
MainApplication.progressData.remainingSize /
MainApplication.progressData.totalSize) * 100);
                      /*  if (progressDialog == null) {
progressDialog = new ProgressDialog(MainActivity.this);
progressDialog.setMessage("Receiving photo...");
progressDialog.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
progressDialog.setProgress(0);
progressDialog.setMax(100);
progressDialog.show();
                        }*/
                       // progressDialog.setProgress((int)
Math.floor(pctRemaining));
break;
                    }

case MessageType.INVALID_HEADER: {
```

```
Toast.makeText(MainActivity.this, "Photo was sent, but the header was
formatted incorrectly", Toast.LENGTH_SHORT).show();
break;
                    }
case MessageType.CMD_RECEIVED: {
Log.i(TAG, "INSIDE SERVER HANDLER: CMD RECIVED");
if (MainApplication.clientThread != null) {
MainApplication.clientThread.cancel();
                        }
                        is_Server=true;
byte[] payload = (byte[]) message.obj;
                        FileName=new String (payload);
                       // MainApplication.clientThread.RecvdFile=new
String (payload); //BB_CH_25
Log.v(TAG, "FILE NAME RECIVED IN SERVER THREAD IS"+FileName);
device = BA.getRemoteDevice(ServerThread.remoteaddr);
if (MainApplication.clientThread != null) {
MainApplication.clientThread.cancel();
                        }
                        MainApplication.clientThread = new
ClientThread(device, MainApplication.clientHandler, false,true);
                        MainApplication.clientThread.RecvdFile=new
String (payload); //BB_CH_25
MainApplication.clientThread.start();
break;
                    }
                }
            }
        };

if (MainApplication.pairedDevices != null) {
if (MainApplication.serverThread == null) {
Log.v(TAG, "Starting server thread.  Able to accept photos.");
                MainApplication.serverThread = new
ServerThread(MainApplication.adapter, MainApplication.serverHandler);
MainApplication.serverThread.start();
            }
        }

if (MainApplication.pairedDevices != null)
        {
edit = (EditText) findViewById(R.id.edit);
edit.setText("00:00:00:00:00:00");
addr = edit.getText().toString();
            Button clientButton = (Button)
findViewById(R.id.clientButton);
            Button serverButton = (Button)
findViewById(R.id.serverButton);
clientButton.setOnClickListener(new View.OnClickListener() {
            @Override
public void onClick(View view) {
edit = (EditText) findViewById(R.id.edit);
```

```
addr = edit.getText().toString();
device = BA.getRemoteDevice(addr);
                is_GET=false;

Log.v(TAG, "Starting client thread in SEND");
if (MainApplication.clientThread != null) {
MainApplication.clientThread.cancel();
                }
                MainApplication.clientThread = new
ClientThread(device, MainApplication.clientHandler, false, false);
MainApplication.clientThread.start();
            }
        });
serverButton.setOnClickListener(new View.OnClickListener() {
                @Override
public void onClick(View view) {
edit = (EditText) findViewById(R.id.edit);
name = (EditText) findViewById(R.id.editFile);
addr = edit.getText().toString();
device = BA.getRemoteDevice(addr);
                    is_GET=true;
Log.v(TAG, "Starting client thread in GET");
Log.v(TAG, "FILE NAME ENTERED BY usr is " +
name.getText().toString());
if (MainApplication.clientThread != null) {
MainApplication.clientThread.cancel();
                    }
                    MainApplication.clientThread = new
ClientThread(device, MainApplication.clientHandler,true,false);
MainApplication.clientThread.start();

                }
            });
        } else {
Toast.makeText(this, "Bluetooth is not enabled or supported on this
device", Toast.LENGTH_LONG).show();
        }
    }

    @Override
protected void onStop() {
super.onStop();
if (progressDialog != null) {
progressDialog.dismiss();
progressDialog = null;
        }
    }

    @Override
protected void onResume() {
super.onResume();
    }
    @Override
protected void onActivityResult(int requestCode, int resultCode,
Intent data) {
super.onActivityResult(requestCode, resultCode, data);

switch (requestCode) {
case FILE_SELECT_CODE:
if (resultCode == RESULT_OK) {
                    // Get the Uri of the selected file
                    Uri uri = data.getData();
Log.d(TAG, "File Uri: " + uri.toString());
                    // Get the path
                    String path = null;
try {
path = getPaths(this, uri);

