2017

Direct

A LIBRARY TO SIMPLIFY THE ANDROID WI-FI PEER-TO-PEER API

TYLER MCBRIDE | 100888344

https://github.com/tylerjmcbride/Direct

CARLETON UNIVERSITY

Contents

Preamble

Direct is a library that I have created to provide a simplified interface to wrap around the Wi-Fi Peer-to-Peer API. In essence, this interface acts as a facade around the Wi-Fi Peer-to-Peer API by hiding its implementation details.

Wi-Fi Direct has peer-to-peer functionality, hence initially being called Wi-Fi Peer-to-Peer; however, this library will be designed to support a client server architecture instead. In order to establish connections and send data amongst these connections, Wi-Fi Direct provides an abundance of details that are of little actual use to the developer.

This library abstracts this abundance of details in order to enable the developer to implement Wi-Fi Direct without having to be aware of the underlying functionality; thus, reducing the both the technical load on the developer and the potential for bugs. Ultimately, this library frees the developer of the specific details of the Wi-Fi Direct API.

What is Wi-Fi Direct?

Wi-Fi Direct is a fairly recent technology which enables Wi-Fi compliant devices to make direct connections without the need for an intermediate access point. In other words, these devices may connect directly to one another without an internet connection. Wi-Fi Direct connections may be made anywhere at any time as these connections are not dependent on a Wi-Fi network. (Alliance, n.d.)

Android's Wi-Fi P2P Framework

Android provides a Wi-Fi P2P framework which complies with the Wi-Fi Alliance's Wi-Fi Alliance's Wi-Fi Direct™ certification program. Inclusively, Android 4.0 or later devices are all built with the appropriate hardware to support Wi-Fi Direct. (Android, n.d.)

Android supplies several APIs that enable Wi-Fi Direct discovery and connections to other devices whom all support Wi-Fi Direct. The WifiP2pManager class in the android.net.wifi.p2p package supplies methods for service discovery, peer discovery, and connection requests. These methods all accept listeners which notify of the success or failure of respective calls. Events detected by the Wi-Fi P2P framework produce intents that notify of changes, such as a dropped connection, or a discovered peer. (Android, n.d.)

There are three main reasons for using Android's Wi-Fi P2P framework. Wi-Fi Direct is useful for applications that share data among users, it uses direct connections via Wi-Fi without an intermediate access point, and last but not least it reaches distances much longer than any Bluetooth connection. (Android, n.d.)

The Solution

Despite support of Wi-Fi Direct being released on Android 4.0 in October 2011, the Wi-Fi Direct API continues to be very complex and difficult to understand. Many developers steer clear of Wi-Fi Direct due to its complex nature and confusing documentation. I wanted to reduce these deterrents from preventing the use of Wi-Fi Direct in future android development.

My solution was to create the Direct library in order to wrap the Android Wi-Fi Peer-to-Peer API producing a much more intuitive API. Unfortunately, as a consequence of this library abstracting details of the Android Wi-Fi Peer-to-Peer API there is a loss of functionality; therefore, developers making use of this library will not be able to customize their application to the full extent that using the raw Android Wi-Fi Peer-to-Peer API provides.

Alternative Solutions

There do exist existing solutions that solve the issue with the Android Wi-Fi Peer-to-Peer API; however, these solutions are less than ideal for my standards. These other libraries lacked documentation and proved to be much too difficult to start a project with.

The Salut library was the only library that worked for me, which I used for an Android application I made in an Object-Oriented Software Engineering course; however, I was forced to fork the library as I encountered a number of bugs some of which caused my application to crash. Forking this library proved to be difficult due to the lack of documentation, making debugging a nightmare. In summary, this library did not live up to the standard of code that I come to expect from an external library.

This inspired me to create my own library that not only provided a solution to my problem but is compliant with general coding standards and is well documented. As a developer could use this library or the Salut library interchangeably, it is worth noting the subtle differences between the two libraries.

The Salut library and this library use different formats for transferring data. The Salut library uses JSON (JavaScript Object Notation) while this library uses serialized Java objects. The Salut library serializes Java objects into JSON through the use of the LoganSquare library. This requires Java objects to be annotated with annotations such as @JsonObject and @JsonField, as well as many other nuances. For example, private fields must implement a getter and setter.

This library on the other hand only requires that the Java objects implement the Serializable interface. The serializable interface is great for versioning, allowing a newer application version to de-serialize previous version models. The main difference between these two formats is that the Java ObjectInputStream and ObjectOutputStream allow any anonymous object to be communicated, while the use of LoganSquare does not allow for anonymous classes to be sent, the host is require to know exactly which class is being received.

