Linnæus University
School of Computer Science, Physics and Mathematics

Master Degree Project

Ad-hoc Data Transfer for Android Devices

Author: Emre Baykal
Date: 2013-03-17
Subject: Ad-hoc Data Transfer for Android Devices
Level: Master
Course code: 5DV00E

Abstract

With the introduction of smart phones and mobile devices, the rate of technology usage has became widespread. It is very likely to see anyone using mobile device on the street, in the bus, at work, at school or even at home no matter if there is a fully functional computer nearby. Particularly, the usage of smart phones is not just limited to basic communication; they are being used as a technological gadget in our daily lives for years. We can easily reach such applications on different fields e.g. image processing, audio tuning, video editing, voice recognition. With the help of these smart applications we are able to interact with our environment much faster, and consequently make life easier; relatively for sure.

Apart from mobile phone industry, sharing and staying connected to our environment has become very popular. A person who has a mobile phones with latest technology desires to communicate with friends, share feelings instantly, and of course meet new people who understand him well.

In this context, technology came to the point where needs of modern human overlaps todays technology. This study aims to provide a basic application to users who has got those needs in terms of sharing and communicating, for maintaining easy and costless solution.

This study currently offers prototype application to this problem using WiFi Direct technology, and intends to be pioneer of its field in consequence of existing small numbers of related applications in the market.

As a consequence of this study a basic application is developed where user can discover peers, transfer images and audio data. Connection is established through peer-to-peer protocol and any possible need of internet connectivity is therefore removed.

With the help of the application, and capabilities of WiFi Direct, users can share files, images and transfer audio in a close range.

Keywords: ad-hoc network, WiFi Direct, P2P, sharing, communicating, file transfer, audio transfer, Android.

Contents

List of Tables

List of Figures

Glossary

3G 3rd generation of mobile telecommunications technology.

ADT Android Development Tool.

API Application Programming Interface.

ARM Advanced RISC Machine.

ICS Ice Cream Sandwich.

IDE Integrated Development Environment.

iOS iPhone Operating System.

IP Internet Protocol.

IPv4 Internet Protocol version 4.

JB Jelly Bean.

JIT Just-in-time.

LAN Local Area Network.

MAC Media Access Control.

P2P Peer-to-peer.

SDK Software Development Kit.

SMS Short Message Service.

UI User Interface.

URI Uniform Resource Identifier.

VM Virtual Machine.

WLAN Wireless Local Area Network.

1 Introduction

This chapter gives the overview of the report and provides fundamental aspects of the study such as describing problem and goal. The chapter advances by giving restrictions of the study. Additionally, the method and structure of the report is summarized in order to provide a flow of the report.

1.1 Background

There is a large variety of mobile applications nowadays. They range from games to social media. Surveys show that games were the most frequently downloaded application by all users, downloaded by 65% of past-30-day smart phone downloaders and 59% of past-30-day feature phone downloaders. (Nielsen Company, 2010)

Although there is a big interest on games, social media applications are also highly rated among downloaders. This is most certainly because of the fact that people need to show their identity to their environment. There are other reasons like "Connections" and "Community" as a reason of joining social networking sites (Shama Kabani, 2010) and using social media applications. In addition to that, *Sharing* also affect people to keep in touch with their environment. We share our pictures with friends, share our videos as a proof of our identity, and share music that we like in order to give impression to people about ourselves.

Sharing and *Communicating* are essential for people and this should be represented in the field of technology. In this context, instant messaging and file sharing applications are the key factors. With the selection of related applications, what we really need in theory is a data connection in the future. We can make and receive free calls using many applications. On the other hand, free messaging applications are for the first time ever putting pressure on the classic SMS paid text model and we share content in all sorts of new ways via a huge variety of applications. (Simple Zesty, 2011)

It is obvious that smart phones need internet connection in order to maintain all the systems mentioned above. Connecting to other people, particularly a person, can only be ensured by connecting to the 'world'. Therefore, there is certainly a need of 3G or network access for all mobile devices and smart phones.

3G is a set of standards used for mobile devices and mobile telecommunication services and networks that comply with the International Mobile Telecommunications-2000 (IMT-2000) specifications by the International Telecommunication Union. (Clint Smith, Daniel Collins 2000, pp.136) It is used in wireless telephony, mobile internet access, video calls and mobile TV.

However, although it seems to be fairly easy and cheap, thanks to developing technology, establishing network connection among smart phones may sometimes be expensive and less effective. For this reason, a new solution might be necessary. This is where our study comes into picture.

1.2 Problem & Motivation

Smart phones require 3G or network access points. However, this technology is not always available and sometimes expensive when it comes to sharing or messaging among other people.

Connecting to a network access point or 3G may not always be possible e.g., in a concert hall, factory buildings, tunnels, museums, etc. where the building constructions might block 3G and access points are not available. Therefore, connection with a group of people to exchange data is a problem in this situations.

Above mentioned problems could be solved by establishing ad-hoc network between two or more smart phones using phones' Wi-Fi network.

1.3 Goals & Criteria

Main goal of this study is to develop a solution for smart phones to communicate (voice and data) between each other at a close range without being dependent on 3G or access points.

The goal is met by implementing this technology and creating a **prototype** application concerning this issue. The prototype application should allow transferring voice at least in one direction (Walky-Talky) in a close range (defined by Wi-Fi hardware on devices), between at least one smart phone platform (Android OS). A general protocol should be defined to allow implementation also on other platforms (iOS, Windows Phone, etc.)

1.4 Restrictions

This study focuses on developing ad-hoc network and establishing this connection only between Android OS devices; in fact the devices that have ICS 4.0 or later releases. Development of prototype application has started with ICS and concluded with JB 4.1. Theoretically, the work is not limited to a single programming language; however, since WiFi Direct technology first came out to smart phones with Android, the study is carried out entirely in Java. Nonetheless, the design of the application can be used on other platforms and also in other programming languages during the development phase.

Furthermore, this work is performed and evaluated as a **prototype** application and therefore it is aimed to build a base product for future developers. In this case, there are certainly some limitations and shortcomings in terms of functionality. This is discussed again as a future work in Section 4.3 and deficiencies is shown to reader along with possible solutions.

1.5 Method

The study starts with collecting necessary information about possible technologies. Afterwards, overall explanation is given for a suitable and selected technology. This section consists more reading as well as understanding the background.

Following the theoretical outline, method and mechanism of particular technology is covered. It is aimed to prepare the reader and provide core description in order to clarify the theory.

Finally, the solution and design is covered and provided to reader. It is expected here to create convincing image of overall study.

All ambiguities, shortcomings and insufficiencies are directed to discussion part and presented to reader for further investigation.

1.6 Structure of the Report

Report consists of five sections starting with Section 1 which gives brief information about overview of the study.

Section 2 is mostly about theory and method description where reader can get solid background towards suitable technology. Tools and structure of the solution finalizes this chapter.

Section 3 consists overall view of software architecture. Also, solution and implementation ideas are covered in this chapter.

Finally, Section 4 comes where reader can find answers to the problems and can consider suggested solutions as a part of discussion part. Future work and further investigation can also be found here.

1.7 Acknowledgements

I would like to thank Rüdiger Lincke for supervising my work and most importantly giving me the freedom to come up with my own solution when it comes to make decisions. Without his contribution and help, I would not see myself as a promising Android developer. I also would like to thank Mathias Hedenborg at my department DFM at Linnaeus University in Växjö for his guidance and feedback.

2 Theory

In order to become familiar with the method and the technology, it is necessary to understand the basics of the whole system. We start by explaining the background as well as current technology, particularly the information regarding the latest protocol called WiFi Direct.

In the next section, motivation and tools are introduced. Overview of Android Operating System and its current releases are described in order to provide a solid background. This is followed by presenting the development tool, Eclipse, and collection of API libraries, Android SDK.

Finally, chapter ends with overall structure of the solution.

2.1 Background

This project aims to establish a connection between two devices without being dependent on 3G or access point. In this respect, one can come up with a simple approach which can be compared to local area network (LAN) in personal computers.

LAN is a computer network that interconnects computers in a limited area such as a home, school, computer laboratory, or office building using network media (Gary A. Donahue 2007, pp 5). LAN offers high transfer rate in smaller area where there is no need for an internet service. Although it offers many advantages, it has shortcomings too. The more obvious one is certainly its dependency on physical connection, i.e. network cable. In order to solve this issue, wireless local area network (WLAN) technology is introduced was 1990s.

WLAN simply links two or more devices over a wireless method via protocols. It usually provides internet connection and let the device connect to an access point. Types of WLAN can be categorized under three segments. Bridge, wireless distribution system and peer-to-peer. Considering the mechanisms of bridge and wireless distribution system, these are not correlated with the system which this study is interested. For this reason, this project adhere to peer-to-peer system.

From a wireless perspective and its correlation with the projects main interest, peer-to-peer system entirely matches with the purposes of this study such as;

i. connection between **mobile** devices.

ii. without a need for **3G** or **wireless access point**.

Thus, following sections focus on describing the existing approaches and exploring the new generation technologies particularly in smart phones and mobile devices. It should also be noted that the rest of this chapter mostly relies on peer-to-peer system.

2.2 Early Approaches

The term *peer-to-peer* refers to the concept that in a network of peers using appropriate information and communication systems, two or more individuals are able to spontaneously collaborate without necessarily needing central coordination (Schoder and Fischbach, 2003). In contrast to client-server networks, P2P networks promise improved scalability, lower cost ownership, self-organized and decentralized coordination of previously underused or limited resources, greater fault tolerance, and

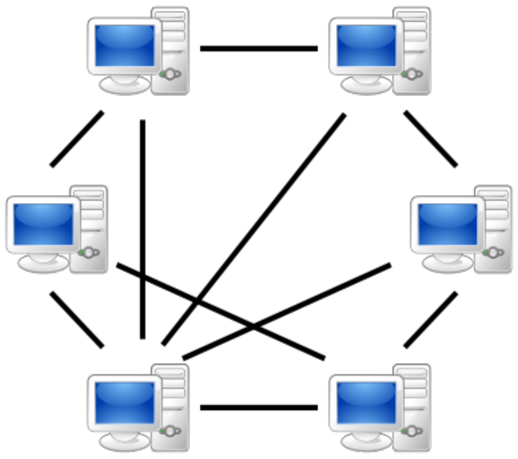


Figure 2.1: A peer-to-peer network

better support for building ad-hoc networks (Schoder and Fischbach, 2005). Figure 2.1 visualizes a very basic peer-to-peer network where each computer is connected to another.

Considering this description, another concept takes place which lets all wireless devices directly communicate with each other. This is called *Wireless ad-hoc network*. A *wireless ad-hoc network* is a decentralized type of wireless network (C K Toh, 2002). This network allows all wireless devices within range of each other to discover and communicate in peer-to-peer fashion without involving central access points (about.com, 2012).

Developers has started working on establishing a successful connection among wireless devices, in particular mobile devices. First approaches were simply about manipulating devices' WiFi protocol and switching them to a basic WiFi hotspot. From this point of view, *tethering* is introduced. Basically *tethering* stands for connecting one device to another. This connection can be done over many different methods such as Bluetooth, physical connection or WLAN. In the context of wireless communication, *tethering* allows sharing device's data connection with other devices to provide network services. Moreover, if *tethering* is done by WiFi, device can act as a wireless access point and thus provide internet connection (only if it has its own network service). Consequently, this allows the device to accept connections coming from other wireless devices. Tethering over Wi-Fi, also known as Personal Hotspot, is available on iOS starting with iOS 4.2.5 (or later) on iPhone 4, 4S, and iPad (3rd generation), certain Windows Phone 7 devices (varies by manufacturer and model) and certain Android phones (varies widely depending on carrier, manufacturer, and software version) (Geek.com, 2010)(Wikipedia, 2012)

However, some service providers do not allow user to use tethering service. This can be achieved by *rooting* the smart phone on Android devices and *jailbreaking* it on iOS devices. These methods allow user to become *superuser* and give extra privileges to change or manipulate device's core settings. Currently, applications such as joikusoft, AllJoyn and The Serval Project provide tethering services despite their shortcomings.

2.3 Current Technology

Certainly, rooting or jailbreaking requires advanced level of programming skills and usually not convenient for daily users. One should grasp the details of the operating system and be necessarily competent on commanding it. This is surely not an easy issue for all users. In this context, this study aims to provide a simpler solution. And this solution can be realized with the latest method, *WiFi Direct*.

This technology simply allows wireless devices to connect each other without any need for access point. No need of access point brings simplicity and functionality as well as independency. This independency removes many requirements such as router and service provider to ensure a connection. Morever, on account of being a licensed product, it is offered in many devices and it saves the user from striving with rooting or jailbreaking.

Finally, compared to Bluetooth, *WiFi Direct* devices can connect over a much greater range and with greater data connection capacities (Informationweek, 2010).

2.4 WiFi Direct

WiFi Direct allows devices to make direct connections to one another quickly and conveniently to do things like print, sync, and share content even when an access point or router is unavailable (Wi-Fi Alliance, 2009). With WiFi Direct network stations can communicate peer-to-peer.