                    } catch (URISyntaxException e) {
e.printStackTrace();
                    }
Log.d(TAG, "File Path: " + path);
                    File file = new File(path);
                    BitmapFactory.Options options = new
BitmapFactory.Options();
                    options.inSampleSize = 2;
                    Bitmap image =
BitmapFactory.decodeFile(file.getAbsolutePath(), options);

                    ByteArrayOutputStream compressedImageStream = new
ByteArrayOutputStream();
image.compress(Bitmap.CompressFormat.JPEG,
MainApplication.IMAGE_QUALITY, compressedImageStream);
byte[] compressedImage = compressedImageStream.toByteArray();
Log.v(TAG, "Compressed image size: " + compressedImage.length);

                    Message message = new Message();
                    message.obj = compressedImage;
MainApplication.clientThread.incomingHandler.sendMessage(message);

                  }
break;
        }
super.onActivityResult(requestCode, resultCode, data);
    }    public String getPaths(Context context, Uri uri) throws
URISyntaxException {

            Cursor cursor = getContentResolver().query(uri, null,
null, null, null);
cursor.moveToFirst();
        String document_id = cursor.getString(0);
        document_id =
document_id.substring(document_id.lastIndexOf(":")+1);
```

```
cursor.close();

cursor = getContentResolver().query(

android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
null, MediaStore.Images.Media._ID + " = ? ", new
String[]{document_id}, null);
cursor.moveToFirst();
        String path =
cursor.getString(cursor.getColumnIndex(MediaStore.Images.Media.DATA));
cursor.close();

return path;
    }

public void readFile(String filename )
    {
Log.v(TAG, "Inside readFile: ");
boolean isfile_present;
byte[] compressedImage =null;
        String root =
Environment.getExternalStorageDirectory().toString();
        //String cmd="Test.jpg";
Log.v(TAG, "FILE NAME received inreaFile" +filename );
        //File myDir = new File(root);
        //Log.v(TAG, "DIRECTORY is" +myDir );
        // myDir.mkdirs();
        File file = new File(root +"/"+ filename);
Log.i(TAG, "" + file);
        isfile_present=file.exists();
if(isfile_present){
            BitmapFactory.Options options = new
BitmapFactory.Options();
            options.inSampleSize = 2;
            Bitmap image =
BitmapFactory.decodeFile(file.getAbsolutePath(), options);

            ByteArrayOutputStream compressedImageStream = new
ByteArrayOutputStream();
image.compress(Bitmap.CompressFormat.JPEG,
MainApplication.IMAGE_QUALITY, compressedImageStream);
compressedImage = compressedImageStream.toByteArray();
Log.v(TAG, "Compressed image size: " + compressedImage.length);
        }
        Message message = new Message();
        message.obj = compressedImage;
MainApplication.clientThread.incomingHandler.sendMessage(message);
    }

public void getfileName(String Path)
    {
Log.v(TAG, "Inside Fuction getfileName Path is: " +Path );
```

```
createfile =Path.substring(Path.lastIndexOf("/") + 1);
if(createfile==null)
createfile=Path;
Log.v(TAG, "File Name from Path is: " + createfile);
    }

public void on(View view){
if (!BA.isEnabled())
        {

BA.enable();
Toast.makeText(getApplicationContext(),"Turned on"
,Toast.LENGTH_LONG).show();
        }
else
        {
Toast.makeText(getApplicationContext(),"Already on",
                    Toast.LENGTH_LONG).show();
        }
    }

public void off(View view){
BA.disable();
Toast.makeText(getApplicationContext(),"Turned off" ,
                Toast.LENGTH_LONG).show();
    }
public void help(View view)
    {

new AlertDialog.Builder(this)
                .setTitle("HELP")
                .setMessage("MAKE SURE THE BLUETOOTH OF REMOTE DEVICE
IS ON")
                .setPositiveButton(android.R.string.yes, new
DialogInterface.OnClickListener() {
public void onClick(DialogInterface dialog, int which) {
dialog.cancel();
                    }
                })
                .setIcon(android.R.drawable.ic_dialog_alert)
                .show();
    }
}
```

ClientThread.java: This file be implement Client thread for socket communication.
It will maintain different STATE of client and handling of different EVENTS that Client thread can receive

//CODE For CLIENT THREAD