The creator of believes that the host receiving data in a single Java object “is particularly useful because it means that you can create a sort of God object that will hold all your data types and is serializable.” (markrjr, n.d.) I wholeheartedly disagree with this statement as it disregards the benefits of object orientated programming. The JSON could instead contain a header indicating its type; however, I believe that this solution is messy and feel much more comfortable using the built in functionality of Java serialized objects.

The sole issue that communication with serialized Java objects is that iOS does not support Java objects as these applications run with Objective-C. With that being said, iOS does not official support Wi-Fi Direct; therefore, an iOS device would not be able to host a service anyways.

One major issue I have with the Salut library is that it uses a single reference to the Activity object passed in on instantiation. This single instance is used to post synchronous tasks to the main thread; however, switching activities will produce unexpected results because the activity will be killed in order to conserve memory.

To prevent these unexpected results, the library I have implemented will use only the application context instead to post synchronous tasks to the main thread; therefore, switching activities will not cause any problems as the application context will persist throughout the entire lifecycle of the application and remains unaffected through switching activities.

The Salut library is riddled with code smell. Examples includes but is not limited to: the bloated Salut class has over eight-hundred lines of code, code duplication, public state variables that if modified could put the application in an unpredictable state, and last but not least returning references to objects rather than cloning said objects.

In addition, no method in the Salut library contains any Javadoc; therefore, as Javadoc is recognized by the IDE the developer will only have the method names themselves to provide context. The library I have implemented has Javadoc for each any every method, even private and protected ones. For example,

```
/**
 * Returns a deep copy of the list of client WifiP2pDevices.
 * @return A deep copy of the list of client WifiP2pDevices.
 */
public List<WifiP2pDevice> getRegisteredClients() {
    List<WifiP2pDevice> deepClientsClone = new ArrayList<>();
    for(WifiP2pDevice device : clients.values()) {
        deepClientsClone.add(new WifiP2pDevice(device));
    }
    return deepClientsClone;
}
```

The Javadoc refers to the comment block above any method. This Javadoc serves as documentation for which the IDE will display when a developer hovers over the method. It is

obvious that this Javadoc provides useful and necessary information to the developer. This will also benefit the developer if for whatever reason they are required to debug the implementation that I have provided.

While the Salut library does provide a certain degree of abstraction, it definitely does not take advantage of its full extent. The bloated Salut class is used for both client and host implementations; therefore, it contains the behaviour and class members for both client and host. For example, only the host needs a WifiP2pDnsSdServiceInfo to pass to clients discovering the respective service but the Salut class contains this information in the client class.

In the Salut library, the code for socket instantiation and communication is duplicated for both the registration and data communication processes while the library that I implemented abstracts this functionality into a few utility classes. The ServerSockets class abstracts server socket initialization while both the ServerSocketRunnable and SocketConnectionRunnable classes listen for connections and establish connections respectively. These three classes are used both by the registration and data communication processes as they both abstract common functionality.

In terms of external libraries, the Salut library requires both LoganSquare and AsyncJobLibrary. These libraries must be included in the Gradle build file otherwise the application will be unable to run. I believe that these dependencies are an unnecessary step, and are difficult to include in an offline environment where Gradle doesn't have access to the internet. The library I have implemented requires no external libraries; therefore, removes the tedious task of including these external libraries in the Gradle build file.

Comparison Chart

The following comparison chart will both illustrate and summarize the differences discussed in the previous section between these two libraries.

Contains	Salut	Direct
Code Smell	An abundance of code smell. For example, the bloated Salut class has over eight-hundred lines of code.	Contains arguably no code smell.
Javadoc	Literally no method contains any Javadoc whatsoever.	Virtually every method contains Javadoc, even including private and protected methods.
Abstraction	Contains minimal abstraction; the universal Salut class contains the behaviour and class members for both client and host.	Contains a significant degree of abstraction; both the distinct behaviour and class members of client and host are encapsulated from one another.
Data Transfer	Transferring of Java objects is done with JSON and requires the host to know exactly which data type to receive. These objects must be marked with annotations from the LoganSquare library. The receiving functionality is left for the developer to define.	Transferring data is done with Java objects and any serialized java object may be sent anonymously. No annotations required. The receiving functionality is left for the developer to define. The receiving functionality is defined by the library itself.
External Libraries	Requires both LoganSquare and AsyncJobLibrary. These libraries must be included in the gradle dependencies.	Does not require any external libraries. No libraries will need to included in the gradle dependencies.
Context Switching	The library uses a reference to an Activity to post synchronous tasks to the main thread; however, switching activities will produce unexpected results.	This library uses only the application context to post synchronous tasks to the main thread; therefore, switching activities will not cause any problems.