In WiFi P2P groups, devices act either as an access point or a client. Device that runs as an access point is determined as group owner. There are two approaches in order to determine the group owner. First approach is simply a manual selection by the user. Second approach is more authenticated where there needs to be a negotiation between devices. Negotiation is handled by a simple intent value. This intent value depends on various conditions such as power condition, received signal strength or device status whether it is already a group owner or not. Device which has higher intent value accepted as a group owner while the other becomes client.

WiFi Direct devices support same performance profiles of regular Wi-Fi devices. They operate data rates of around 25 Mbps. For devices based on 802.11 a or g, data rates will be about 54 Mbps and a coverage range of about 100 meters. It's network can be one-to-one or one-to-many and it also allows user to connect to a regular network while connected to a WiFi Direct network at the same time. As a result, user can simultaneously use internet over his service provider and P2P network. In context of reaching internet, WiFi Direct network may share internet connectivity with other devices in its WiFi Direct network. In this case, an access point or router will still provide internet connection to the device. Another feature about WiFi Direct is its frequency range. It operates in both 2.4GHz and 5GHz. Finally, Wi-Fi Direct gives devices the ability to discover other devices and limited information about device services prior to association (and before having an IP address). Pre-association discovery improves the user experience where the users will know whether a desired service (e.g. printing) will be available on the Wi-Fi Direct network before connecting. (Wi-Fi Alliance, 2009).

Although security is not an issue for this study, it is necessary to mention the shortcomings of WiFi Direct about security. Jim Rapoza, commenter in InformationWeek says:

> *"Of course, there are potential issues. WiFi Direct uses WPA2 to secure the connection, which is fairly secure but not ironclad. Plus, how devices connect to each other and whether there are measures to prevent unwanted persons or devices from connecting to your WiFi Direct devices will be a big issue. After all, you probably don't want just anyone in the airport to be able to download all the pictures from your vacation from your digital camera."* (InformationWeek, 2010)

It is known that WiFi Direct is secured with WPA2, but it operates separately from the security system and is independent from any infrastructure network. That means, it is not required to have credentials for the infrastructure network to connect to the WiFi Direct network.

In addition to all its feature, WiFi Direct API on Android platform give effective developing ability to all developers. Starting with ICS, Google has introduced new API (level 14) that allows developing WiFi Direct applications. Using this API along with other Android APIs, one can discover peer, connect to other devices and communicate over a fast connection across longer distances than a Bluetooth connection (Android developer, 2011). There are several important packages for creating WiFi Direct application such as:

i. *android.net.wifi.p2p* - Provides classes to create peer-to-peer (P2P) connections with Wi-Fi Direct.

ii. *java.net* - Consists libraries which maintains socket connection

iii. *java.io* - Packages for input/output operation

2.4.1 API Overview

Core operations mostly handled by a class named *WifiP2pManager* under *android.net.wifi.p2p* package. This class provides P2P connectivity as well as discovery, connection setup among devices and query of the list of peers. Other classes are:

- *WifiP2pDevice* - Represents Wi-Fi P2P device.
- *WifiP2pDeviceList* - Represents a Wi-Fi P2P device list.
- *WifiP2pGroup* - Represents Wi-Fi P2P group.
- *WifiP2pInfo* - Represents connection information about a Wi-Fi P2P group.
- *WifiP2pConfig* - Represents a Wi-Fi P2P configuration for setting up a connection.

Additionally, WiFi Direct API requires some permissions in order to establish a connection. These must have configured in Android manifest file. These permissions are:

i. *ACCESS_WIFI_STATE*

ii. *CHANGE_WIFI_STATE*

iii. CHANGE_NETWORK_STATE

iv. ACCESS_NETWORK_STATE

v. INTERNET

2.5 Motivation and Tools

The main goal of this study is to create a simple and functioning system without any extra manipulation in mobile device. From this point of view, Android OS has been selected as it offers suitable APIs. Although Bada OS also introduced its own API for WiFi Direct, Android OS has stronger hand considering its popularity and the fact that most of the smart phones in the market using Android OS.

Eclipse is preferred as the development environment thanks to its contribution for Android development, and more importantly Android's big support as of choosing Eclipse its official development tool.

2.6 Android OS

As it is widely known, Android is a Linux-based operating system led by Google. It is mostly developed for mobile devices to bring simplicity, functionality and efficiency to the market. Android is an open source project and it has a large number of developers writing applications. Developers write applications primarily in Java (Stephen Shankland, 2007) and applications can be downloaded mostly through official online store called Google Play. Currently there is 600,000 applications available on Google Play and so far 20 billion applications downloaded from this store (engadget, 2012).

2.6.1 Overview

Android runs on Linux with libraries and libraries written in C. Dan Morrill, Android Engineer in Google, explained that:

> *"Android is not a specification, or a distribution in the traditional Linux sense. It's not a collection of replaceable components. Android is a chunk of software that you port to a device."* (Dan Morrill, 2010)

Android uses the Dalvik Virtual Machine to run Dalvik Executable code translated from Java bytecode. All standart APIs are defined in terms of classes, interfaces, methods and objects. In terms of hardware platform, ARM architecture is main platform for Android. However, there is also support for x86 architecture.

2.6.2 Architecture

Android runs on Linux under Dalvik VM. Dalvik has a just-in-time compiler where the byte code stored in memory is compiled to a machine code. Byte code can be defined as 'intermediate level'. JIT compiler reads the bytecode in many sections and compiles dynamically in order to run the program faster. Java performs checks on different portions of the code and thus the code is compiled only before it is executed. When it is compiled once, it is cached and set to be ready for later uses.

Essentially, Android system has four different layers. *Application layer* is the top layer where user interacts with the device. It is written in Java and executing

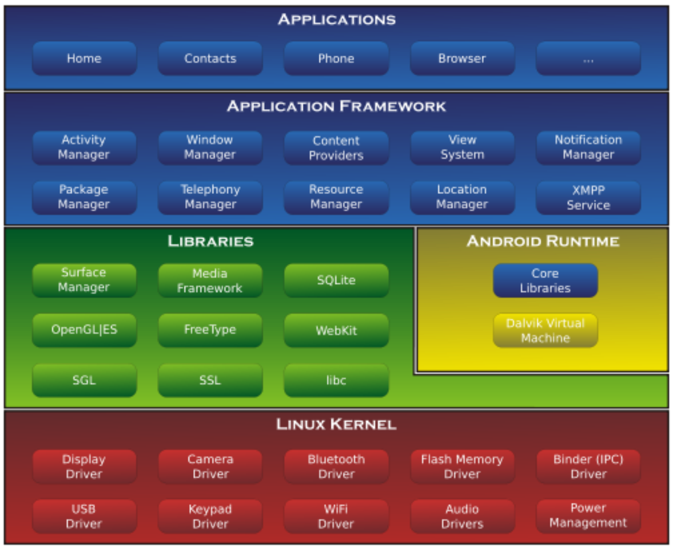


Figure 2.2: Android architecture diagram (Android, 2010)

in Dalvik. Next level, *application framework*, consists services and libraries. Here, applications and framework codes executes in Dalvik too. *Libraries* include native libraries, daemons and services which are written in C/C++. Lastly, *Linux kernel* which includes drivers for many features, networking, file system access and interprocess communication protocol. Figure 2.2 visualizes architecture overview of Android system in different layers. Green items indicate C/C++ and blue items indicate Java that run in Dalvik.

2.6.3 Releases & Ice Cream Sandwich (ICS) 4.0

Android version history started in late 2007. Early releases of Android like Android 1.0 and Android 1.1 have been developed between November 2008 - February 2009. Every update mainly fix bugs and add new features. Android has 10 different versions by now and they are named as *Cupcake*, *Donut*, *Eclair*, *Froyo*, *Gingerbread*, *Honeycomb*, *Ice Cream Sandwich* and *Jelly Bean*. *Honeycomb* was released as the first tablet-only update.

Ice Cream Sandwich, Android's previous release, became available in October 2011. With this release, also known as API 15, Android introduced WiFi Direct support along with other new features. Although there were various issues, WiFi Direct feature offered the possibility of developing one of the first applications in this field. Main flaw was in connection establishing when user runs it repetitively. It was causing an error and eventually switching the device off.

2.6.4 Jelly Bean (JB) 4.1

On June 2012, Google announced its latest Android OS called Jelly Bean. Jelly Bean was an update with improved user interface and performance. In addition to new features, Google has finally resolved bugs with its WiFi Direct feature thanks to new API 16. It is clear that the previous issues have been fixed with the latest update but essentially there are still various flaws which need to be reviewed, both in terms of functionality and performance.

2.6.5 Eclipse and Android SDK

Eclipse is an integrated development environment (IDE) that is used to develop Java applications. It is written in Java and it offers various plug-ins in order to support other programming languages such as C, C++, Perl, PHP, R, Ruby.

Android provides a custom plug-in for Android development called Android Development Tool (ADT). It is designed to built Android applications. It lets the developer to establish new Android projects, build and debug applications, and export APKs.

2.7 Overall Structure of the Solution

Project consists of 6 packages and 11 classes. Classes are grouped as packages in terms of their relevance between each other. Core classes such as *DeviceFragment.java* and *ListedFragment.java* are gathered under *com.adhoc.fragment* package while main activity and broadcast classes (*AdhocNetworkActivity.java*, *ReceiveBroadcast.java*) are grouped together as of main package, *com.adhoc.* Additionally, *com.adhoc.controller* package is responsible for controlling message passing among classes. There are two more packages which are relatively less responsive *com.adhoc.listener* and *com.adhoc.values* packages include interfaces in order to simplify the ease of access to connection establishment methods and static values of the project respectively. Finally, *com.adhoc.service* package is formed for handling file and audio transfer methods. Each class in this package contains various sub classes responsible for socket connection, audio recording, track playing, and file storing. Cohesion level of packages differs with package type. For instance, *com.adhoc*, *com.adhoc.fragment* and *com.adhoc.service* packages have higher cohesion values while *com.adhoc.listener* and *com.adhoc.values* have much less.

3 Design & Implementation

This chapter describes features, fragments, classes, architecture and the application itself by providing necessary information of major components. First, an overall information is given along with project's components and classes. Subsequently, the architecture details of the application is discussed. Section 3.3 describes classes and methods under five sections.

3.1 Overview

Application starts with instantiating broadcast receivers. These receivers briefly controls changes in P2P peers, P2P connection, P2P state and host device's P2P status. This is needed to keep control of session during or before the connection. Detailed description regarding broadcast receivers is provided in this section. Next, UI and user interaction handling sets up all necessary selections.

Following the process in activity class and broadcast receiver intent, core fragments takes place where they manage all handling and methods both in listing peers and establishing connection. Core operations of the entire system is carried out with these two classes, namely as *DeviceFragment.java* and *ListedFragment.java*

After all negotiations, peer discoveries and connection set ups, service classes sort out managing file and audio data transfer features. These services are called when connection is established between devices, and stopped after the disconnect operation. Therefore, they essentially run in the background and simply wait for available connection.

Peer discovery and data transfer is maintained as it is visualised in Figure 3.1. Both peer is able to run "Discover Peer" method and one who invoke the method is assigned as client while other is server. Once client and server are assigned, client request connection through "Request Connection" method. Method sends request to other peer - server, and waits for a response. If the result is "True" peer automatically connects to server and stands by for either transferring audio or sending image. Screenshots regarding the application and overall process can be found in Appendix A.

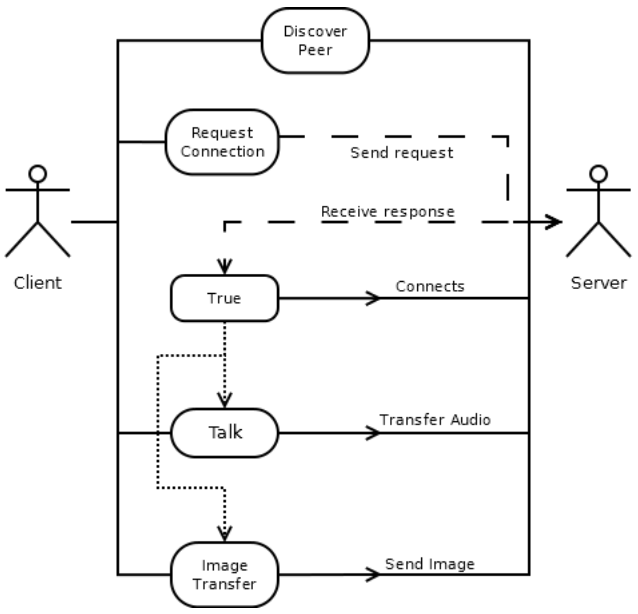


Figure 3.1: Use Case Diagram Among Peers

3.2 Components

In order to provide a detailed view concerning system mechanism, project can be grouped in three segments. These are **BroadcastReceiver**, **Fragments** and **AsyncTask & Services**.