```
package com.example.ronakb3.share;

import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import android.util.Log;
import android.widget.Toast;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.UUID;
import java.lang.reflect.Method;

public class ClientThread extends Thread {
private final String TAG = "android-btxfr/ClientThread";
private /*final*/ BluetoothSocket socket;
private final Handler handler;
public Handler incomingHandler;
public boolean get;
public static boolean server;
public static String RecvdFile;
public BluetoothDevice tempdevice;

public ClientThread(BluetoothDevice device, Handler handler, boolean
is_get, boolean is_server) {
        BluetoothSocket tempSocket = null;
        this.handler = handler;
Log.v(TAG, "Inside Client thread GET value is" + is_get);
get=is_get;
server=is_server;
tempdevice=device;
try {
Log.v(TAG, "Bluetooth Address of remote device b4 creating Client
thread is " + device.getAddress());
tempSocket =
device.createInsecureRfcommSocketToServiceRecord(UUID.fromString(Const
ants.UUID_STRING));
if(tempSocket==null)
Log.v(TAG, "Client Socket Creatin FAIL");
```

```
                } catch (Exception e) {
Log.e(TAG, e.toString());
        }
        this.socket = tempSocket;
    }

public void run() {
try {
            //Log.v(TAG, "Opening client socket");
Log.v(TAG, "Opening client socket: BT Address of remote device" +
socket.getRemoteDevice().getAddress());
socket.connect();
Log.v(TAG, "Connection established");

        } catch (IOException ioe) {
try {
Log.e(TAG, ioe.toString());
Log.v(TAG,"Connect FAIL: Trying fallback ...");

                this.socket =(BluetoothSocket)
tempdevice.getClass().getMethod("createInsecureRfcommSocket",
new Class[]{int.class}).invoke(tempdevice,1);
socket.connect();

Log.v(TAG, "Connected in Fall back");
            }
catch (Exception e) {
Log.v(TAG, "Connection Failed in Fall BACK!");
try {
socket.close();
                } catch (IOException ce) {
Log.e(TAG, "Socket close exception: " + ce.toString());
                }
            }
handler.sendEmptyMessage(MessageType.COULD_NOT_CONNECT);


        }

Looper.prepare();

incomingHandler = new Handler(){
            @Override
public void handleMessage(Message message)
            {
Log.v(TAG, "Inside Client Thread Handle Data");
if (message.obj != null) {
Log.v(TAG, "Handle received data to send");
byte[] payload = (byte[]) message.obj;

try {
Log.v(TAG, "TRY:  Handle received data to send");
```

```
                        OutputStream outputStream =
socket.getOutputStream();

                        // Send the header control first

if (get ) {
byte[] bytes = new byte[32];
Log.v(TAG, "Inside Client GET , need to send one Extra byte");
handler.sendEmptyMessage(MessageType.SENDING_COMMAND);
outputStream.write(0x21);
                           // String cmd = "TEST.jpg";// logic to
convert to 32 byte command file name
                           // bytes = cmd.getBytes();
bytes=payload;
outputStream.write(payload);
get=false;
                        } else {
handler.sendEmptyMessage(MessageType.SENDING_DATA);
outputStream.write(0x00);
outputStream.write(Constants.HEADER_MSB);
outputStream.write(Constants.HEADER_LSB);
Log.v(TAG, "Inside CLIENT SEND ");
Log.v(TAG, "TRY:  Write HEADER to Scket data to send");
                            // write size
outputStream.write(Utils.intToByteArray(payload.length));
Log.v(TAG, "TRY:  Write Digest to Scket data to send");
                            // write digest
byte[] digest = Utils.getDigest(payload);
outputStream.write(digest);
Log.v(TAG, "TRY:  Write payload to Scket data to send");

                            // now write the data
outputStream.write(payload);
outputStream.flush();