Basic Usage

This section will explain how to use this library to quickly set up a Wi-Fi Direct enabled application to send and receive serializable Java objects.

Initial Setup

Setting Permission & Minimum SDK Version

Before getting starting, the following permissions must be explicitly stated within the Android manifest. These permissions are required to ensure that the application may use the Wi-Fi hardware. A minimum SDK version of 14 is required as only Android 4.0 or later devices are built with the appropriate hardware to support Wi-Fi Direct.

```
<uses-sdk android:minSdkVersion="14" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Initialization

It is recommended that the initialization is done in an Application class as it is necessary that the class persists between context switches.

Host Initialization

Below is the code to initialize a host, it is worth noting that this instance should persist through switching context as this instance registers itself within the application context. The service tag is the unique identifier for the application. This tag should be unique, as to prevent confusion between different services being run on different applications. The instance tag should attempt to be unique, but is not necessary.

```
Host host = new Host(getApplication(), "UNIQUE_SERVICE_TAG",
"UNIQUE_INSTANCE_TAG");
```

Client Initialization

Below is the code to initialize a client, it is worth noting that this instance should persist through switching context as this instance registers itself within the application context. The service tag **must** equal the service tag that the host implements.

```
Client client = new Client(getApplication(), "UNIQUE_SER-
VICE_TAG");
```

Service Management

Starting a Service

Below is the code for a host to create a service and advertise said service to client devices. More details after code snippet.

```
host.startService(new ObjectCallback() {
    @Override
    public void onReceived(Object object) {
        // Object received from client
    }
}, new ClientCallback() {
    @Override
    public void onConnected(WifiP2pDevice clientDevice) {
        // Client has connected
    }

    @Override
    public void onDisconnected(WifiP2pDevice clientDevice) {
        // Client has disconnected
    }
}, new ServiceCallback() {
    @Override
    public void onServiceStopped() {
        // The service has stopped, possibly unexpectedly
    }
}, new ResultCallback() {
    @Override
    public void onSuccess() {
        // Succeeded to request for the service to be started
    }

    @Override
    public void onFailure() {
        // Failed to request for the service to be started
    }
});
```

The ObjectCallback is called whenever the host receives an object from a client. The ClientCallback is called whenever a client connects or disconnects from the host. The ServiceCallback is called when the service has stopped, which is useful in the event where the service may have stopped unexpectedly. It is important that any callback does not contain any references to objects that will not persist between a switch in context.

Stopping a Service

Below is the code for a host to stop a service and discontinue advertisement to client devices, effectively closing all client connections. If this request is successful, it will trigger the `onServiceStopped()` method in the ServiceCallback provided when starting the service as the peer group will cease to exist.

```
host.stopService(new ResultCallback() {
    @Override
    public void onSuccess() {
        // Succeeded to request for the service to be stopped
    }

    @Override
    public void onFailure() {
        // Failed to request for the service to be stopped
    }
});
```

Discovering Services

Below is the code for a client to discover services. The DiscoveryCallback will be called whenever a new host has been discovered. If the client would rather see the entire list of available hosts, call `client.getNearbyHosts()`.

```
client.startDiscovery(new DiscoveryCallback() {
    @Override
    public void onDiscovered(WifiP2pDevice hostDevice) {
        // New service discovered
    }
}, new ResultCallback() {
    @Override
    public void onSuccess() {
        // Succeeded to start discovery
    }

    @Override
    public void onFailure() {
        // Failed to start discovery
    }
});
```

Connecting to a Service

After the client has discovered a service, that client may now connect to that service with the respective host WifiP2pDevice.

The ObjectCallback is called whenever the client receives an object from the host. The ConnectionCallback is called when the client officially establishes connection—the request itself is not enough to guarantee success—with the host, or when the client loses said connection with the host. It is important that either callback does not contain any references to objects that will not persist between a switch in context.

