University of Amsterdam

MSc System and Network Engineering

Research Project 2

Adding An Unusual Transport To The Serval Project

Authors:
Alexandros Tsiridis
Joseph Hill

July 8, 2016

Supervised by:
Dr. Paul Gardner-Stephen

http://www.biglittlegeek.com/wp-content/uploads/2015/02/transfer-data-smartphones.jpg

Abstract

The Serval Project aims to provide a means of establishing a communication network when circumstances limit the availability of infrastructure. One way this is done is by using mobile devices to create a multi-hop wireless ad-hoc network. Mobile devices support many communication technologies that can be used to advance this goal. Serval already supports communication protocols such as ad-hoc Wi-Fi and Bluetooth on Android devices. However, these technologies have their disadvantages and more options are desired. Wi-Fi Peer-to-Peer is specification allowing wireless devices to form connections directly to each other without the need for other infrastructure. This research is not about creating Wi-Fi Peer-to-Peer connections. This research aims to exploit one of the protocols designed to support Wi-Fi Peer-to-Peer. Service discovery within Wi-Fi Peer-to-Peer differs from standard Wi-Fi service discovery in that it takes place before connections are formed. By leveraging this service discovery as a transport, data can be exchanged between any devices that are in range of each other without forming a connection. This allows for the establishing of true mesh typologies. Implementation issues and limitations imposed by the Android API limit its performance below what can currently be used to support Serval Mesh Datagram Protocol traffic. However, this research shows that this transport can be used to create low bandwidth multi-hop wireless ad-hoc networks and has the potential to be useful as an auxiliary communication channel, including as a transport for the delay-tolerant Serval Rhizome bundle protocol.

Contents

Terminology

Frame A wireless frame between Wi-Fi Peer-to-Peer devices.

Packet A sequence of bytes received from or delivered to the Serval Mesh App.

Service A string consisting of the service description concatenated with additional meta data such as, UUID, device, type or instance.

Service description The part of a service that describes its function.

Service info A collection of locally available services.

Service request A possibly empty string that is intended to be used to match against a list of services.

Service request frame A wireless frame containing one or more service requests.

Service response A possibly empty collection of services containing all the local services that match a set of service requests.

Service response frame A wireless frame that contains the service response from a device. It is sent in response to a service request frame.

1. Introduction

The goal of the Serval Project is to provide networking capabilities to mobile devices independently of telecommunications infrastructure. To accomplish this a secure mesh network is established between mobile devices. This allows for voice calls, text messaging and file sharing even when the standard communications infrastructure is unavailable due to failures such as those caused by natural disasters [4].

The described functionality is accomplished on the Android platform with the Serval Mesh app. Originally, wireless ad-hoc networks were created using the Wi-Fi adapter in IBSS mode (also known as ad-hoc). However, on the Android platform this requires system privileges ("root access"), and potentially recompiling of the kernel [5]. While it is often possible with sufficient effort to enable these privileges, the process is not trivial and not something a typical user can be expected to endure.

Wi-Fi Peer-to-Peer (also referred to as Wi-Fi Direct) is a specification from the Wi-Fi Alliance that allows direct communication between mobile devices and is implemented on the Android platform. However, Wi-Fi Peer-to-Peer is not a complete replacement for ad-hoc mode [6]. Wi-Fi Peer-to-Peer has limitations when it comes to supporting large or transient wireless ad-hoc networks. The Wi-Fi Peer-to-Peer specification includes service discovery which takes place before connections are established. This research has been conducted to investigate the possibility of using this service discovery as a data transport in support of a multi-hop wireless ad-hoc topology.

1.1 Research Questions

Our research project revolves around adding a data transport to the Serval Project's Android app, Serval Mesh. It currently supports five ways of setting up connection between devices: The first method is by connecting to the same Wi-Fi wireless network. The second method is by making your device a portable Wi-Fi hotspot that other users will connect to. These two options, however, do not provide a multi-hop topology. The third option is by connecting to an ad-hoc mesh, which requires root privileges and/or kernel recompilation. The fourth option is to use a Serval Mesh Extender device that acts simultaneously as a Wi-Fi access point, ad-hoc mode Wi-Fi node, and optionally also includes a UHF, VHF or HF packet radio. The final and most recent transport is by the use of Bluetooth names to form a multi-hop topology. This research project will aim to add Wi-Fi Direct as another option to connect and transfer data. This would allow for an alternative that provides the range of ad-hoc Wi-Fi mode without requiring root privileges, or any additional hardware. This leads to our main research question:

> How can Wi-Fi Direct be used to support the Serval Mesh network on the Android platform?