Log.v(TAG, "Data sent.  Waiting for return digest as confirmation");
                            InputStream inputStream =
socket.getInputStream();
byte[] incomingDigest = new byte[16];
int incomingIndex = 0;

try {
while (true) {
byte[] header = new byte[1];
inputStream.read(header, 0, 1);
incomingDigest[incomingIndex++] = header[0];
if (incomingIndex == 16) {
if (Utils.digestMatch(payload, incomingDigest)) {
Log.v(TAG, "Digest matched OK.  Data was received OK.");
ClientThread.this.handler.sendEmptyMessage(MessageType.DATA_SENT_OK);
                                        } else {
```

```
Log.e(TAG, "Digest did not match.  Might want to resend.");
ClientThread.this.handler.sendEmptyMessage(MessageType.DIGEST_DID_NOT_
MATCH);
                                        }

break;
                                    }
                                }
                            } catch (Exception ex) {
Log.e(TAG, ex.toString());
                            }
                        }
Log.v(TAG, "Closing the client socket.");
socket.close();

                    } catch (Exception e) {
Log.e(TAG, e.toString());
                    }

                }
else{

ClientThread.this.handler.sendEmptyMessage(MessageType.FILE_NOT_PRESEN
T);
Log.v(TAG, "Closing the client socket in ELSE condition.");
try {
socket.close();
}catch(IOException e)
                    {
Log.e(TAG, "Socket Close faile in else" + e.toString());
                    }
                }
            }
        };

handler.sendEmptyMessage(MessageType.READY_FOR_DATA);
Looper.loop();
    }

public void cancel() {
try {
if (socket.isConnected()) {
socket.close();
            }
        } catch (Exception e) {
Log.e(TAG, e.toString());
        }
    }

}
```

Serverthread.java: `This file will implement server Thread functionlity`

```
//code
package com.example.ronakb3.share;

import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothServerSocket;
import android.bluetooth.BluetoothSocket;
import android.os.Handler;
import android.util.Log;

import java.io.IOException;
import java.util.UUID;

public class ServerThread extends Thread {
private final String TAG = "android-btxfr/ServerThread";
private final BluetoothServerSocket serverSocket;
private Handler handler;
public static String remoteaddr;
public static boolean is_file_cmd=false;
public ServerThread(BluetoothAdapter adapter, Handler handler) {
        this.handler = handler;
        BluetoothServerSocket tempSocket = null;
try {
tempSocket =
adapter.listenUsingInsecureRfcommWithServiceRecord(Constants.NAME,
UUID.fromString(Constants.UUID_STRING));
        } catch (IOException ioe) {
Log.e(TAG, ioe.toString());
        }
serverSocket = tempSocket;
    }

public void run() {
        BluetoothSocket socket = null;
if (serverSocket == null)
        {
Log.d(TAG, "Server socket is null - something went wrong with
Bluetooth stack initialization?");
return;
        }
while (true) {
try {
Log.v(TAG, "Opening new server socket");

socket = serverSocket.accept();
remoteaddr= socket.getRemoteDevice().getAddress();
Log.d(TAG, "SERVER SIDE: Bluetoth address of remote is" +remoteaddr);
try {
```

```
Log.v(TAG, "Got connection from client.  Spawning new data transfer
thread.");
                    DataTransferThread dataTransferThread = new
DataTransferThread(socket, handler);
dataTransferThread.start();
                } catch (Exception e) {
Log.e(TAG, e.toString());
                }

            } catch (IOException ioe) {
Log.v(TAG, "Server socket was closed - likely due to cancel method on
server thread");
break;
            }
        }
    }

public void cancel() {
try {
Log.v(TAG, "Trying to close the server socket");
serverSocket.close();
        } catch (Exception e) {
Log.e(TAG, e.toString());
        }
    }
}
```

DataTransferThread.java: This file will implement data transfer thread which will do data handling of all received data i.e. data received at Server side.

// code for data transfer thread

```
package com.example.ronakb3.share;

import android.bluetooth.BluetoothSocket;
import android.os.Handler;
import android.os.Message;
import android.util.Log;

import java.io.ByteArrayOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Arrays;

class DataTransferThread extends Thread {
private final String TAG = "android-btxfr/DataTransferThread";
private final BluetoothSocket socket;
private Handler handler;

public DataTransferThread(BluetoothSocket socket, Handler handler) {
        this.socket = socket;
        this.handler = handler;
    }

public void run() {
try {
Log.d("RONAK","24");