3.2.1 Broadcast Receiver

Broadcast receiver allows user to receive intents broadcast by the Android system, so that the application can respond to events that you the user interested in (Android, 2012). An intent is an abstract description of an operation to be performed. It provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed (Android, 2012). By using broadcast receiver, user can perform operations when it is received by its `onReceive()` method.

In this project, it is used to receive intents fired from main activity class, *Adhoc-NetworkActivity.java*. When the activity starts off, receiver is registered to be run in main activity thread by using `registerReceiver(rBroadcast, intentFilter);` method. *rBroadcast* will than be called with broadcast intent that matches *intent-Filter* in the main application thread (Android, 2012). Here, intent consists various actions that are previously assigned and are ready to broadcast when events happen such as peer discovery or state changes of a device. List of actions and all the process is covered later in this chapter.

3.2.2 Fragments

Fragments handle core operations in this project such as listing peers, establishing connection, opening sockets, copying file, playing audio, showing details of devices and/or peers. They are briefly responsible for managing all session and configuring operations. Fragments also include asynchronous methods with respect to file and audio transfer as well as socket connection.

A Fragment is a piece of an application's user interface or behaviour that can be placed in an Activity and it represents a particular operation or interface that is running within a larger Activity. A Fragment is closely tied to the Activity it is in, and can not be used apart from one. Though Fragment defines its own life cycle, that life cycle is dependent on its activity: if the activity is stopped, no fragments inside of it can be started; when the activity is destroyed, all fragments will be destroyed (Android, 2012).

Certainly, views can also be used in this regard. However, it is considerably reasonable here to use fragments for some reasons. First of all, fragments are effective on creating applications for various devices such as tablets and phones. If a developer desires to split up views on different devices with different orientations and show them in two activities; or show all the content as one on other devices, using fragment gives high flexibility. In other words, fragments can act as a small activity where it can essentially have multiple of them on one screen. Moreover, these multiple activities can cooperate and co-work in terms of communicating with each other while they are visible. Thanks to its *back stack management*, pressing the back button removes dynamically added fragments before the activity itself is eventually finished. Lastly, it can have all sorts of services, operations such as AsyncTask, listeners, file and database access.

Contribution of fragments to this project is explained more detailed under Section 3.3.

3.2.3 AsyncTask & Services

Connection and data transfer operations are made by *DeviceFragment.java* class through methods,

```
public static class FileServerAsyncTask extends AsyncTask<> {...}
public static class TalkServerAsyncTask extends AsyncTask<> {...}
```

`AsyncTask` process provides a simple way to maintain a background process without working on other details such as threads and message loops. Its callback methods help user with scheduling tasks and updating UI. When a task executed, it goes through 4 steps. However, in order to use `AsyncTask` class, at least one method, `doInBackground()`, must be overridden. Methods are,

- `doInBackground()` - This part runs in a seperate thread from the UI and consists all the code which the application performs. It is called after `onPreExecute()` and before `onPostExecute()`.
- `onPreExecute()` - Called before thread starts running. It is used to setup a task that is needed to be done in advance.
- `onProgressUpdate()` - Lets `doInBackground()` method pass data to UI thread.
- `onPostExecute()` - Runs after the background thread. It takes results from `doInBackground()` method.

In this project, `AsyncTask`s are invoked on server side - `public void onConnecti onInfoAvailable() {...}` method in *DeviceFragment.java* class - once a group owner is assigned. All tasks then go into a background process where they wait for socket connection coming from services.

Services are built for opening, binding a client socket and establishing connection between server side. When a connection is established, file and audio data is stored and copied to a stream where the actual data is transferred. Due to its native characteristic, it runs in the background and waits for a possible transfer operation. They are called right after the related button is pressed. Services retrieve values from static class called *FinalValues.java* through intent's extended data. They mainly include `FinalValues.EXTRAS_GROUP_OWNER_ADDRESS` and `FinalValues.EXTRAS_GROUP_OWNER_PORT`. Port numbers are determined as 8989 and 8988 for audio data transfer and file transfer respectively.

Services and AsyncTasks are used in this study for particular reason. By definition, service is used if there is a need for executing any long process in the background. File sharing and audio data transfer are maintained using service in order not to block any foreground process and guarantee that the operation will not be interrupted. Services are best practice when there are services for critical operation like sharing file or transferring data. On the other hand, AsyncTasks provide a suitable way to maintain background processing without handling low-level details such threads, message loops. It's callback methods help to schedule tasks and update interface when it is required. Since file sharing and audio data transfer are done that are essentially isolated related to UI, using AsyncTask is best practice

3.3 Implementation

Project is designed in four parts where each part is responsible for different aspects. Essentially, main activity handles instantiate methods and pre-configuration process for WiFi Direct API such as peer discovery, connect/disconnect operations and status check whether WiFi Direct is off or on. It also consists standard Android life cycle methods e.g. `onPause()`, `onResume()` and `onDestroy()`. Particularly, `onDestroy()` and `onKeyDown()` functions manage post-configuration process in order to set the device for regular use. Killing services and setting speaker off again can be named as these processes.

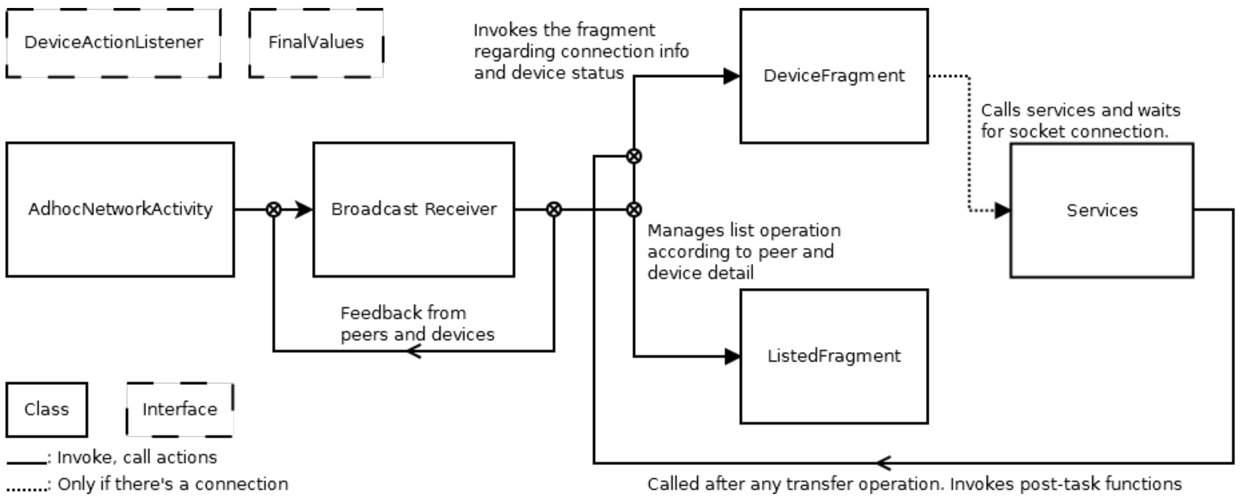

DeviceActionListener
FinalValues
Invokes the fragment regarding connection info and device status
DeviceFragment
Calls services and waits for socket connection.
AdhocNetworkActivity
Broadcast Receiver
Services
Manages list operation according to peer and device detail
Feedback from peers and devices
ListedFragment
Class
Interface
: Invoke, call actions
: Only if there's a connection
Called after any transfer operation. Invokes post-task functions

Figure 3.2: Basic system overview

As it is shown in Figure 3.2, both fragment classes are mostly invoked from *BroadcastReceiver* class in order to manage list process with respect to peer and device details, or retrieve connection info along with device status. In this regard, classes implement listeners and thus are associated with some additional methods. It can be understood more clearly by checking their listeners below.

```
DeviceFragment extends Fragment implements ConnectionInfoListener {}
ListedFragment extends ListFragment implements PeerListListener {}
```

Here, `WifiP2pManager.ConnectionInfoListener` is associated with `requestConnectInfo()` method, and `WifiP2pManager.PeerListListener` is associated with `requestPeers()` method.

3.3.1 Main Activity

AdhocMainActivity consists of six getter/setter methods, seven overridden methods coming from *android.app.Activity* API and five explicitly created methods. Getter/setter methods, namely as,

- `isPeers()`, `setPeers(boolean isPeers)` are responsible for setting and retrieving peer status
- `isDisconnected()`, `setDisconnected(boolean isDisconnected)` are responsible for checking connection status, and
- `isWifiP2pEnabled()`, `setWifiP2pEnabled(boolean isWifiP2pEnabled` are determined for setting WiFi Direct on, or returning its status.

On the other hand, callback methods are declared in order to perform operations between activity states. activity lifecycle, in other words Activity states, is an important aspect of Android OS where it ensures control on Activity process. In Figure 3.3, activity diagram is presented to briefly grasp the concept. *AdhocMainActivity* class includes only four of these callbacks, namely as `onCreate()`, `onPause()`, `onResume()` and `onDestroy()`.

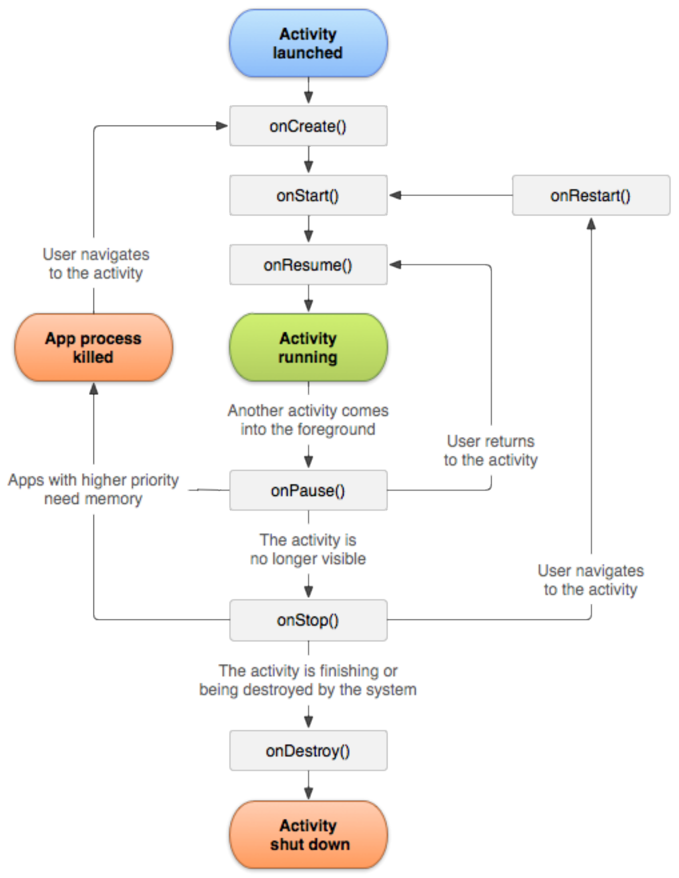

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 3.3: Activity lifecycle diagram

In this project `onCreate()` method holds one particular inside method, `initializeWiFiDirect()` for initializing WiFi Direct.

```
private void initializeWiFiDirect() {
        manager = (WifiP2pManager)
                        getSystemService(Context.WIFI_P2P_SERVICE);
        channel = manager.initialize(this, getMainLooper(),
                new ChannelListener() {
                        public void onChannelDisconnected() {
                                initializeWiFiDirect();
                                }
                        });
        }
```

Here, `manager.initialize(...);` method registers the application with Wi-Fi framework. It is the first function that is called before P2P operations are performed (Android, 2012).

Following `initializeWiFiDirect()`, intent is created and added where broadcast receiver checks for.

```
intentFilter = new IntentFilter();
intentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION)
intentFilter
        .addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION)
intentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION)
intentFilter
        .addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION)
```

These actions are included in the filter to be matched agains. As a description, these are,

- `WIFI_P2P_PEERS_CHANGED_ACTION` - Broadcast when the state of the device's WiFi connection changes
- `WIFI_P2P_CONNECTION_CHANGED_ACTION` - Broadcast when `discoverPeers()` is called. It is also required to call `requestPeers()` in order to get the updated list of peers
- `WIFI_P2P_STATE_CHANGED_ACTION` - Broadcast when WiFi Direct is enabled or disabled
- `WIFI_P2P_THIS_DEVICE_CHANGED_ACTION` - Broadcast when device's details have changed

`onPause()` and `onResume()` methods are simply for registering and unregistering broadcast receiver when user returns to the activity, or another activity comes into foreground. These methods are, `registerReceiver(rBroadcast, intentFilter)` for `onResume()` and `unregisterReceiver(rBroadcast)` for `onPause()`.