```
client.connect(hostDevice, new ObjectCallback() {
    @Override
    public void onReceived(Object object) {
        // Object received from host
    }
}, new ConnectionCallback() {
    @Override
    public void onConnected() {
        // Connected to host
    }

    @Override
    public void onDisconnected() {
        // Disconnected with host, perhaps the host discontinued
the service?
    }
}, new ResultCallback() {
    @Override
    public void onSuccess() {
        // Connection request succeeded
    }

    @Override
    public void onFailure() {
        // Connection request failed
    }
});
```

Disconnecting from a Service

When the client would like to disconnect from the host, that client may easily disconnect by the following method The ResultCallback does not guarantee a successful disconnect, this event will be captured in the ConnectionCallback provided in the connect method.

```
client.disconnect(new ResultCallback() {
    @Override
    public void onSuccess() {
        // Succeeded to request a disconnect from the service
    }

    @Override
    public void onFailure() {
        // Failed to request a disconnect from the service
    }
});
```

Sending Objects

Sending an Object to a Client

After a client has connected, the host may send a serializable object to said client.

```
host.send(clientDevice, serializableObject, new ResultCallback()
{
    @Override
    public void onSuccess() {
        // Succeeded to send object
    }

    @Override
    public void onFailure() {
        // Failed to send object
    }
});
```

Sending an Object to the Host

After a client has connected with the host, that client may send a serializable object to said host.

```
client.send(serializableObject, new ResultCallback() {
    @Override
    public void onSuccess() {
        // Succeeded to send object
    }

    @Override
    public void onFailure() {
        // Failed to send object
    }
});
```

Internal API

This section will explore in depth the underlying functionality that each class within the library provides. Before reading the following documentation, it is important to understand that all of the Wi-Fi P2P framework methods are one way communication; in other words, any method called with the Wi-Fi P2P framework is only a request, and it's success only reflects the success of the hardware receiving said request, not that the request has been fulfilled. The state of the hardware is only available through the DirectBroadcastReceiver.

For example, calling connect from the WifiP2pManager will only send the hardware the request to connect to the respective WifiP2pDevice. The only way to determine if the connection has been establish is through the DirectBroadcastReceiver receiving a WIFI_P2P_CONNECTION_CHANGED_ACTION intent.

DirectBroadcastReceiver

`abstract class DirectBroadcastReceiver extends BroadcastReceiver`

The `DirectBroadcastReceiver` class extends BroadcastReceiver, which receives and handles broadcasts intents. The BroadcastReceiver is essentially the means of reacting to changes in the Android system. In particular, there are five intents that the `DirectBroadcastReceiver` is interested in:

- `WIFI_P2P_STATE_CHANGED_ACTION`
  - This action indicates whether Wi-Fi P2P is enabled or disabled.
- `WIFI_P2P_DISCOVERY_CHANGED_ACTION`
  - This action indicates whether the peer discovery has either been started or stopped.
- `WIFI_P2P_PEERS_CHANGED_ACTION`
  - This action indicates that the available peer list has changed. The peer list will be changed when peers are lost, found, or updated. This will be exclusively used by the host to unregister clients who have been lost.
- `WIFI_P2P_CONNECTION_CHANGED_ACTION`
  - This action indicates that the Wi-Fi P2P connectivity has changed. This will be used to get a handle on the:
    - WifiP2pInfo
      - This class represents Wi-Fi P2P group connection information. This class contains the field groupFormed which indicates whether a Wi-Fi P2P group has been successfully formed. This class also contains the field groupOwnerAddress which may be used to retrieve the host IP address, which is necessary for the client to register with the host.
    - NetworkInfo
      - This class represents the current network connection. This class is used to call the method isConnected() to determine whether the current device has established a connection and is able to perform data transactions.
    - WifiP2pGroup
      - This class represents the current Wi-Fi P2P group. This group consists of the group owner and one or more clients. In particular, this class will be used call the method getOwner() in to retrieve the host WifiP2pDevice.
- `WIFI_P2P_THIS_DEVICE_CHANGED_ACTION`
  - This action indicates that the Wi-Fi P2P device has changed. This will be used to get a handle on the current WifiP2pDevice.

The BroadcastReceiver requires the extending class to implement the abstract method onReceive(), which is the method that handles received intents; however, this method generally promotes the If-Then-Else code smell, you may see this code smell in the official Android documentation for creating a broadcast receiver. To combat this, the DirectBroadcastReceiver splits the abstract method onReceive() into method into the following more intuitive methods:

- `stateChanged(boolean wifiEnabled)`
  - This method is called when the `WIFI_P2P_STATE_CHANGED_ACTION` intent has been broadcasted.
- `discoveryChanged(boolean discoveryEnabled)`
  - This method is called when the `WIFI_P2P_DISCOVERY_CHANGED_ACTION` intent has been broadcasted.
- `peersChanged()`
  - This method is called when the `WIFI_P2P_PEERS_CHANGED_ACTION` intent has been broadcasted.
- `connectionChanged(WifiP2pInfo p2pInfo, NetworkInfo info, WifiP2pGroup p2pGroup)`
  - This method is called when the `WIFI_P2P_CONNECTION_CHANGED_ACTION` intent has been broadcasted.
- `thisDeviceChanged(WifiP2pDevice thisDevice)`
  - This method is called when the `WIFI_P2P_THIS_DEVICE_CHANGED_ACTION` intent has been broadcasted.

HostRegistrar