            InputStream inputStream = socket.getInputStream();
boolean waitingForHeader = true;
boolean cmd=false;
            ByteArrayOutputStream dataOutputStream = new
ByteArrayOutputStream();
byte[] headerBytes = new byte[23];
byte[] cmdbytes = new byte[32];
byte[] digest = new byte[16];
int headerIndex = 0;
byte[] datacmd = new byte[32];
            ProgressData progressData = new ProgressData();

while (true) {
if (waitingForHeader && !cmd) {
Log.d("RONAK", "25");

byte[] header = new byte[1];
inputStream.read(header, 0, 1);
Log.v(TAG, "Received Command Byte: " + header[0]);
if(header[0]==0x21 && (headerIndex==0) )
                    {
Log.v(TAG, "INSIDE COMMAND CONDITION SERVER: " + header[0]);
cmd=true;
Log.v(TAG, "Header Received.  Now obtaining length");
byte[] buffer = new byte[32];
int bytesRead = inputStream.read(buffer);
Log.v(TAG, "Read cmd" + bytesRead + " bytes into buffer");
dataOutputStream.write(buffer, 0, bytesRead);
datacmd= dataOutputStream.toByteArray();
break;
                    }
if(header[0]==0x0 && (headerIndex==0))
                    {
Log.v(TAG, "INSIDE NO COMMAND CONDITION SERVER: " + header[0]);
cmd=false;
                    }
Log.v(TAG, "Received Header Byte: " + header[0]);
headerBytes[headerIndex++] = header[0];

if (headerIndex == 23)
                    {
```

```
Log.d("RONAK","26");

if ((headerBytes[1] == Constants.HEADER_MSB) && (headerBytes[2] ==
Constants.HEADER_LSB)) {
Log.v(TAG, "Header Received.  Now obtaining length");
byte[] dataSizeBuffer = Arrays.copyOfRange(headerBytes, 3, 7);
                            progressData.totalSize =
Utils.byteArrayToInt(dataSizeBuffer);
                            progressData.remainingSize =
progressData.totalSize;
Log.v(TAG, "Data size: " + progressData.totalSize);
digest = Arrays.copyOfRange(headerBytes, 7, 23);
waitingForHeader = false;
sendProgress(progressData);
                        } else {
Log.e(TAG, "Did not receive correct header.  Closing socket");
socket.close();
handler.sendEmptyMessage(MessageType.INVALID_HEADER);
break;
                        }
                    }

                } else
                {
                    // Read the data from the stream in chunks
byte[] buffer = new byte[Constants.CHUNK_SIZE];
Log.v(TAG, "Waiting for data.  Expecting " +
progressData.remainingSize + " more bytes.");
int bytesRead = inputStream.read(buffer);
Log.v(TAG, "Read " + bytesRead + " bytes into buffer");
dataOutputStream.write(buffer, 0, bytesRead);
                    progressData.remainingSize -= bytesRead;
sendProgress(progressData);

if (progressData.remainingSize <= 0) {
Log.v(TAG, "Expected data has been received.");
break;
                    }
                }
            }

            // check the integrity of the data
if(cmd)
            {
                Message message = new Message();
                message.obj = datacmd;
                String val = new String (datacmd);
Log.v(TAG, "Inside CMD recived VALUE of CMD is"+ val);
                message.what = MessageType.CMD_RECEIVED;
handler.sendMessage(message);
cmd=false;
```

```
            }
else {
final byte[] data = dataOutputStream.toByteArray();
if (Utils.digestMatch(data, digest)) {
Log.v(TAG, "Digest matches OK.");
                    Message message = new Message();
                    message.obj = data;
                    message.what = MessageType.DATA_RECEIVED;
handler.sendMessage(message);

                    // Send the digest back to the client as a
confirmation
Log.v(TAG, "Sending back digest for confirmation");
                    OutputStream outputStream =
socket.getOutputStream();
outputStream.write(digest);