`onDestroy()` performs final operations before the activity is destroyed. In this regard, method includes functions such as cancelling P2P connection, removing P2P group, stopping both file and audio transfer services, and setting the speaker off.

```
protected void onDestroy() {
        manager.cancelConnect(channel, null);
        Log.d(AdhocNetworkActivity.TAG, "Connection cancelled");
        manager.removeGroup(channel, null);
        Log.d(AdhocNetworkActivity.TAG, "Groups removed");

        if (DeviceFragment.talkServiceIntent != null) {
                Log.d(AdhocNetworkActivity.TAG, "Talk Service killed");
                stopService(DeviceFragment.talkServiceIntent);
        }

        if (DeviceFragment.fileServiceIntent != null) {
                Log.d(AdhocNetworkActivity.TAG, "File Service killed");
                stopService(DeviceFragment.fileServiceIntent);
        }

        AudioManager aM = (AudioManager) getApplicationContext()
                .getSystemService(Context.AUDIO_SERVICE);
        aM.setSpeakerphoneOn(false);
        finish();
        super.onDestroy();
}
```

Apart from activity callbacks, *AdhocMainActivity.java* consists three extra methods, two for managing and configuring menu items and buttons, and one for handling operations when a key was pressed down. These are called, `onCreateOptionsMenu()`, `onOptionsItemSelected()` and `onKeyDown()` respectively. Here, `onKeyDown()` is doing mostly the same job as `onDestroy()` callback. It is basically an *exit button*. However, the difference is that it is implemented to prevent undesirable results which may potentially affect device's regulations. It is invoked only when user presses back button. Then, a simple dialog interface comes forward, asking user if he/she wants to quit application. If the result is positive, same process inside `onDestroy()` callback runs.

As for the last piece of overridden activity methods, `onOptionsItemSelected()` is managing the layout and button configuration. Main activity screen offers two buttons at first. These stand for enabling/disabling WiFi Direct - *with ICS (API14), WiFi Direct is enabled automatically. Thus, extra button is not needed in this release* - and discovering peers.

```
public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {

        // Enabling Wi-Fi Direct on a device.
        case R.id.atn_direct_enable:
        if (manager != null && channel != null) {
                startActivity(new Intent(Settings
                                .ACTION_WIRELESS_SETTINGS));
        } else {
                Log.e(TAG, "Channel or Manager is null");
        }
        ......
        // Discovering Wi-Fi Direct peers.
        case R.id.atn_direct_discover:
        ......
        final ListedFragment fragment = (ListedFragment)
                getFragmentManager()
                        .findFragmentById(R.id.frag_list);
        fragment.onInitiateDiscovery();

        // Discover peers.
        manager.discoverPeers(channel, new WifiP2pManager.
                                ActionListener() {

                @Override
                public void onSuccess() {
                        ......
                        setPeers(true);
                }
                @Override
                public void onFailure(int reasonCode) {
                        ......
                }
        });
        return true;
        default:
        return super.onOptionsItemSelected(item);
        }
}
```

Finally, there are four more methods in this class which are in charge of operating connection and organizing device/peers details. Major ones are `connect()` and `disconnect()`. As their names suggest, they establish and cancel connection among peers. `connect()` function takes `WifiP2pConfig config` as a parameter where `WifiP2pConfig` represents a WiFi P2P configuration for setting up a connection.

```
public void connect(WifiP2pConfig config) {
        manager.connect(channel, config, new ActionListener() {

        @Override
        public void onSuccess() {
                Log.d(AdhocNetworkActivity.TAG, "Connected");
                c.groupFormed();
        }

        @Override
        public void onFailure(int reason) {
                Log.e(AdhocNetworkActivity.TAG,
                                "Connection failed.
                                Reason code: " + reason);
                Toast.makeText(AdhocNetworkActivity.this,
                                "Connection failed.
                                Reason code: " + reason,
                                Toast.LENGTH_SHORT).show();
                c.getControlPacket().setRequestType(
                                ControlRequestType.GROUP_FAIL);
        }
        });
}
```

`manager.connect(channel, config, new ActionListener() {...}` function starts a P2P connection to a device with the specified configuration. It returns after sending a connection request to the framework and the application is notified of a success or failure to initiate connect through listener callbacks `onSuccess()` or `onFailure(int)`. The point here is, the registration for `WIFI_P2P_CONNECTION_CHANGED_ACTION` intent determines when the framework notifies of a change in connectivity. If the device is not part of a P2P group, a connect request initiates a group negotiation with the peer. Otherwise, if the device is part of an existing P2P group or has created a P2P group with `createGroup(WifiP2pManager.Channel,` WifiP2pManager.ActionListener), an invitation to join the group is sent to the peer device (Android, 2012).

On the other hand, disconnect() function removes P2P group and calls resetData() method. By calling resetData(), both fragment classes are invoked to clear all fields and reset details.

```
public void resetData() {
        ListedFragment fragmentList = (ListedFragment)
                getFragmentManager()
                        .findFragmentById(R.id.frag_list);
        DeviceFragment fragmentDetails = (DeviceFragment)
                getFragmentManager()
                        .findFragmentById(R.id.frag_detail);
        if (fragmentList != null) {
                fragmentList.clearPeers();
        }
        if (fragmentDetails != null) {
                fragmentDetails.resetDetails();
        }
}
```

Moreover, after resetting all data, manager.removeGroup(channel, new ActionListener() {...} removes the current P2P group. Same as connect() function, application is notified here again to initiate the process through listener callbacks onSuccess() or onFailure(int).

```
public void disconnect() {
        resetData();

        manager.removeGroup(channel, new ActionListener() {

        @Override
        public void onSuccess() {
                Log.d(AdhocNetworkActivity.TAG, "Disconnected");
                c.getControlPacket().setRequestType(
                                ControlRequestType.GROUP_RESOLVED);
                setDisconnected(true);
                setPeers(false);
        }

        @Override
        public void onFailure(int reason) {
                Log.e(AdhocNetworkActivity.TAG,
                                "Disconnection failed.
                                Reason code: " + reason);
                Toast.makeText(AdhocNetworkActivity.this,
                                "Disconnection failed.
                                Reason code: " + reason,
                                Toast.LENGTH_SHORT).show();
                c.getControlPacket().setRequestType(
                                ControlRequestType.GROUP_FAIL);
        }
        });
}
```

`showDetails()` method, as of its same characteristic like `resetData()` method, calls for fragment class *DeviceFragment* in this case, to show device details on text views once the peer is clicked. It is coded as,

```
public void showDetails(WifiP2pDevice device, boolean check) {
        DeviceFragment fragment = (DeviceFragment)
                getFragmentManager()
                        .findFragmentById(R.id.frag_detail);
        fragment.showDetails(device, check);
}
```

3.3.2 Broadcast Receiver

ReceiverBroadcast.java class checks for intents that the user interested in `onReceive()` method. Intents that are defined in this class are explained briefly in the previous section. This class is the implementation of typical broadcast receiver where it carries out necessary actions when an intent is received.

```
public class ReceiveBroadcast extends BroadcastReceiver {

private WifiP2pManager wifiP2pManager;
private Channel channel;
private AdhocNetworkActivity activity;

public ReceiveBroadcast(WifiP2pManager wifiP2pManager,
                Channel channel,
                AdhocNetworkActivity activity) {
                super();
        this.wifiP2pManager = wifiP2pManager;
        ths.channel = channel;
                this.activity = activity;
}

@Override
public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();

        if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION
                .equals(action)) {...}
        else if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION
                .equals(action)) {...}
        else if (WifiP2pManager
                .WIFI_P2P_CONNECTION_CHANGED_ACTION
                        .equals(action)) {...}
        else if (WifiP2pManager
                .WIFI_P2P_THIS_DEVICE_CHANGED_ACTION
                        .equals(action)) {...}
        }
}
```

Each if condition checks for intent actions and runs related code with respect to desired action. First condition is responsible for calling `WifiP2pManager.requestPeers()` method in order to retrieve the list of current peers. *ListedFragment.java* class is invoked in this regards as it is associated with interface `WifiP2pManager.PeerListListener`.

```
ListedFragment lFrag = (ListedFragment) activity
        .getFragmentManager().findFragmentById(R.id.frag_list);
wifiP2pManager.requestPeers(channel, lFrag);
```

Next is checking to see if WiFi is enabled. If that condition is true, main activity is notified. When it is disabled, all data is set to null as well as the constructors. Additionally, device details are reset through fragment class.

```
int state = intent.getIntExtra(WifiP2pManager.EXTRA_WIFI_STATE, -1);

if (state == WifiP2pManager.WIFI_P2P_STATE_ENABLED) {
        activity.setWifiP2pEnabled(true);
} else {
        activity.setWifiP2pEnabled(false);
        activity.setPeers(false);
        activity.resetData();
        ListedFragment fragOff = (ListedFragment) activity
                        .getFragmentManager()
                        .findFragmentById(R.id.frag_list);
        fragOff.resetThisDevice();
}
```

Third condition checks if the state of Wi-Fi P2P connectivity has changed. If so, network info is extracted using `NetworkInfo` class. Once network connectivity exists and is possible to establish connections, *DeviceFragment.java* class is invoked with respect to its associated method `requestConnectionInfo()` through `WifiP2pManager.ConnectionInfoListener` interface. In other case, button configurations are managed where network connectivity does not exist, such as disabling disconnect button and enabling connect button.

```
String extraKey = WifiP2pManager.EXTRA_NETWORK_INFO;
NetworkInfo networkInfo = (NetworkInfo) intent
                .getParcelableExtra(extraKey);

if (networkInfo.isConnected()) {
        Log.d(AdhocNetworkActivity.TAG, "Wifi p2p state has changed!");
        DeviceFragment frag = (DeviceFragment) activity
                        .getFragmentManager()
                .findFragmentById(R.id.frag_detail);
        wifiP2pManager.requestConnectionInfo(channel, frag);
} else {
        activity.resetData();

        DeviceFragment.view.findViewById(R.id.btn_disconnect)
                .setEnabled(false);
        DeviceFragment.view.findViewById(R.id.btn_connect)
                .setVisibility(View.VISIBLE);
        DeviceFragment.view.findViewById(R.id.btn_connect).setEnabled(
                true);

}
```

Finally, intent action is issued indicating if device details have changed. Updating device details, cancelling current tasks or clearing all fields are managed here according to device's WiFi state.

```
if (activity.isWifiP2pEnabled) {

        if (!activity.isDisconnected) {
                ListedFragment fragment = (ListedFragment) activity
                        .getFragmentManager().
                                findFragmentById(R.id.frag_list);
                fragment.updateThisDevice((WifiP2pDevice) intent
                        .getParcelableExtra(WifiP2pManager.
                                EXTRA_WIFI_P2P_DEVICE));
        }

        if (DeviceFragment.info != null) {
                if (DeviceFragment.info.groupFormed
                        && DeviceFragment.info.isGroupOwner) {
                        ......
                        DeviceFragment.task.cancel(true);
                        DeviceFragment.task2.cancel(true);
                }
        }

} else {
        activity.resetData();
        ......
}
```

3.3.3 Controller

Controller package is designed for message passing between classes in the project. In many occasions, classes need to communicate in order to inform related methods. However, this might not be possible as a reason of using different instances of the same class. Therefore, a controller package and various message passing method is usually needed. In this project, control system is mostly used for setting and getting request type. But essentially the package and classes consist more functions such as getting source/destination IP and getting source/destination port number. These segments are not used in this project. They are implemented for future use and for future developers.

- **Controller**: This class consists of two constructors, five methods for retrieving new *ControlPacket.java* instance and source/destination IP - port number. Last two methods state for setting requesting type in respect of group configuration.

```
public Controller() {
        controller = new ControlPacket();
}

public Controller(String sourceIP, int sourcePort,
                String destinationIP,
                int destinationPort) {
        try {
                controller = new ControlPacket();
                controller.setSourceIP(sourceIP);
                controller.setSourcePort(sourcePort);
                controller.setDestinationIP(destinationIP);
                controller.setDestinationPort(destinationPort);
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

  Methods for getting source/destination IP and port numbers basically get values from *ControlPacket.java* class through `controller` and one extra method for reaching *ControlPacket.java* class returns this `controller` instance.

```
public ControlPacket getControlPacket() {
        return controller;
}

public String getSourceIP() {
        return controller.getSourceIP();
}

public int getSourcePort() {
        return controller.getSourcePort();
}

public String getDestinationIP() {
        return controller.getDestinationIP();
}

public int getDestinationPort() {
        return controller.getDestinationPort();
}
```

Finally, methods for setting request type is handled in this class again through *ControlPacket.java* class over `controller` whether group is formed or resolved.

- **ControlPacket**: *Control Packet* is a class where all necessary information is kept about either session, connection or a device. This is done by various methods and all data is supposed to be stored in this class. Instance of this class is created in *Controller.java* and data access can be done easily from there. As their name suggest, methods manage storing data and certainly returning its values. Although only two methods - `getRequestType()` and `setRequestType()` - are usually used in this project, all methods are listed below.

```
public int getRequestType() {...}
public void setRequestType(int requestType) {...}
public String getSourceIP() {...}
public void setSourceIP(String sourceIP) {...}
public int getSourcePort() {...}
public void setSourcePort(int sourcePort) {...}
public String getDestinationIP() {...}
public void setDestinationIP(String destinationIP) {...}
public int getDestinationPort() {...}
public void setDestinationPort(int destinationPort) {...}
```

- **ControlRequestType**: This interface holds all message types associated as static integers. With the help of this interface, more content can be added along with its integer value if desired.