```
class HostRegistrar
```

The HostRegistrar is in charge of handling the registration of clients, this class will be used by the Host in order to encapsulate the registration functionality.

start

```
void start(ServerSocketInitializationCompleteListener
initializationCompleteListener)
```

This method will start the registration process. To begin, this method will initialize a new ServerSocket which will accept incoming client connections; however, the ServerSocket has a blocking method which requires the registration process to run on a separate Thread. This blocking method is the ServerSocket accept method as it must wait for a request to come in over the network. To process these requests, the registrar will then spawn another new thread for each individual client request through an ExecutorService. The communication between these sockets are done through the use of an ObjectInputStream and an ObjectOutputStream.

These client requests aim to either register or unregister said clients from the host. Clients who wish to register will send a Handshake object and clients who wish to unregister will send over an Adieu object. In the case of a client registering, the host will then reply with its own Handshake object.

Both the Handshake and the Adieu class contain the MAC address of the device and the port number of the ObjectReceiver running on the device, while the IP address can easily be derived from the connecting Socket. This enables both the client and host to establish a connection with one another's ObjectReceiver for the purpose of sending one another objects. Depending on which object is processed the registrar will update the library accordingly.

stop

```
void stop()
```

This method will stop the registration process, the ExecutorService will be shut down, the ServerSocket closed, and the Thread running the registrar will be interrupted; hence why a new ServerSocket will need to be initialized on registrar start. This is to ultimately clean up resources while there are no Wi-Fi P2P services running.

ClientRegistrar

```
class HostRegistrar
```

The ClientRegistrar is in charge of handling registration with the host, this class will be used by the Client in order to encapsulate the registration functionality.

register

```
void register(InetSocketAddress address, RegisteredWithServerListener
registeredWithServerListener)
```

This method will register the client with the appropriate host. To begin, this method will initialize a new Socket which will attempt to connect to the HostRegistrar ServerSocket. This connection is made on a separate Thread to prevent a NetworkOnMainThreadException as a networking operation must be completed to establish a connection.

In order to register, the client will send a Handshake object and then the host will reply with its own Handshake object. This is done in order for the client to retrieve the port that the host ObjectReceiver running on, as well as for the host to retrieve the port that the client ObjectReceiver is running on as well as the client IP address.

Once the transfer of the Handshake objects are complete the ClientRegistrar will close the Socket effectively ending the connection. The communication between these sockets are done through the use of an ObjectInputStream and an ObjectOutputStream.

unregister

```
void unregister(InetSocketAddress address,
UnregisteredWithServerListener unregisteredWithServerListener)
```

This method will unregister the client from the appropriate host. To begin, this method will initialize a new Socket which will attempt to connect to the HostRegistrar ServerSocket. This connection is made on a separate Thread to prevent a NetworkOnMainThreadException as a networking operation must be completed to establish a connection.

In order to register, the client will send an Adieu object to the HostRegistrar. This is done in order to notify the HostRegistrar that the client would like to unregister. The host will have no reply for this action as the client is disconnecting and need no further information.

Once the transfer of the Adieu object is complete the ClientRegistrar will close the Socket effectively ending the connection. The communication between these sockets are done through the use of an ObjectInputStream and an ObjectOutputStream.

ObjectReceiver

```
class ObjectReceiver
```

The ObjectReceiver is in charge of handling incoming objects from an ObjectTransmitter, this class will be used by both the Host and Client in order to encapsulate the object receiving functionality. There is virtually no difference between the functionality that the host uses to receive objects sent by the client compared to the client receiving the objects sent by the host; therefore, both the client and host use the same code to receive objects.

start

```
void start(ObjectCallback objectCallback,
ServerSocketInitializationCompleteListener
initializationCompleteListener)
```

This method will enable the object receiver to accept incoming objects. To begin, this method will initialize a new ServerSocket which will accept incoming connections; however, the ServerSocket has a blocking method which requires the object receiving process to run on a separate Thread. This blocking method is the ServerSocket accept method as it must wait for a request to come in over the network. To process these requests, the registrar will then spawn another new thread for each individual request through an ExecutorService. The communication between these sockets are done through the use of an ObjectInputStream and an ObjectOutputStream.

stop