Since the ideal topology of a multi-hop ad-hoc wireless network is not actually what Wi-Fi Direct was designed to support. Our first sub question will address this.

> Can the discovery protocols supported by Wi-Fi Direct be used to transfer data in a multi-hop wireless ad-hoc topology?

If we are able to use the discovery service channels in Wi-Fi Direct to transport data, then the second sub-question will examine practicality and performance.

> Which types of Serval traffic can be supported and what performance can be obtained using this technique?

This may involve using different methods to support the different forms of communication provided by the Serval.

2. Literature and Technology survey

2.1 Wi-Fi Direct

Wi-Fi Direct is a certification for devices that implement the Wi-Fi Peer-to-Peer (Wi-Fi P2P) specification. Wi-Fi Peer-to-Peer is a technology that allows two or more devices to communicate directly with out the need of an access point. This is done by establishing groups of devices [7]. When two devices want to setup a Wi-Fi Peer-to-Peer connection, one of the devices takes the role of a group owner and acts as an access point, while the rest of the devices act as clients and connect to the group owner. All communication between the devices in the group is routed through the group owner. The following list shows the components of the P2P architecture:

1. P2P Device
   - Supports both P2P Group Owner and P2P Client roles.
   - Negotiates P2P Group Owner or P2P Client role.
   - Supports WSC and P2P Discovery mechanism.
   - May support WLAN and P2P concurrent operation
2. P2P Group Owner role
   - "AP-like" entity that provides BSS functionality and services for associated Clients (P2P Clients or Legacy Clients).
   - Provides WSC Internal Registrar functionality.
   - May provide communication between associated Clients.
   - May provide access to a simultaneous WLAN connection for its associated Clients.
3. P2P Client role
   - Implements non-AP STA functionality.
   - Provides WSC Enrollee functionality.

2.1.1 Device Discovery

Device discovery is the procedure that takes place in order to find the devices that are in range and have Wi-Fi P2P enabled. This procedure uses three channels, 1, 6 and 11, in the 2.4 GHz spectrum. The procedure is split into two states, the listen state and the search state. These states are complimentary meaning that one device must be in the listen state and the other in the search state, otherwise they will not be able to communicate.

The search state is active on all three channels. This means that it sends a broadcast probe request on all three channels in order to discover devices. Devices in this state will not answer probe requests, they will only accept probe responses which mean that a device has been discovered.

The listening state is performed on only one channel. The device chooses a random channel to listen on and that channel will not change until the P2P discovery is completed. The device is in that state for a random amount of time given by the equation $N \times 100TU$ where N is a random integer. The specification requires that a device be in this state at least 500ms every 5 second [7]. Figure 2.1 shows how the two states work in order for devices to discover each other. In addition, figure 2.1 shows the discovery process from only one device, the same process has to be performed by the other device as well.