```
public interface ControlRequestType {
        public static final int WIFI_DIRECT_ENABLED = 1;
        public static final int WIFI_DIRECT_DISABLED = 2;
        public static final int PEERS_DISCOVERED = 3;
        public static final int PEERS_DISCOVERED_FAIL = 4;
        public static final int SOCKET_CONNECTION_ESTABLISHED = 5;
        public static final int SOCKET_CONNECTION_FAIL = 6;
        public static final int SOCKET_CONNECTION_ABOLISHED = 7;
        public static final int FILE_TRANSFER_SERVICE_STARTED = 8;
        public static final int FILE_TRANSFER_SERVICE_STOPPED = 9;
        public static final int AUDIO_TRANSFER_SERVICE_STARTED = 10;
        public static final int AUDIO_TRANSFER_SERVICE_STOPPED = 11;
        public static final int GROUP_FORMED = 12;
        public static final int GROUP_RESOLVED = 13;
        public static final int GROUP_FAIL = 14;
}
```

3.3.4 Discovery and Connection

Peer discovery and connection run through fragments and asynchronous tasks. When the relevant intent is invoked, either *DeviceFragment.java* or *ListedFragment.java* class is called and involved methods are fetched in order to discover peers and eventually establish connection.

- **DeviceFragment**: As it was discussed before in Section 3.3.2, this class is invoked when there is a connection to a WiFi Direct peer is detected. First method to run in this regard is `onCreateView()` method. It includes four listener, namely as `onClickListener` where connect, disconnect, file share and talk buttons are associated with respectively. Each callback is invoked when a related view is clicked. On the other hand, different callback is invoked right after the fragment starts running, which is called `onConnectionInfoAvai lable()`. This callback is overridden from `WifiP2pManager.ConnectionInfoListe ner` and is associated with `requestConnectionInfo()` action. Once a connection to a peer is detected in broadcast receiver, the code below runs and this automatically starts the action in *DeviceFragment.java* class.

  ```
  DeviceFragment frag = (DeviceFragment) activity
          .getFragmentManager().findFragmentById(R.id.frag_detail);
  wifiP2pManager.requestConnectionInfo(channel, frag);
  ```

  The main action in `onConnectionInfoAvailable(WifiP2pInfo info)` is simply assigning the group owner as the server through `WifiP2pInfo.info`. With the help two boolean methods, `info.groupFormed` and `info.isGroupOwner`, device status is checked and asynchronous tasks are started unless the result is false. If the device is client, get file and talk button are enabled and no task run.

  In order to grasp the concept detailed, it is necessary to clarify the main issue inside method. Below is the first piece of code which defines the group owner IP address and sets the text to this IP value in related view. This is handled only for information purposes.

  ```
  public void onConnectionInfoAvailable(WifiP2pInfo info) {
          ...

          DeviceFragment.info = info;
          this.getView().setVisibility(View.VISIBLE);

          if (!info.groupFormed) {
                  gOwnerIp = "Group Owner IP - null";
                  gOwner = "null";
          } else {
                  gOwnerIp = "Group Owner IP - "
                          + info.groupOwnerAddress.getHostAddress();
                  gOwner = (info.isGroupOwner == true) ? getResources()
                  .getString(
                          R.string.yes) : getResources().
                                  getString(R.string.no);
          }

          TextView textView = (TextView) view
                  .findViewById(R.id.group_owner);
          textView.setText(getResources()
                  .getString(R.string.group_owner_text)
                          + (gOwner));
  ```

```
        textView = (TextView) view.findViewById(R.id.device_info);
        textView.setText(gOwnerIp);
```

Second part of the method is the section where asynchronous tasks are set to start according to group owner information. No task execution is made if the device is not the group owner. Additionally, `if (firstCheck) {...} else {...}` condition indicates whether the server actions have started for the first time. If not, a double check is performed by the methods below in order not to start new tasks before the previous ones are completed.

```
if (task.getStatus() == Status.FINISHED) {...}
if (task2.getStatus() == Status.FINISHED) {...}
```

Finally, here is the second part where tasks are executed.

```
if (info.groupFormed && info.isGroupOwner) {

        view.findViewById(R.id.btn_connect).setEnabled(false);
        view.findViewById(R.id.btn_disconnect).setEnabled(true);

        localIp = getDottedDecimalIP(getLocalIPAddress());

        if (firstCheck) {

                // Start File server as a thread.
                ......
                task = new FileServerAsyncTask(getActivity(),
                                view.findViewById(R.id.status_text))
                                .execute();

                // Start Talk server as a thread.
                ......
                task2 = new TalkServerAsyncTask(getActivity(),
                                view.findViewById(R.id.status_text))
                                .executeOnExecutor(AsyncTask
                                        .THREAD_POOL_EXECUTOR);

                firstCheck = false;
        }

        else {
                if (task.getStatus() == Status.FINISHED) {

                // Start File server as a thread.
                ......
                task = new FileServerAsyncTask(getActivity(),
                                view.findViewById(R.id.status_text))
                                .execute();
                }
                if (task2.getStatus() == Status.FINISHED) {

                // Start Talk server as a thread.
                ......
                task2 = new TalkServerAsyncTask(getActivity(),
                                view.findViewById(R.id.status_text))
                                .executeOnExecutor(AsyncTask
                                .THREAD_POOL_EXECUTOR);
                }
        }
}

else if (info.groupFormed) {
        localIp = getDottedDecimalIP(getLocalIPAddress());
        view.findViewById(R.id.btn_start_client)
                        .setVisibility(View.VISIBLE);
        view.findViewById(R.id.btn_talk).setVisibility(View.VISIBLE);
        ((TextView) view.findViewById(R.id.status_text))
                        .setText(getResources()
                        .getString(R.string.client_text));
        // Hide connect button.
        view.findViewById(R.id.btn_connect).setVisibility(View.GONE);
}
}
```

As well as `onConnectionInfoAvailable()` callback another method takes essential part in *DeviceFragment.java* class. `onCreateView()` is called to have the fragment instantiate its user interface view. In this regard, method consists four listeners which are invoked when its related button was pressed. These listeners indicate buttons namely as *connect*, *disconnect*, *file share* and *talk*.

Starting from *connect* button, each run different code section. When *connect* button is pressed Wifi P2P is configured for setting up a connection. Than, device MAC address is assigned to `config` parameter along with extra WiFi setup information. Following initial WiFi setup, group owner selection process runs. User who presses connect button will have the least inclination and thus is assigned as a group owner. By definition, 0 indicates the least inclination to be a group owner and 15 indicates the highest inclination to be a group owner (Android, 2012).

```
WifiP2pConfig config = new WifiP2pConfig();
config.deviceAddress = device.deviceAddress;
config.wps.setup = WpsInfo.PBC;
config.groupOwnerIntent = 0;
```

`connect(config)` method is called next from *DeviceActionListener.java* interface which is essentially implemented in main activity. Details have been discussed in the Section 3.3.1 where main activity is described.

```
((DeviceActionListener) getActivity()).connect(config);
```

When *disconnect* button is pressed, `disconnect()` method is called from interface similar as *connect* button. However, different from *connect* button, here, broadcast receiver is re-registered in order to refresh P2P connection. In addition to this, button visibilities and other layout configurations are also handled in here.

```
((DeviceActionListener) getActivity()).disconnect();

view.findViewById(R.id.btn_disconnect)
                .setEnabled(false);
view.findViewById(R.id.btn_connect).setVisibility(
                View.VISIBLE);
view.findViewById(R.id.btn_connect).setEnabled(true);

Log.d(AdhocNetworkActivity.TAG,
                "Receivers reregistered");
getActivity().unregisterReceiver(
                AdhocNetworkActivity.rBroadcast);
getActivity().registerReceiver(
                AdhocNetworkActivity.rBroadcast,
                AdhocNetworkActivity.intentFilter);
```

file share button is configured to start activity through intent. When it is pressed, `startActivityForResult(intent, FinalValues.CHOOSE_FILE_RESUL T_CODE)` is called on the fragment's containing activity. Following this operation, `onActivity Result()` method receives the result from a previous call and creates intent for file sharing.

```
public void onActivityResult(int requestCode, int resultCode,
        Intent data) {
        Uri uri = data.getData();

        fileServiceIntent = new Intent(getActivity(),
                FileTransferService.class);
        fileServiceIntent.setAction(FinalValues.ACTION_SEND_FILE);
        fileServiceIntent
                .putExtra(FinalValues
                .EXTRAS_FILE_PATH, uri.toString());
        fileServiceIntent.putExtra(FinalValues.
                EXTRAS_GROUP_OWNER_ADDRESS,
                info.groupOwnerAddress.getHostAddress());
        fileServiceIntent.putExtra(FinalValues.
                EXTRAS_GROUP_OWNER_PORT, 8988);

        // Set control message.
        c.getControlPacket().setRequestType(
                ControlRequestType.
                FILE_TRANSFER_SERVICE_STARTED);
        // Start service
        getActivity().startService(fileServiceIntent);
}
```

Intent is attached to a service class called *FileTransferService* and it is configured to start a service intent along with the addition of extended data which are listed below. Lastly, service starts after control message is set for related request type.

- `FinalValues.EXTRAS_FILE_PATH, uri.toString()` - URI of the data which this intent is targeting.
- `FinalValues.EXTRAS_GROUP_OWNER_ADDRESS, info.groupOwnerAddress .getHostAddress()` - Group owner's IP address.
- `FinalValues.EXTRAS_GROUP_OWNER_PORT, 8988` - Group owner's port number. 8988 is assigned for file sharing.

Last callback is set for *talk* button. Although the concept of this listener is similar - almost same in many cases - to *file share* button, it implements a dialog interface where the user can stop recording arbitrarily. - *Talk process works as transferring audio data that is stored in a file, not like a live streaming. This is covered under Section 4.2 in detail.*

First section specifies additional information for alert dialog. Following this, intent is created for audio transfer and extended data is added together with certain information regarding IP address and port number. Unlike file sharing, port number is assigned to **8989** to avoid duplication. Lastly, service start is handled before dialog interface and its sub method which includes code to run when an item on the dialog is clicked. Below is the code which notifies *TalkService* class about recording state; **false** in this case.

```
alertDialog.setButton("Stop",
        new DialogInterface.OnClickListener() {

        @Override
        public void onClick(DialogInterface dialog,
                int which) {
                TalkService.setRecording(false);
                }
        });

// Start dialog.
alertDialog.show();
```

- **AsyncTask**: One of the main aspects of this application is certainly task process. Once a connection is established, data can be transferred between devices with sockets. However, creating a socket should be done in a background thread because `ServerSocket` waits for a connection from a client on a specific port and blocks until it happens. In this regards, *DeviceFragment.java* class includes two methods called `FileServerAsync Task()` and `TalkServerAsyncTask()`. Owing to early provided description, this part discusses and explains the methods here.

Starting with `doInBackground(Void... params)` method, server socket is created. This socket waits for client connections. This blocks until a connection is accepted from a client. Here, client is invoked through service and expected acceptance will be received from this service, also through `Socket`.

```
serverSocket = new ServerSocket(8988);
final Socket client = serverSocket.accept();
```

When a connection happens, server can receive data from client. Subsequently, a file description is created for incoming input stream.

```
final File f = new File(
        Environment.getExternalStorageDirectory() + "/"
                + context.getPackageName() + "/picture-"
                + System.currentTimeMillis() + ".jpg");
```

For audio data, file description is provided in a different way and is named as `audio-sample.pcm`

`InputStream` indicates reading data from the file system (FileInputStream), the network (getInputStream()/getInputStream()), or from an in-memory byte array (ByteArrayInputStream) (Android, 2012). Regarding this application, data will be read from the network, in other words `socket`. In this respect, the following code is used.

```
InputStream inputstream = client.getInputStream();
```

Here, `getInputStream()` returns an input stream to read data from this socket. Folllowing this, a method indicated below is called for copying data from stream. This method is called both in AsyncTasks and Services and can also be used for copying `to` stream.

```
copyFile(inputstream, new FileOutputStream(f));
```

And the actual code that runs inside the method is,

```
public static void copyFile(InputStream inputStream, OutputStream out)
{
        byte buf[] = new byte[1024];
        int len;
        try {
                while ((len = inputStream.read(buf)) != -1) {
                        out.write(buf, 0, len);
                }
                out.close();
                inputStream.close();
        } catch (IOException e) {
                Log.d(AdhocNetworkActivity.TAG, e.toString());
        }
}
```

Basically, `inputStream.read(buf)` function reads at most length bytes from this stream and stores them in the byte array - `buf` (Android, 2012). When it is complete, `out.write(buf, 0, len)` function takes place where it writes count bytes from the byte array buffer - `buf` - starting at position offset - `len` - to this stream (Android, 2012). It must be noted here that data is written in to `FileOutputStream(f)` where `f` is denoted as a file which is created for incoming input stream.

`AsyncTask` is concluded by closing server socket and setting related request type as an instance of controller.

onPostExecute(): This callback has different roles in each asynchronous task e.g. `FileServerAsyncTask()` and `TalkServerAsyncTask()`. In terms of file sharing, an intent is called for showing the transferred file. In other words, image is shown to the user after a successful transfer operation. This is a simple intent action and is coded as below.