```
void stop()
```

This method will stop the object receiving process, the ExecutorService will be shut down, the ServerSocket closed, and the Thread running the registrar will be interrupted; hence why a new ServerSocket will need to be initialized on registrar start. This is to ultimately clean up resources while there are no Wi-Fi P2P services running.

ObjectTransmitter

```
class ObjectTransmitter
```

The ObjectTransmitter is in charge of sending objects to an ObjectReceiver, this class will be used by both the Host and Client in order to encapsulate the object transmitting functionality. There is virtually no difference between the functionality that the host uses to send objects to the client compared to the client sending the host objects; therefore, both the client and host use the same code to transmit objects.

send

```
void register(Serializable object, InetSocketAddress address,
SocketInitializationCompleteListener listener)
```

This method will send the respective ObjectReceiver an object. To begin, this method will initialize a new Socket which will attempt to connect to the ObjectReceiver ServerSocket. This connection is made on a separate Thread to prevent a NetworkOnMainThreadException as a networking operation must be completed to establish a connection.

Once the connection is established, the ObjectTransmitter will then send over a Serializable object to the respective ObjectReceiver. Once the object has been received the ObjectTransmitter will close the Socket effectively ending the connection. The communication between these sockets are done through the use of an ObjectInputStream and an ObjectOutputStream.

Direct

`abstract class Direct`

The Direct abstract class contains common behaviour and variables that are inherited by both the `Host` and `Client` classes. In particular this class creates an instance of IntentFilter which listens for the following intents:

- `WIFI_P2P_STATE_CHANGED_ACTION`
- `WIFI_P2P_DISCOVERY_CHANGED_ACTION`
- `WIFI_P2P_PEERS_CHANGED_ACTION`
- `WIFI_P2P_CONNECTION_CHANGED_ACTION`
- `WIFI_P2P_THIS_DEVICE_CHANGED_ACTION`

These actions are received by the abstract class `DirectBroadcastReceiver` which is extended by an anonymous class within both the `Host` and `Client` constructors and registered with the application context. While this class is used by both the `Host` and `Client` classes, they have different implementations of the five methods the `DirectBroadcastReceiver` provides. The reason why these classes are anonymously extended is that they require access to private members of both the `Host` and `Client` classes.

Apart from creating the IntentFilter, the Direct class initializes the Channel which is the link that connects the given application to the Wi-Fi P2P framework. This class will also use the respective Application to retrieve the application context in order to create a Handler in order for asynchronous methods to post a Runnable to the main thread.

Host

```
class Host extends Direct
```

The Host class is responsible for hosting services.

Host Constructor

```
Host(Application application, String service, String instance)
```

This constructor will create an instance of an anonymous class inheriting from DirectBroadcastReceiver and register said instance with the application context. This constructor will create an instance of `Map<String, String> record` to store within an instance of WifiP2pDnsSdServiceInfo.

The WifiP2pDnsSdServiceInfo will eventually be passed to clients who are discovering the respective service. In particular, an entry will be put into the `Map<String, String> record` with the `SERVICE_NAME_TAG` key with the value as the respective `service` that will be hosted. This constructor will also create an instance of HostRegistrar to handle the registration of clients and ObjectReceiver to receive data from said clients.

startService

```
void startService(ObjectCallback dataCallback, ClientCallback
clientCallback, ServiceCallback serviceCallback, ResultCallback
callback)
```

This method will begin by clearing all local services, or in other words, stopping any previously existing service that the host may be providing.

Afterwards, the method will start the ObjectReceiver and HostRegistrar, initializing their respective ServerSockets. The host now has a handle on the HostRegistrar's ServerSocket port as it has been initialized. This port number will be put into the `Map<String, String> record` with the `REGISTRAR_PORT_TAG` as the key. Then the host instance of WifiP2pDnsSdServiceInfo will be updated to reflect the updated `record`.

Once the above has been successfully completed, this method will add the local service accompanied by the WifiP2pDnsSdServiceInfo for service discovery. This is important as the clients will use the WifiP2pDnsSdServiceInfo to look at the `Map<String, String> record` on discovery.

stopService