                } else {
Log.e(TAG, "Digest did not match.  Corrupt transfer?");
handler.sendEmptyMessage(MessageType.DIGEST_DID_NOT_MATCH);
                }
Log.v(TAG, "Closing server socket");
socket.close();
            }
           // Log.v(TAG, "Closing server socket");
            //socket.close();

        } catch (Exception ex) {
Log.d(TAG, ex.toString());
        }
    }

private void sendProgress(ProgressData progressData) {
        Message message = new Message();
        message.obj = progressData;
        message.what = MessageType.DATA_PROGRESS_UPDATE;
handler.sendMessage(message);
    }
}
```

Util.java: This files have all utility functions defined

//code for utils.java

```
package com.example.ronakb3.share;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.Environment;
import android.os.Message;
import android.util.Log;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.security.MessageDigest;
import java.util.Arrays;

class Utils {
private final static String TAG = "android-btxfr/Utils";

public static byte[] intToByteArray(int a) {
byte[] ret = new byte[4];
ret[3] = (byte) (a & 0xFF);
ret[2] = (byte) ((a >> 8) & 0xFF);
ret[1] = (byte) ((a >> 16) & 0xFF);
ret[0] = (byte) ((a >> 24) & 0xFF);
return ret;
    }

public static int byteArrayToInt(byte[] b) {
return (b[3] & 0xFF) + ((b[2] & 0xFF) << 8) + ((b[1] & 0xFF) << 16) +
((b[0] & 0xFF) << 24);
    }

public static boolean digestMatch(byte[] imageData, byte[] digestData)
{
return Arrays.equals(getDigest(imageData), digestData);
    }

public static byte[] getDigest(byte[] imageData) {
try {
            MessageDigest messageDigest =
MessageDigest.getInstance("MD5");
return messageDigest.digest(imageData);
        } catch (Exception ex) {
Log.e(TAG, ex.toString());
throw new UnsupportedOperationException("MD5 algorithm not available
on this device.");
        }
    }
}
```

Messagetype.java: This file have all messages registered which is handled at CLEINT and SERVER THREAD

//Code for Messagetype.java

```
package com.example.ronakb3.share;


public class MessageType
{
public static final int DATA_SENT_OK = 0x00;
public static final int READY_FOR_DATA = 0x01;
public static final int DATA_RECEIVED = 0x02;
public static final int DATA_PROGRESS_UPDATE = 0x03;
public static final int SENDING_DATA = 0x04;
public static final int CMD_RECEIVED = 0x05;
public static final int FILE_NOT_PRESENT=0x06;
public static final int SENDING_COMMAND=0x07;

public static final int DIGEST_DID_NOT_MATCH = 0x50;
public static final int COULD_NOT_CONNECT = 0x51;
public static final int INVALID_HEADER = 0x52;
}
```

Constants.java: This file have all the constants values defined

// *code for constants.java*

```
package com.example.ronakb3.share;