```
statusText.setText("File copied - " + result);
Intent intent = new Intent();
intent.setAction(android.content.Intent.ACTION_VIEW);
intent.setDataAndType(Uri.parse("file://" + result), "image/*");
context.startActivity(intent);
```

Nevertheless, `TalkServerAsyncTask()` performs only broadcast receiver re-registration.

TalkServerAsyncTask(): This task performs almost the same code as `File ServerAsyncTask()` with one exception. As its name suggests, this task issues audio data transfer. However, task also performs `playAudio()` function as well as socket connection and reading data.

`playAudio()` method simply plays audio data from file. Method first writes data from input stream to a short array, then plays it over `AudioTrack`. Short array has a size of buffer size which is declared as a static integer value. 0 value is set to each array in order not to catch `NullPointerException`. More details is provided regarding buffer size in next Section 3.3.5.

```
short[] dump = new short[FinalValues.BUFFER_SIZE];
for (int i = 0; i < dump.length; i++)
        dump[i] = 0;
```

Following this, data is read through input stream and stored in a short array which is created before. Data reading operation is managed by `DataInputStream` class. `DataInputStream` wraps an existing InputStream and reads big-endian typed data from it (Android, 2012). After reading process, instance of `DataInputS tream` is closed.

```
InputStream isFile = new FileInputStream(f);
BufferedInputStream bisFile = new BufferedInputStream(isFile);
DataInputStream disFile = new DataInputStream(bisFile);
```

```
while (len < disFile.available()) {
        dump[len] = disFile.readShort();
        len++;
}
disFile.close();
```

Additionally, speaker setting is managed using `AudioManager` class for hearing the data from device speakers. `AudioTrack` initialization is maintained according to static values set in *FinalValues.java* class. Here, a listener is set to notify `AudioTrack` when a previously set marker is reached or for each periodic playback head position update (Android, 2012). In other words, listener sends notification in every second during playback process. Currently, this function does not provide any practical code, but it is implemented for future uses when an interaction or intervention is needed during playback. Lastly, data has been written to the audio hardware for playback and it is eventually played. - *Track does not stop playing before all data in short array has been processed. It is discussed under Section 4.2.*

```
aManager = (AudioManager) context
        .getSystemService(Context.AUDIO_SERVICE);
aManager.setMode(AudioManager.MODE_NORMAL);
aManager.setSpeakerphoneOn(true);

AudioTrack aTrack = new AudioTrack(AudioManager.STREAM_MUSIC,
        FinalValues.SAMPLE_RATE, FinalValues.CHANNEL_OUT_MODE,
        FinalValues.ENCODING_MODE, FinalValues.BUFFER_SIZE,
        AudioTrack.MODE_STREAM);

aTrack.setPositionNotificationPeriod(FinalValues.SAMPLE_RATE);
aTrack.setPlaybackPositionUpdateListener(new
        AudioTrack.OnPlaybackPositionUpdateListener() {
                int count = 1;

                @Override
                public void onPeriodicNotification(AudioTrack track)
                {
                Log.e(AdhocNetworkActivity.TAG,
                        "Period notification: "
                                + count++);
                }

                @Override
                public void onMarkerReached(AudioTrack track) {

                }
});

aTrack.setPlaybackRate(FinalValues.SAMPLE_RATE);
aTrack.play();
aTrack.write(dump, 0, dump.length);
```

Before concluding this class description, there are also few additional methods that should be covered. These can be grouped in two where first group manages views, configures layout while second group handles IP display.

`showDetails(WifiP2pDevice device, boolean c)` method shows device details on text views when a peer presses. Parameter `c` controls if it is clicked once or twice and sets the view to invisible if `c` is true. On the other hand, `resetDetails()` basically resets all data and text fields in fragments when WiFi Direct is disabled.

IP display approach consists 2 methods, `getLocalIPAddress()` and `getDotted DecimalIP(byte[] ipAddr)`. Through these methods, local IPv4 address is retrieved and returned as a byte array, eventually is changed to dotted format for providing nice format.

- **ListedFragment**: This fragment has similar characteristic as the previous one. It is also invoked from broadcast receiver, but with one difference; to get a list of current peers. Other methods in this class are `resetThisDevice()` and `updateThisDevice()`. Although these classes are also called from broadcast receiver, each has got a particular precondition.

  Before describing each method, process should be well understood. As it was mentioned before, *ListedFragment.java* includes associated methods. When available peer list has changed, fragment is invoked through broadcast receiver in order to run function called `requestPeers()`. It requests current list of peers and gets a callback as a parameter, in other words *listener*.

```
ListedFragment lFrag = (ListedFragment) activity
        .getFragmentManager().findFragmentById(R.id.frag_list);
wifiP2pManager.requestPeers(channel, lFrag);
```

  Here, `lFrag` is used as a listener and it can be seen that its type is `ListedFragmen t`. So when `requestPeers()` function is called, it invokes *ListedFragment.java* class. *ListedFragment* implements `PeerListListener` interface for callback invocation when peer list is available, and this interface overrides one method called `onPeersAvailable`. Lastly, this method runs a few lines of code which clears the device list and forms new one when the requested peer list is available.

```
public void onPeersAvailable(WifiP2pDeviceList device) {
        ......
        deviceList.clear();
        deviceList.addAll(device.getDeviceList());
        ((DeviceListAdapter) getListAdapter()).notifyDataSetChanged();
}
```

  Certainly there are other methods running for generating the list. `onCreateVie w()` and `onActivityCreated()` are the methods when fragment's activity has been created and starts off. Methods simply generate list view for peer list and provides cursor for the list view. Here, list has been set using `setListAdapter()` method along with proper parameters.

```
this.setListAdapter(new DeviceListAdapter(getActivity(),
        R.layout.row_devices, deviceList));
```

- **DeviceListAdapter**: `DeviceListAdapter` is a class that extends `ArrayAdapter` where lists or arrays of custom objects can be added by overriding various methods. For the array display, `getView(int, View, ViewGroup)` is overridden in order to fill the list view (Android, 2012).

```
public View getView(int position, View convertView, ViewGroup parent)
{
        View v = convertView;
        if (v == null) {
                LayoutInflater vi = (LayoutInflater) getActivity()
                        .getSystemService(Context
                        .LAYOUT_INFLATER_SERVICE);
                v = vi.inflate(R.layout.row_devices, null);
        }
        WifiP2pDevice device = items.get(position);

        if (device != null) {
                TextView top = (TextView) v.
                        findViewById(R.id.device_name);
                TextView bottom = (TextView) v
                .findViewById(R.id.device_details);
                if (top != null) {
                        top.setText(device.deviceName);
                }
                if (bottom != null) {
                        bottom.setText(getDeviceStatus(device.status));
                        if (getDeviceStatus(device.status) == "Failed")
                        {
                                Toast.makeText(
                                getContext(),
                                "Problem in API occured.
                                Please restart WiFi Direct",
                                        Toast.LENGTH_LONG).show();
                        }
                }
        }
        return v;
}
```

First condition checks for null instance of `View` and inflates a new view hierarchy from the specified xml resource - `R.layout.row_devices` - unless `v` is null. Subsequently, element is fetched at the specified location and set as an instance of `WifiP2pDevice`. Lastly, text view is set with respect to device name, detail and status that are received from WiFi P2P device.

Apart from that, additional methods such as `getDeviceStatus(int status)`, `clearPeers()`, `updateThisDevice()`, `resetThisDevice()` and `onInitiateDisc overy()` help configuring and managing list view according to relevant input. `getDevi ceStatus(int status)` switches between declared status according to parameter `status`. This method is called from `DeviceListAdapter` class and also from `update ThisDevice()` method in response to text view generation. Finally, `onInitiateDisc overy()` method runs `ProgressDialog` process which shows dialog interface while discovering peers.

3.3.5 Data Transfer and Services

Data transfer is established through services. These services act as client and connect to the server socket with a client socket and eventually transfer data. There are two services responsible for the operation, *FileTransferService* and *TalkService*.

- **FileTransferService**: Class extends `IntentService` which includes one method called `onHandleIntent()`. This method is invoked on the worker thread with a request to process. After the initialization process, `socket` is opened and binding process starts. `socket.bind(null)` method binds this socket to the given local host address and port specified. Here, because of local address is set to null, this socket will be bound to an available local address on any free port (Android, 2012). Upon this process, socket connects to the given remote host address and port.

```
String host = intent.getExtras().getString(
        FinalValues.EXTRAS_GROUP_OWNER_ADDRESS);
int port = intent.getExtras().getInt(
        FinalValues.EXTRAS_GROUP_OWNER_PORT);
        ......
socket.connect((new InetSocketAddress(host, port)),
        FinalValues.SOCKET_TIMEOUT);
```

  Specified host address and port are received from interface class *FinalValues.java* as static values. Timeout value is in milliseconds or 0 for an infinite timeout. In this case it is **5000**, or 5 seconds.

  After setting control message regarding establishing connection, `OutputStream` is used to write data to the network via `getOutputStream()`. This method returns an output stream to write data into socket. Following this operation, an input stream is opened on to the content associated with a content URI. Input stream is used in order to read data from the network for later use.

```
OutputStream stream = socket.getOutputStream();
ContentResolver cr = context.getContentResolver();
InputStream is = null;
try {
        is = cr.openInputStream(Uri.parse(fileUri));
} catch (FileNotFoundException e) {
        Log.d(AdhocNetworkActivity.TAG, e.toString());
}
```

  After fetching data URI, data is copied to stream using the same method on server side, `copyFile(is, stream)`. Here, data is written to output stream through socket and thus transfer is completed. Final process simply closes socket and sets control message with respect to request type.

- **TalkService**: Unlike previous service, *TalkService* includes one extra class and one method `record Audio()` and `socketConnection(context, host, port)`. `socketConnection()` runs mostly the same code in *FileTransferService* for transfer process. Only difference is its URI value and the way it is fetched. URI is created from a file using the method below. Socket binding, connection and file transfer process is handled using the same technique as in *FileTransferService*. *- Audio data is transferred as a file. Live streaming could not be used due to inabilities. For more information, see Section 4.2.*

```
Uri uri = android.net.Uri.fromFile(file);
```

  Service begins with invoking `onHandleIntent()` method on the worker thread with a request to process. Host address and port number is fetched for socket connection and a new, empty file is created on the file system according to the path information stored in the file, - `/com.adhoc/audio-sample.pcm`. Subsequently, a thread `recordAudio()` is generated and started with following code below. Process is described in detail under *recordAudio()*.

```
tRecord = new recordAudio();
tRecord.start();
```

  Certainly, thanks to thread's characteristic, recording process runs in the background while service waits for an invocation. By saying invocation, service runs `while` loop while thread is still alive. When thread reaches the end, `isDone` is set to true and `socketConnection()` method is invoked for transferring data through socket and finalizing the service. Here, another problem might occur about thread handling. Occasionally, thread does not end and therefore process does not advance to socket connection section. This leads to incomplete asynchronous task and eventually blocks following audio transfers. This issue is discussed under Section 4.2.

- **recordAudio()**: `recordAudio()` extends `Thread` and runs various code regarding audio recording. Here, audio is recorded from device microphone according to sample rate and buffer size. These values specify the recording time. Table 3.1 shows recording time for different sample rates and buffer sizes.

Frequency / Buffer Size	8k	11k	22k	44k
128 kHz	8 sec	5 sec	2,5 sec	1,25 sec
256 kHz	16 sec	10 sec	5 sec	2,5 sec
512 kHz	32 sec	25 sec	12,5 sec	5,5 sec

Table 3.1: Average Recording Time in Seconds

  Thread runs three lines of code that manages streaming process. First `FileOut putStream` is constructed as an instance of `OutputStream` for writing bytes to `file`. Then, `file` is wrapped and the output is buffered. Lastly, `DataOutputStre am` is constructed on the `BufferedOutputStream bosFile`.

```
OutputStream osFile = new FileOutputStream(file);
BufferedOutputStream bosFile = new BufferedOutputStream(osFile);
final DataOutputStream dosFile = new DataOutputStream(bosFile);
```

After setting constructor `AudioRecord` with respect to *audio source*, *sample rate*, *channel configuration*, *audio format* and *buffer size*; short array is created with the size of given buffer size. Recording starts from `AudioRecord` instance just after this process and audio data is read from the audio hardware for recording into a buffer. `FinalValues.BUFFER_SIZE` implies the total number of data to be read and `0` is the offset value from which the data is written until reaches the highest value, in other words buffer size.