```
void stopService(final ResultCallback callback)
```

This method will practically work in the reverse order of the `startService` method. This method will remove the local service respective to the WifiP2pDnsSdServiceInfo instance.

After removing the local service, this method will remove the current P2P group, and through reflection, this method will attempt to delete the persistent P2P group as well.

This is because P2P groups are by default persisted in the Wi-Fi P2P framework. By reflection I mean that the method to within the Wi-Fi P2P framework to delete these persistent groups is not visible and must be accessed through reflection. Overall, this method will effectively end the P2P group for all devices that are connected.

send

```
void send(WifiP2pDevice clientDevice, Serializable object, final
ResultCallback callback)
```

With the client IP address and the client ObjectReceiver port obtained from the respective `clientDevice`'s registration handshake, this method will effectively send the respective `object` through the host ObjectTransmitter. The `object` is required to implement Serializable as communication between the ObjectTransmitter and the ObjectReceiver make use of ObjectInputStream and ObjectOutputStream.

Client

```
class Client extends Direct
```

This class is responsible for discovering services, and connecting to said services.

Client Constructor

```
Client(Application application, String service)
```

This constructor will create an instance of an anonymous class inheriting from DirectBroadcastReceiver and register said instance with the application context. The constructor will finally set the DnsSdServiceResponseListener and DnsSdTxtRecordListener to be reused with each service request, more on this will be covered in the explanation of the `startDiscovery` method. This constructor will also create an instance of ClientRegistrar to handle the registration with the HostRegistrar.

startDiscovery

```
void startDiscovery(DiscoveryCallback discoveryCallback,
ResultCallback callback)
```

This method will create a new service request instance and send it to the Wi-Fi P2P framework. If successful, this method will then initiate service discovery. Service discovery is a process that involves scanning for requested services for the purpose of establishing a connection to a peer that supports an available service.

The service discovery notifies the library through the use of both a DnsSdServiceResponseListener and DnsSdTxtRecordListener. For the purpose of this library, only the DnsSdTxtRecordListener is used. This is because the DnsSdTxtRecordListener retrieves the `Map<String, String> record` which contains two entries that are useful to the client. These two entries are `SERVICE_NAME_TAG` and `REGISTRAR_PORT_TAG`. The `SERVICE_NAME_TAG` entry used to filter the discovered services, this entry will contain the unique identifier of the application. The `REGISTRAR_PORT_TAG` will contain the host registrar port, which in combination with the IP address of the host will be used to both register and unregister.

The services discovered which contain the proper `SERVICE_NAME_TAG` will be stored in `Map<WifiP2pDevice, Integer> nearbyHostDevices`, which is a map which the key is the respective host `WifiP2pDevice` and value is the respective registrar port stored in the `REGISTRAR_PORT_TAG`.

stopDiscovery

```
void stopDiscovery(final ResultCallback callback)
```

This method will remove the service request created in `public void startDiscovery(final ResultCallback callback)` if said service request is not null. `Map<WifiP2pDevice, Integer> nearbyHostDevices` will be cleared and all peer discovery will be ceased.

connect

`void connect(WifiP2pDevice hostDevice, ObjectCallback dataCallback, ConnectionCallback connectionCallback, ResultCallback callback)`

This method will attempt to connect to the given `hostDevice` based on a `WifiP2pConfig` consisting of the MAC address from said host device. This method will only attempt to establish this connection if the given host device is contained within `Map<WifiP2pDevice, Integer> nearbyHostDevices`, this is to prevent the client from connecting to host devices that are not running with the proper unique identifier.

This method will not actually establish the connection but rather sends a connection request to the Wi-Fi P2P framework. In the event of a successful connection between the client and host device, a WIFI_P2P_CONNECTION_CHANGED_ACTION intent will be broadcasted and notify the client of a change in connectivity.

When a successful connection is broadcasted, the client will first create an instance of ObjectReceiver, which is essentially a ServerSocket listening for objects to be received from the host. At this point, the client has access to the host device IP address through the WifiP2pGroup. Given the host IP address and registrar port, the client then connects to the HostRegistrar through the ClientRegistrar and sends a Handshake object consisting of the client MAC address and port the client ObjectReceiver is listening on. This is to notify the host of the established connection and to provide the host with means of sending objects to the client ObjectReceiver.

The host will then send a Handshake object in return consisting of the MAC address and port of the host ObjectReceiver is listening on. This is to provide the client of the established connection and to provide the host with means of sending objects to the client ObjectReceiver.

disconnect

`void disconnect(ResultCallback callback)`

This method will begin by unregistering the client from the host, this is done by sending an Adieu object to the host through the ClientRegistrar to the HostRegistrar. This will notify the host that the client is disconnecting to prevent the host form continuing to send objects to the client ObjectReceiver. This is required as the Wi-Fi P2P framework has no reliable functionality to detect client disconnects. The registrar is required to unregister before disconnecting, for when the client leaves the P2P group the ClientRegistrar will no longer be permitted to connect to the HostRegistrar's ServerSocket.

After unregistering, this method will then remove the current P2P group, and through reflection, this method will attempt to delete the persistent P2P group, as P2P groups are by default persisted in the Wi-Fi P2P framework. By reflection I mean that the method to within the Wi-Fi P2P framework to delete these persistent groups is not visible and must be accessed through reflection. This will effectively disconnect the client from the host.

As usual, this method will not actually remove the P2P group but rather sends a request to the Wi-Fi P2P framework. In the event of a successful disconnect between the client and host device, a WIFI_P2P_CONNECTION_CHANGED_ACTION intent will be broadcasted and notify the client of a change in connectivity. When a successful disconnect is broadcasted, all host information will be cleared and the client ObjectReceiver will be stopped.

send