class Constants
{
protected static final int CHUNK_SIZE = 4192;
protected static final int HEADER_MSB = 0x10;
protected static final int HEADER_LSB = 0x55;
protected static final int REQUEST_BROWSE = 0x55;
protected static final String NAME = "ANDROID-BTXFR";
protected static final String UUID_STRING = "00001101-0000-1000-8000-
00805F9B34FB;//AC";
protected static final int CMD_LEN = 20;
}
```

activity_main.xml: xml file for Application layout

// code for activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
    android:background="@drawable/d"
>

<ImageView
android:layout_width="fill_parent"
android:layout_height="fill_parent"
        android:id="@+id/imageView"
android:layout_gravity="center_vertical|left"
        android:layout_below="@+id/clientButton"
        android:layout_toRightOf="@+id/textView"
        android:layout_toEndOf="@+id/textView" />

<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceLarge"
android:text="ENTER THE BT ADDRESS"
        android:id="@+id/textView"
android:layout_centerVertical="true"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
android:textColor="#000000"/>

<EditText
android:layout_width="wrap_content"
android:layout_height="wrap_content"
        android:id="@+id/edit"
android:textColor="#000000"
android:layout_alignParentLeft="true"
android:layout_marginTop="10dp"
android:layout_marginLeft="20dp"
android:layout_marginRight="20dp"
        android:layout_below="@+id/textView"
        android:layout_alignRight="@+id/imageView"
        android:layout_alignEnd="@+id/imageView" />

<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="SEND FILE"
        android:id="@+id/clientButton"
```

```
android:layout_gravity="center"
        android:layout_below="@+id/edit"
android:layout_centerHorizontal="true"
android:textColor="#000000"
android:background="#008880"
         />


<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceLarge"
android:layout_marginTop="20dp"
android:text="ENTER THE FILE NAME"
        android:id="@+id/File"
android:textColor="#000000"
        android:layout_alignTop="@+id/imageView"
                                                      />

<EditText
android:layout_width="wrap_content"
android:layout_height="wrap_content"
        android:id="@+id/editFile"
android:textColor="#000000"
        android:layout_below="@+id/File"
android:layout_alignParentLeft="true"
android:layout_marginTop="10dp"
android:layout_alignParentStart="true"
android:layout_marginRight="20dip"
android:layout_marginLeft="20dip"
        android:layout_alignRight="@+id/imageView"
        android:layout_alignEnd="@+id/imageView" />

<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="GET FILE"
        android:id="@+id/serverButton"
android:layout_gravity="center"
android:layout_centerHorizontal="true"
android:textColor="#000000"
android:background="#008880"
        android:layout_below="@+id/editFile"
   />
<TextView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:textAppearance="?android:attr/textAppearanceLarge"
android:text="SHARE VIA BLUETOOTH"
        android:id="@+id/textView2"
android:layout_alignParentTop="true"
android:layout_centerHorizontal="true"
```

```
android:textColor="#000000"/>

<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="TURN ON"
        android:id="@+id/button"
android:layout_alignParentLeft="true"
android:layout_alignParentStart="true"
android:layout_marginTop="75dp"
android:textColor="#000000"
android:background="#008880"
android:onClick="on"/>

<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="TURN OFF"
        android:id="@+id/button2"
        android:layout_alignTop="@+id/button"
        android:layout_alignRight="@+id/imageView"
        android:layout_alignEnd="@+id/imageView"
android:textColor="#000000"
android:background="#008880"
android:onClick="off"/>

<Button
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="HELP"
        android:id="@+id/button3"
android:textColor="#000000"
android:background="#008880"
android:onClick="help"
        android:layout_below="@+id/button"
android:layout_centerHorizontal="true"
android:elegantTextHeight="false" />

</RelativeLayout>
```

REFERENCES

1. ERIC S. HALL, DAVID K. VAWDREY, CHARLES D. KNUTSON, and JAMES K ARCHIBALD**“Enabling Remote Accessto Personal ElectronicMedical Records”** IEEE ENGINEERING IN MEDICINE AND BIOLOGY MAGAZINE June 2003.

2. Divyashikha, et al.”**Portable computing device based securemedical records management**”, filed in India, Application#1313/DEL/2015, 12 December 2015.

3. Sethia, Divyashikha, et al. "**NFC based secure mobile healthcare system."***Communication Systems and Networks (COMSNETS), 2014 Sixth International Conference on*. IEEE, 2014.

4. Yujin Noishiki Hidetoshi Yokota Akira Idou et al “**Design and Implementation of Ad-hoc Communication and Application on Mobile Phone Terminals”** KDDI R&D Laboratories, Inc. 2-1-15 Ohara, Fujimino-Shi, Saitama

5. Nikita Mahajan *et al*, **Design of Chatting Application Based on Android Bluetooth** International Journal of Computer Science and Mobile Computing, Vol.3 Issue.3, March-2014,

6. Juha T. Vainio et al **“Bluetooth Security”** Helsinki University of Technology 2000-05-25

7. Saparkhojayev, Nurbek, et al. **"NFC-enabled access control and management system."** Web and Open Access to Learning (ICWOAL), 2014 International Conference on. IEEE, 2014.

8. The Bluetooth Special Interest Group.Bluetooth Specification Core v4.0.(2009-02).Http://www.bluetooth.org.

9. Host Based Card Emulation.
https://developer.android.com/guide/topics/connectivity/nfc/hce.html

10. Host card emulation
http://en.wikipedia.org/wiki/Host_card_emulation

11. Android Developers,// http://developer.android.com/index.html