```
aRecorder.startRecording();
......
aRecorder.read(data, 0, FinalValues.BUFFER_SIZE);
......
```

Notification period and listener is set in order to notify for each periodic record head position update, in other words every second. For `onPeriodNotification()` method, stop process is partly carried out to stop recording when user presses the button and invokes related method. However, an issue occurred on this process. When user desires to stop the ongoing process prematurely, system is supposed to stop and release the recorder and eventually finalize the action. From time to time, this related code does not execute and does not respond to user's interaction. When there is no response, it is observed that process does not advance to next section and therefore data transfer is not completed. To resolve this issue, a very basic and simple escape condition is implemented which simply adds extra integer value to ongoing counter and this counter value is checked in every second through `onPeriodNotification` method. Nonetheless, problem still exists and it is discussed in Section 4.2.

```
aRecorder = new AudioRecord(MediaRecorder.AudioSource.MIC,
        FinalValues.SAMPLE_RATE, FinalValues.CHANNEL_IN_MODE,
        FinalValues.ENCODING_MODE, FinalValues.BUFFER_SIZE);

final short[] data = new short[FinalValues.BUFFER_SIZE];
aRecorder.setPositionNotificationPeriod(FinalValues
                                .SAMPLE_RATE);
aRecorder.setRecordPositionUpdateListener(new
        AudioRecord.OnRecordPositionUpdateListener() {
                int count = 1, c = 0;

                @Override
                public void onPeriodicNotification(
                        AudioRecord recorder) {
                if (c == count + 2) {
                        recorder.stop();
                        recorder.release();
                        ......
                        requestStop();
                        ......
                        if (recorder.getRecordingState() ==
                                AudioRecord.RECORDSTATE_RECORDING) {
                                ......
                                stopSelf();
                                }
                        }
                Log.e(AdhocNetworkActivity.TAG, "Period notf: " +
                        count++);
                if (isRecording() == false) {
                        c = count + 2;
                        }
                }

                @Override
                public void onMarkerReached(AudioRecord recorder) {}
});
```

In the final part, recording state is checked whether it is still recording or stopped. If it is still recording, additional functions `stop()` and `release()` are invoked for ensuring the process ended. In both cases, writing process runs. Below, `dosFile. writeShort(data[i])` function writes the specified data to file.

```
for (int i = 0; i < data.length; i++) {
        try {
                dosFile.writeShort(data[i]);
        } catch (IOException e) {
                e.printStackTrace();
        }
}
```

4 Conclusion

This study aimed to solve problem of exchanging data with a group of people where connecting to a network access point or 3G is not always possible in particular areas. This problem is solved with this study through mobile application using WiFi Direct. Therefore, goal is met with this application where peer connection and data transfer is obtainable.

This section presents the evaluation of the project by few simple facts in order to sum up the work. This process covers the actual purpose of the project and the achievements that have been reached throughout the study. Section also intends to provide current issues of the project and possible solutions to these problems. Furthermore, it consists future work.

4.1 Result

The purpose of the project is to establish connection among mobile device and maintain file and audio transfer without using any 3G service or access point. In addition to that, study is developed to guide the future developers regarding the overview of the system, and finally denote encountered issues. With the help of this document, user can obtain useful and adequate information of the study about implementation as well as theory and background.

WiFi Direct technology has been researched and well grasped throughout the study. Relevant application is developed with respect to study purpose. P2P connection is maintained for Android phone and peer discovery is succeeded. Moreover, connection is established successfully among two devices and most importantly data transfer is realized. Subsequently, audio data is recorded on one device, transferred to other peer and is played.

Although the study's many utilities such as file transfer, peer discovery and connection, it also has few drawbacks when it comes to audio transfer. Project does not provide a satisfactory solution in terms of live streaming. Instead, audio data is recorded and stored in a file, and this file is sent to peer through sockets. Socket connection is obtained by the same method as file transfer.

As a result, the project assures a practical solution to the problem regardless of the minor issues.

4.2 Discussion

There are essentially five issues that still remain as possible problem and need to be resolved. Additionally, there is an aspect which is not a major issue but can be considered as a minor inconvenience. This inconvenience is basically about messages provided to user when a problem occurs. Messages can be reviewed in order to maintain a user-friendly output.

Below is the list of current issues along with extra information regarding the problem itself.

i. *Removing groups*: After establishing connection and transferring data successfully, disconnect operation does not respond to users needs on one side. For instance when disconnect button is pressed, groups are actually removed but menu still shows that peer is still connected. This is not only a layout but also discovery issue which essentially works through broadcast receivers. So, when the group is removed, system automatically searches for peers and discovers the peer instantly. Therefore peer list is not refreshed and reconnection can only be established when the application is restarted.

Possible solution might be adding exception to broadcast receiver where connection and peer status is checked.

ii. *Playing audio data*: Track is played on server side after data transfer as expected. However, audio data plays until all buffer is read completely. In other words, if client records audio for 5 seconds with frequency of 256 kHz; 22k of buffer size is needed. This means 22k of buffer size is stored in audio data file regardless of actual size of data. When this data is played on client side, it does not stop even recorded part is complete. This leads to unnecessary waiting on listening - server - side and therefore does not let user make transfer right after the previous session.

Possible solution could be working around on `onPeriodNotification()` callback under `playAudio()` method for controlling track position and notifying `AudioTrack` class when it reaches the desired position.

iii. *Peers*: One other issue is more about characteristic of `WiFiP2pManager` class. After a successful connection and possible data transfer, peers remain connected for a limited time, which is around 5 minutes. During this period, peers can connect-disconnect and transfer data several times. However, discovery operation needs to be invoked again for searching peers after 5 minutes. This leads to inconsistency in system and restrains constant connection ability.

There is currently no possible solution for this issue. Yet a new release of WiFi Direct API might resolve it.

iv. *Live stream*: This project does not provide live stream audio data transfer even though it was previously desired. In fact audio transfer was first developed in order to maintain live streaming. However, buffering errors are experienced in respect of reading data on server side. It has been observed that data is read only partially through input stream via sockets. In other words, `read(buffer, 0, buffer.length)` method reads only data part in byte array, `buffer`, while it is essentially supposed to be done including empty arrays. Here, empty arrays have essential part, e.g. buffer size in terms of playing audio track. Buffer size is expected to have double size of actual data size in array. In this regard when array does not consist empty arrays, `write()` method and eventually `play()` method only perform half of track playing.

Possible solution could be adding `0` values to array after reading it from stream with the method `read(buffer, 0, buffer.length)`, copy the new array as another instance of array and play this new instance including enough buffer size as well as actual audio data.

v. `aRecorder.read()`: Final issue is about stopping reading data from array prematurely. In other words, audio recording occasionally can not be stopped before buffer size limit is reached. When user wants to finalize recording process and presses stop button, action occasionally does not stop even `recorder.stop()` and `recorder.release()` methods are invoked. Furthermore, this issue can not get caught by checking recording state of the core class `AudioRecord` and thus service can not be stopped explicitly. This issue leads to not proceeding to the next step which is socket connection. And certainly data transfer can not be completed without socket connection, which is a major issue for this study.

There is currently no possible solution found regarding this problem. However, it is suspected that working with multiple threads causes concurrency problems and possible improvements on thread handling might resolve this issue.

In addition to these issue above, there are also some portions which either used to be an issue but solved or still remain as a functionality issue.

WiFi Direct API issue was once a big error for entire WiFi Direct world. This was essentially due to some stability issues of a Broadcom driver issue on ICS release. Problem was briefly about re-connection process. Devices were turning off in every 4th connection process and a new connection can only be established once the device is turned on again. Fortunately, this has been fixed on Android OS 4.1 JB release with feature enhancements and API updates.

As for functionality issue, application has got limited recording time. Maximum recording time is measured as 32 seconds with values of 512 kHz sample rate and 8k of buffer size. Although this does not excessively affect the mechanism of the project, it causes functionality issues when user desires to record data longer than 32 seconds. This could be resolved by implementing live streaming transfer.

4.3 Future Work

The study needs to be reviewed and re-handled with respect to current issues as well as new improvements on Android WiFi Direct API. For instance live streaming process can be re-developed with respect to offered solutions or by using other methods. Moreover, there are few more subjects that can be added into the system in order to insert new features and increase usability. Code duplication analysis can also be considered as a future work for maintaining solid program.

A simple and basic message sending tool can be added to the application where peers can send and receive instant messages among each other. Technically this is much easier process compared to file or audio transfer. Passing strings through sockets can easily be done by using various specific methods of input - output stream classes. Another feature can be adding internet sharing among devices. Currently, WiFi Direct API does not provide this feature for Android system. However, WiFi Direct itself practically offers internet sharing in other platforms. For this reason this improvement can be stated as a future work. By doing so, as for further investigation, peers will be able to share their internet connection in particular area and this sharing can advance to other peers where eventually everyone in the group can use one internet connection without paying extra fees. Subsequently, Controller package and message passing system might be reviewed as for creating couple of instances of related classes. In this regard, re-organizing Controller class and providing more detailed system could be counted as a future work.

References

developer.android.com. Activity life cycle. Available at: http://developer.android.com/reference/android/app/Activity.html

developer.android.com. ArrayAdapter. Available at: http://developer.android.com/reference/android/widget/ArrayAdapter.html [Accessed August 2012]

developer.android.com. bind(). Available at: http://developer.android.com/reference/java/net/Socket.html#bind(java.net.SocketAddress) [Accessed August 2012]

developer.android.com. BroadcastReceiver. Available at: http://developer.android.com/reference/android/content/BroadcastReceiver.html [Accessed August 2012]

developer.android.com. DataInputStream. Available at: http://developer.android.com/reference/java/io/DataInputStream.html [Accessed August 2012]

developer.android.com. Fragment. Available at: http://developer.android.com/reference/android/app/Fragment.html [Accessed August 2012]

developer.android.com. InputStream. Available at: http://developer.android.com/reference/java/io/InputStream.html [Accessed August 2012]

developer.android.com. Intent. Available at: http://developer.android.com/reference/android/content/Intent.html [Accessed August 2012]

developer.android.com. read (byte[], int, int). Available at: http://developer.android.com/reference/java/io/InputStream.html#read(byte[], int, int) [Accessed August 2012]

developer.android.com. registerReceiver. Available at: http://developer.android.com/reference/android/content/ContextWrapper.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter) [Accessed August 2012]

developer.android.com. WifiP2pConfig. Available at: http://developer.android.com/reference/android/net/wifi/p2p/WifiP2pConfig.html [Accessed August 2012]

developer.android.com. Wi-Fi Direct API. Available at: http://developer.android.com/guide/topics/connectivity/wifip2p.html [Accessed August 2012]

developer.android.com. WifiP2pManager. Available at: http://developer.android.com/reference/android/net/wifi/p2p/WifiP2pManager.html [Accessed August 2012]

developer.android.com. write (byte[], int, int). Available at: http://developer.android.com/reference/java/io/OutputStream.html#write(byte[], int, int) [Accessed August 2012]

Donahue G. 2007. Network Warrior. O'Reilly. p. 5 by Gary A. Donahue. [Accessed August 2012]

Harbison, N. 2011. The Fragmentation Of Communication Apps - by Niall Harbison. Available at: http://www.simplyzesty.com/mobile/the-fragmentation-of-communication-apps/

Hurt, J. 2011. Why People Join Social Networking Sites - by Jeff Hurt. Available at: http://jeffhurtblog.com/2011/01/05/why-people-join-social-networking-sites/

Kabani, S. 2010. The Zen of Social Media Marketing: An Easier Way to Build Credibility, Generate Buzz, and Increase Revenue - by Shama Kabani.

Mitchell, B. What is Ad-Hoc Mode in Wireless Networking? - by Bradley Mitchell. Available at: http://compnetworking.about.com/cs/wirelessfaqs/f/adhocwireless.ht m [Accessed August 2012]

Nielsen Company, 2010. Games Most Popular Mobile App. Available at: http:// www.marketingcharts.com/direct/games-most-popular-mobile-app-13104/

Schoder, D et al. 2005. Core Concepts in Peer-to-Peer Networking - by Detlef Schoder, Kai Fischbach, Christian Schmitt. Available at: http://www.econbiz.de/ archiv1/2008/42151_concepts_peer-to-peer_networking.pdf

Shankland, S. 2007. Google's Android parts ways with Java industry group - by Stephen Shankland CNET News. Available at: http://news.cnet.com/8301-13580_3-9815495-39.html

Smith C., Collins D. 2000. 3G Wireless Networks, page 136 - by Cling Smith, Daniel Collins. [Accessed August 2012]

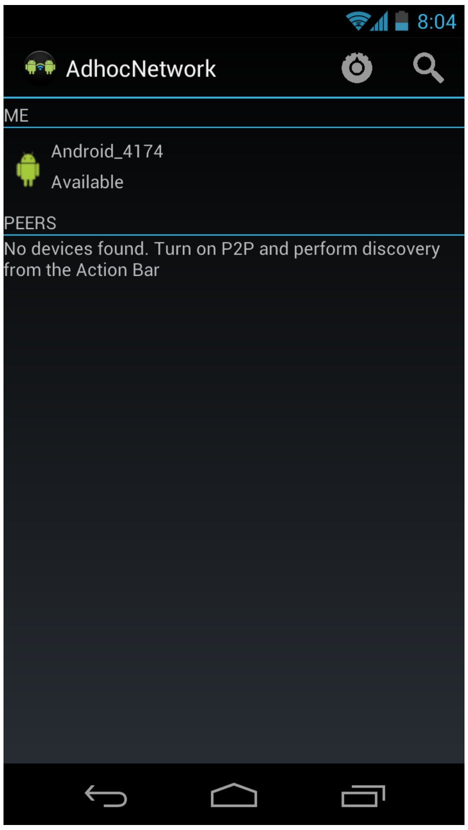
Rapoza, J. 2010. WiFi Direct Could Surpass Bluetooth - by Jim Rapoza. Available at: http://www.informationweek.com/hardware/virtualization/wifi-direct-coul d-surpass-bluetooth/227900750

wi-fi.org. Wi-Fi Direct^{TM}. Available at: http://www.wi-fi.org/discover-and-learn/wi-fi-direct [Accessed August 2012]

Zibreg, C. 2010. Android 2.2 to support tethering and turn your phone into a mobile WiFi hotspot - by Christian Zibreg. Available at: http://www.geek.co m/articles/mobile/android-2-2-to-support-tethering-and-turn-your-phone-into-a-mo bile-wifi-h otspot-20100514/ [Accessed August 2012]

Appendix A: Screenshots

A1: Client Main Screen



8:04

AdhocNetwork

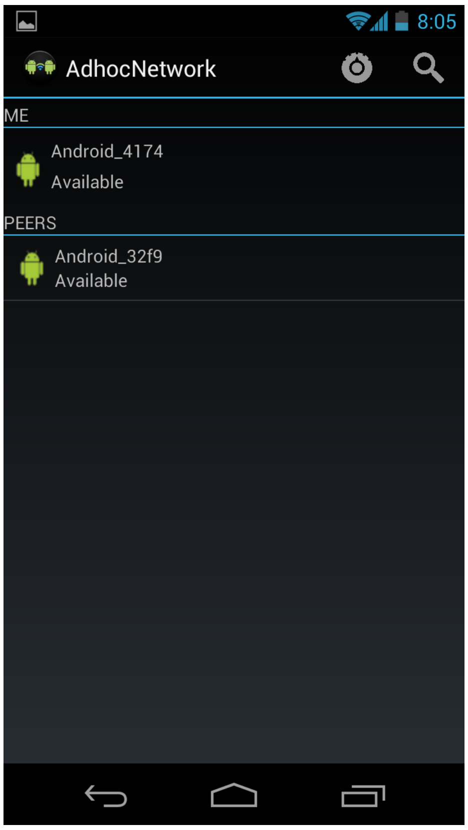
ME

Android_4174
Available

PEERS

No devices found. Turn on P2P and perform discovery from the Action Bar

A2: Client With Peer Discovered



8:05

AdhocNetwork

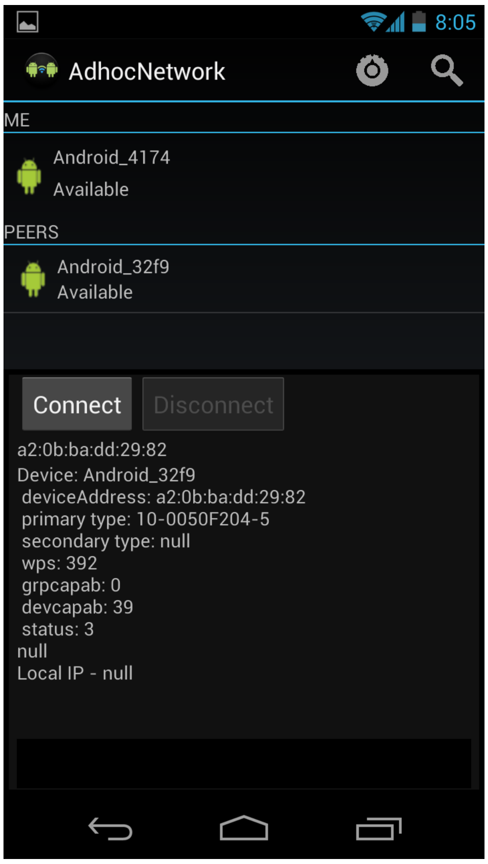
ME

Android_4174
Available

PEERS

Android_32f9
Available

A3: Client Connection Menu

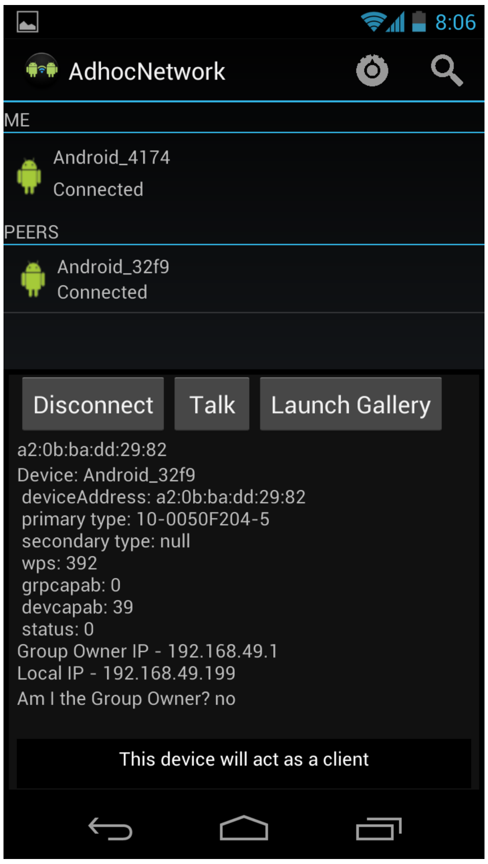


Android_4174
Available

Android_32f9
Available