```
void send(Serializable object, ResultCallback callback)
```

With the host IP address and the host ObjectReceiver port obtained from the registration handshake, this method will effectively send the host the respective `object` through ObjectTransmitter. The `object` is required to implement Serializable as communication between the ObjectTransmitter and the ObjectReceiver make use of ObjectInputStream and ObjectOutputStream.

Future Prospects

Improved Testing

This library was only tested with a Nexus 5, HTC One M7, and a Galaxy S6; therefore, it will need to be tested with many more devices. I plan to get my hands on at least a few more devices before publishing this library onto JitPack and advertising it in the community.

In addition, the library will need more unit tests. The reason these tests weren't implemented originally is that the Wi-Fi P2P framework is extremely difficult to mock; however, these tests are a must in order to improve reliability of the library.

Client Connection Timeout

I will be implementing a timeout on a client connection with a host device that the developer will be able to specify. This is to prevent idle connection requests that may potentially degrade user experience if the application must block for the Wi-Fi P2P framework to establish the connection. More than likely, once this time out occurs, this library will cancel said connection, alleviating the user from a potential lengthy connection request.

Customizable Streams

I plan to—in the future—implement the Strategy design pattern in order to allow developers whom use this library to implement their own functionality for process such as the OutputStream and InputStream for transmitting and receiving data. Forcing the developers to only have the option to use the ObjectInputStream and an ObjectOutputStream with serializable objects is clearly a design flaw; therefore, I will implement a way for developers using this library to provide their own logic if desired.

The goal eventually is, if a developer using my library implements their own functionality for, they may submit a pull request for me to add to my library for other developers to benefit.

Customizable Constructors

In addition, I would like to make use of the Builder pattern by allowing a developer whom uses the library to define their own constant values. This builder would likely be packed full of options for the developer using the library to choose from. For example, the pool size for the ExecutorService in the ObjectReceiver. Ideally the builder would look something like the following:

```
Host host = new HostBuilder(getApplication(),
"UNIQUE_SERVICE_TAG",
"UNIQUE_INSTANCE_TAG").setObjectReceiverThreadPoolSize(5).build(
);
```

Conclusion

I intend this library to be used by many other developers who wish to implement the Wi-Fi Direct functionality and potentially become a community to further progress the strength and reliability of the library through collaborative efforts.

Bibliography

Alliance, W.-F. (n.d.). *Wi-Fi Direct*. Retrieved from Wi-Fi Alliance: http://www.wi-fi.org/discover-wi-fi/wi-fi-direct

Android. (n.d.). *android.net.wifi.p2p*. Retrieved from Android Developers: https://developer.android.com/reference/android/net/wifi/p2p/package-summary.html

Android. (n.d.). *Wi-Fi Peer-to-Peer*. Retrieved from Android Developers: https://developer.android.com/guide/topics/connectivity/wifip2p.html

Android. (n.d.). *WifiP2pManager*. Retrieved from Android Developers: https://developer.android.com/reference/android/net/wifi/p2p/WifiP2pManager.html

Arasthel. (n.d.). *AsyncJobLibrary*. Retrieved from GitHub: https://github.com/Arasthel/AsyncJobLibrary

bluelinelabs. (n.d.). *LoganSquare*. Retrieved from GitHub: https://github.com/bluelinelabs/LoganSquare

markrjr. (n.d.). *Salut*. Retrieved from GitHub: https://github.com/markrjr/Salut