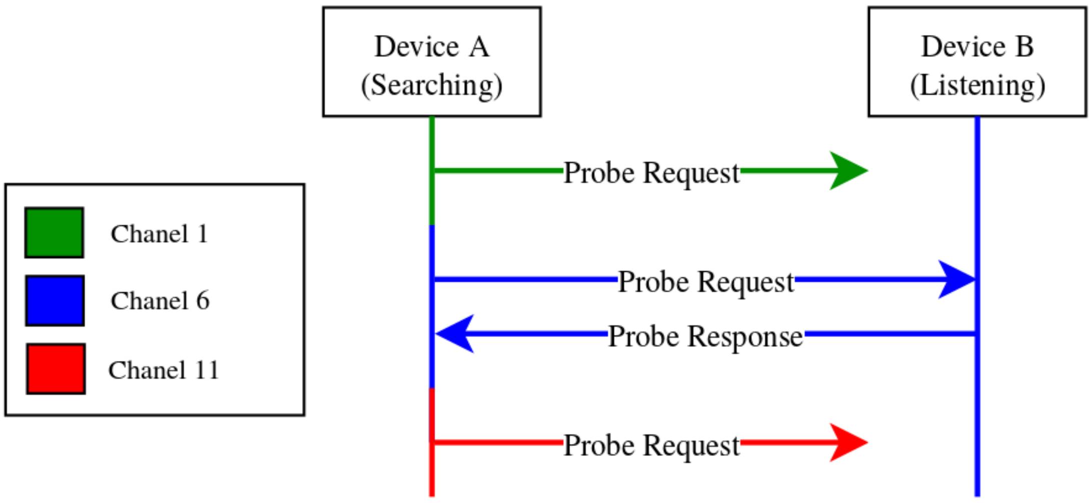

Figure 2.1: Device discovery process

2.1.2 Service Discovery

This functionality of the Wi-Fi P2P technology allows devices to check what services an already discovered (not connected) device offers [7]. This way a device can determine the compatibility of a discovered device as well as connect to the one that provides the service it is looking for. This is done by exchanging frames between the discovered devices prior to group formation. Due to the protocol being flexible and extensible it allows higher layer service advertisement protocols. For example the service discovery of Wi-Fi P2P is enabled at Android APIs 16 and higher implementing both Bonjour and UPnP [8].

The procedure uses the Generic Advertisement Service (GAS) protocol/frame exchange. The procedure initiates by having the requesting device sending a GAS Initial Request frame. The receiving device that supports service discovery answers back with a GAS Initial Response frame.

This process depends on device discovery as it is performed as an additional step in device discovery process. It is also performed during the search state of the device discovery. After the device that wants to perform service discovery receives the probe response during device discovery it will send a service request. The peer will respond back with a service response back to the sender. This service response encodes the local services of the device that sends it. The response can be up to 64K Bytes of data or empty if there are no data.

The response can also be fragmented if needed by using a slightly different method. The device A performing service discovery will initiate an initial service request. The receiving device B will answer back with a null response asking to use fragmentation. The device A will then answer back with a GAS comeback request. Device B will then respond with a comeback response sending the first fragment. The device A will again send a comeback request and device B will answer with a comeback response having the next fragment and the process is repeated until there are no more fragments.

The process of service discovery is not continuously running and not performed automatically with device discovery. The process needs to be initiated by the user or a program. Figure 2.2 shows how service discovery is combined with device discovery and what messages are exchanged when a device initiates it. The bold text shows the messages exchanged for service discovery, the rest are the device discovery messages.

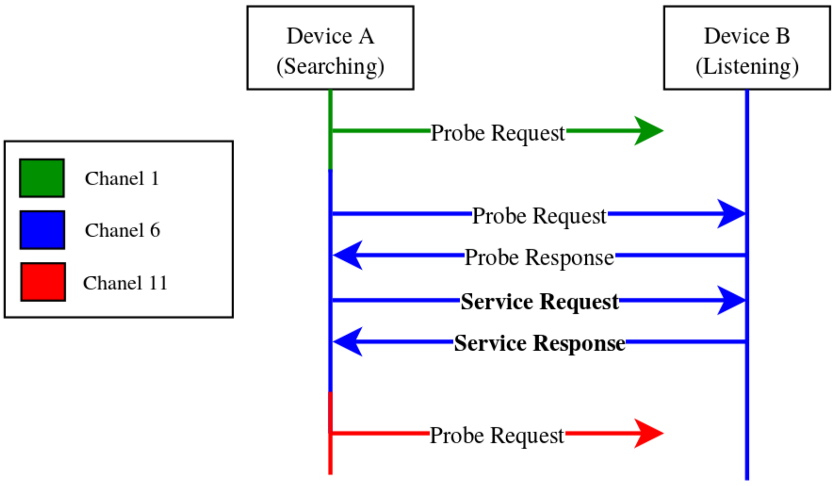

Figure 2.2: Service discovery process

2.2 Related Work

2.2.1 Automatic Android-based Wireless Mesh Networks [1]

Wong et al aim to allow Android devices to automatically establish and participate in a wireless mesh network topology using Wi-Fi Direct. In order to establish this goal they tackled two problems, the lack of ad-hoc networking in the Wi-Fi API and the elimination of user interaction at the Wi-Fi Direct WPS authentication.

In order to eliminate the user interaction they used a method that revolved around groups. By creating a group using one device it takes the role of the group owner and acts as an Access Point. Using this method users could simply connect to them as normal Wi-Fi clients and this process can be automated removing the need of user interaction.