```
a2:0b:ba:dd:29:82
Device: Android_32f9
 deviceAddress: a2:0b:ba:dd:29:82
 primary type: 10-0050F204-5
 secondary type: null
 wps: 392
 grpcapab: 0
 devcapab: 39
 status: 3
null
Local IP - null
```

A4: Client Connection Details



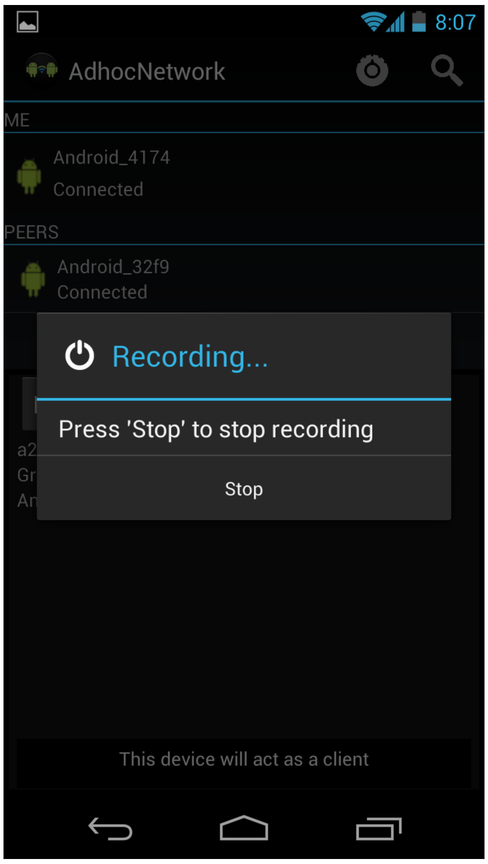
Android_4174
Connected

Android_32f9
Connected

```
a2:0b:ba:dd:29:82
Device: Android_32f9
 deviceAddress: a2:0b:ba:dd:29:82
 primary type: 10-0050F204-5
 secondary type: null
 wps: 392
 grpcapab: 0
 devcapab: 39
 status: 0
Group Owner IP - 192.168.49.1
Local IP - 192.168.49.199
Am I the Group Owner? no
```

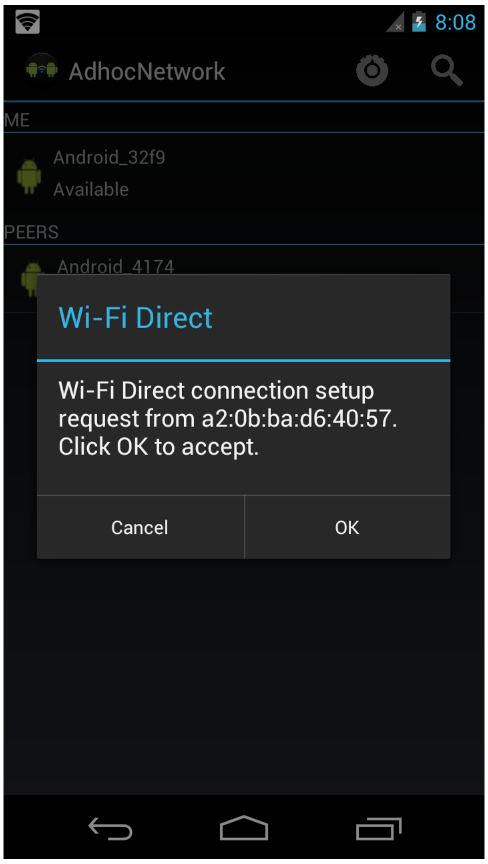
This device will act as a client

A5: Client Audio Recording



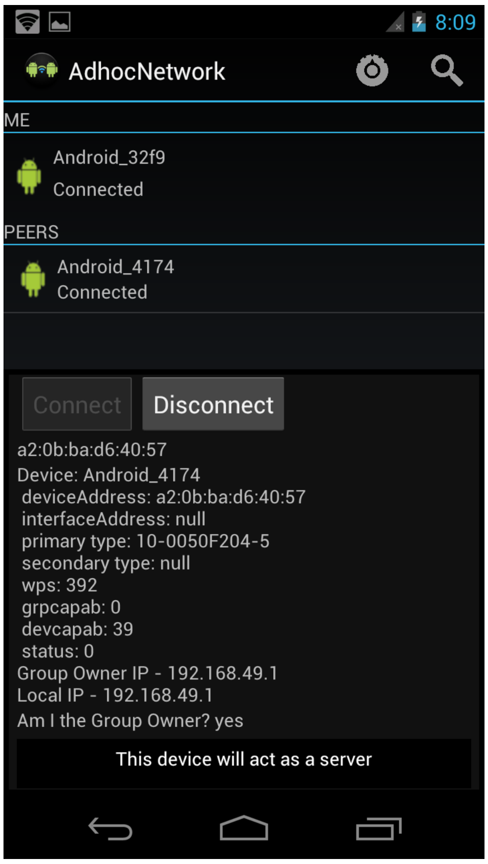
Press 'Stop' to stop recording

A6: Server Connection Notification



Wi-Fi Direct connection setup request from a2:0b:ba:d6:40:57. Click OK to accept.

A7: Server Connection Details



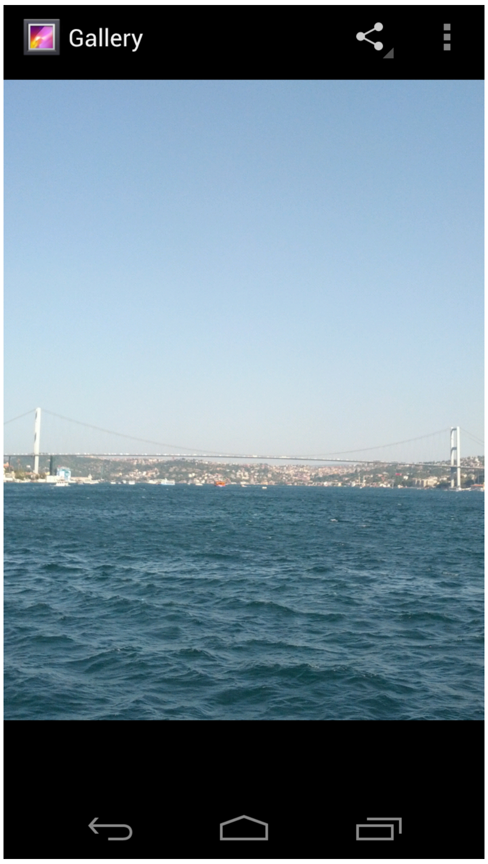
Android_32f9
Connected

Android_4174
Connected

```
a2:0b:ba:d6:40:57
Device: Android_4174
 deviceAddress: a2:0b:ba:d6:40:57
 interfaceAddress: null
 primary type: 10-0050F204-5
 secondary type: null
 wps: 392
 grpcapab: 0
 devcapab: 39
 status: 0
Group Owner IP - 192.168.49.1
Local IP - 192.168.49.1
Am I the Group Owner? yes
```

This device will act as a server

A8: Server Showing Transferred Image



Linnæus University
School of Computer Science, Physics and Mathematics

SE-391 82 Kalmar / SE-351 95 Växjö
Tel +46 (0)772-28 80 00
dfm@lnu.se
Lnu.se/dfm