In the Wi-Fi Peer-to-Peer specification it says that a device has the possibility to act both as a group owner (AP) as well as a client to other groups but it does not specify how. Unfortunately in the current implementation this is not the case. A group owner is only connected to its own group and the clients are only allowed to connect to one Wi-Fi Direct group. To eliminate this problem they used a device that is called a relay. This device connects to one of the groups and collects data that are to be sent to the other groups. It then disconnects from that group and connects to another in order to relay the data it collected.

This project managed to eliminate the problem of user interaction using groups as well as create a mesh like topology through the use of relays. The problem of this method is that the relay takes significant time in order to disconnect from one Wi-Fi Direct group and connect to an other. In addition the topology created is not an actual mesh topology as the relay disconnects from one network to connect to another network although the behavior is similar.

2.2.2 Serval Bluetooth transport [2]

One of the transport options of the Serval application is Bluetooth. They created an interface using Bluetooth with which they can perform both broadcast and unicast messaging using an interesting methodology. They send broadcast messages by encoding the packet into the device name. When other devices perform Bluetooth device discovery they can read the name of the device and take the packet from it. They send unicast messages by forming a peering relationship.

2.2.3 Serval Low-Bandwidth Asynchronous Rhizome Delivery (LBARD) [3]

Serval also supports the use of low-bandwidth packet radio interfaces to transport Rhizome bundles. Rhizome [9] is a delay-tolerant bundle-based protocol that functions by replicating Rhizome bundles between one-hop neighbours, and thus functions as a flood-routing based protocol. LBARD is currently able to make effective use of low-speed radio transports. The current implementation of Rhizome supports the RFD900 [10] UHF packet radio using an effective data rate of approximately 200 bytes per second, although the protocol is designed to accommodate links as slow as several bytes per second, and with latencies as high as tens of seconds (private correspondance with the Serval Project).

2.3 Conclusion

Looking at the Wi-Fi P2P specifications it is a promising technology that can be used to achieve the goal of this project. The “Automatic Android-based Wireless Mesh Networks” shows that Wi-Fi P2P can be used to create a mesh topology without the need of having other devices than phones and without the need of having root access or recompiling the kernel on those phones. In addition they managed to do the connectivity without the need of user interaction.

On the other hand the solution that they provide needs a great coordination as the components have different roles and in order to achieve their multi-hop network they need to assign those roles correctly. In addition having a device act as a relay which will disconnect from one group and connect to the other in order to transfer the data you loose the meaning of an multi-hop wireless ad-hoc network as the two networks are not actually connected. In addition operations such as choosing another relay, in case one has malfunctioned, or selecting another AP are expensive and introduces undesirable complexity.

We saw value in the discovery service functionality of the Wi-Fi P2P protocol. Looking into this functionality it allows us to transfer data between devices using a polling mechanism. In addition as the devices do not need to form groups it allows them to be part of an actual multi-hop wireless ad-hoc network due to the fact that they can communicate with anyone on their range directly. Moreover this process can be performed without the need of user interaction and can be automated. This project will aim to use Wi-Fi P2P service discovery as a transport mechanism.

3. Methodology

In this section we are going to discuss how we will implement our transport using Wi-Fi P2P service discovery, what features does service discovery have that can be used for transferring data and what are the advantages that it has which can be beneficial to the Serval project.

3.1 Service discovery as a transport

This procedure is designed to be used for letting other devices know what services your device provides and let them decide if they can benefit by connecting to you or an other device. Those data are encoded into a services and are polled by other devices. This means that we can use the service discovery as a poll based transport. A device will poll the other devices for data that they might have for it.

In addition according to the specification we can transfer up to 64K bytes of data which has the potential result for substantial throughput. Moreover, being poll based allows the receiver to read and ask for its data at its own pace. Further investigation and results of those aspects are described in later chapters of this report.

3.2 Why service discovery?

Service discovery was chosen to be the base of our transport for a number of reasons that makes it a good candidate for the Serval Project. First of all, it has functionality that allows sending requests to any of the devices in range. Also the ability to communicate without being exclusively paired with a device gives the opportunity to create a multi-hop wireless ad-hoc network. In contrast with the normal Wi-Fi P2P connections it does not create a star topology but it can communicate directly with every device that is in range.

Wi-Fi service discovery does not need any user interaction. This is unlike Wi-Fi P2P connections which, as implemented in Android, require a user to accept the connection. In addition, the Android API does not have a way to work around the message authentication step. To avoid this message, you have to implement groups that other clients will connect to as normal Wi-Fi devices, not Wi-Fi Direct clients. This adds complexity to the implementation without need and creates the problem of how to connect each independent star topology. This is due to the current implementation of Wi-Fi P2P on

Android not allowing concurrent connections to multiple groups. Using service discovery that problem is avoided as it can transfer data without user interaction and also has a multi-hop ad-hoc topology without adding complexity to our implementation.

3.3 How to use it

As mentioned previously, Wi-Fi direct discovery is a poll based protocol. The sender encodes data using the receiver identifier, and the receiver will ask all devices around for data that they have for him using his identifier.

Due to the nature of service discovery, it can be used for both reliable and unreliable data transfer, although it best fits a reliably acknowledged transport. The data are stored in the memory of the sending device until they are requested. In order to create an unreliable data transfer, a Time to Live (TTL) can be used to expire the data after some time, or to delete them after new data are to be sent to that recipient. The problem with this is that making the TTL too small risks the chance of never sending the data, and the same applies if the data volume is too high, where congestion based packet loss could prevent any effective throughput.

On the other hand a reliable method is easier to implement and fits the nature of the process. This can be implemented by imitating the behavior of TCP using ACKs and SEQUENCE numbers. Each service will have a sequence number and an acknowledgment number. Sequence number will help the receiver sort his data on a correct order. Acknowledgments of received data will allow the sender to delete the current service and create a new service with the next data.

3.4 Device Identifier

Each device has its own MAC address that should be unique across the world and that might be a good unique identifier for our devices. However, we where informed by our supervisor as well as the Serval Project contributors and programmers that this is not always the case. There are cases that they have personally experienced with cheap smart phones that had the same MAC address and we where advise not to use it (private correspondence).

This means that identifiers have to be implemented in such a way that they are unique across the network. In order to build true unique identifiers a database is needed. Using that database identifiers will never be the same and can be incremented when a device joins the network. For this project this is not an ideal solution as devices have to connect naturally without this central control.

The next option is to use random numbers form IDs in a space big enough to make the probability of a device having the same ID nearly impossible, taking account of the Birthday Paradox [11]. In order to do that an 8 byte length identifier was used. To check if this is a good choice the probability of having a duplicate identifier after 2^{20} devices was calculated. The formula $p = 1 - e^{-\frac{n^2}{2 \cdot 2^x}}$ where n is the number of devices and x is the number of bits was used.

$$p = 1 - e^{-\frac{2^{40}}{2 \cdot 2^{64}}} = 2.98023219e - 8$$

Given that it is implausible for anywhere near 2^{20} devices to be within Wi-Fi range of one another simultaneously, and the very low probability of collision, we conclude that an identifier composed of 8 random bytes is a safe choice for this project.

3.5 Conclusion

This chapter described how service discovery can be used according to its technical documentation. Looking at the features that it has along with the nature of this procedure we are certain that it can be used to transport our data between devices. The actual implementations of Wi-Fi P2P, such as the Android implementation, do not strictly follow the technical documentation and this creates other limitation that will need to be taken care of. The following chapters describe what limitations where faced using the Android API and how did that affect the actual implementation and performance.

4. Data Gathering

4.1 Equipment and Monitoring

Before an implementation of a data transport over Wi-Fi Peer-to-Peer can be developed, the behavior of the Android implementation must be examined. The four devices used for this research are listed in table 4.1. These devices range from the oldest API level that supports Wi-Fi Peer-to-Peer service discovery, 16, to the latest API level available, 23.

Table 4.1: Devices used for testing.

Device	Android Version	API Level
LG Nexus 5	6.0.1	23
Asus Zen Fone 2	5.0	21
Samsung GT-P5110	4.2.2	17
HTC Desire 500	4.1.2	16

Wi-Fi Direct devices may choose any of the three social channels as their listen channel. When device discovery is done each of these channels is checked in rapid succession. Shortly after a device is discovered, service discovery messages are exchanged on the channel the device was discovered on. This means a device doing service discovery may exchange messages with peers on three separate channels in a short period of time. In order to monitor the behavior of device and service discovery, as implemented within Android, wireless monitoring needed to be performed on multiple channels simultaneously. To accomplish this, a single workstation was connected to three wireless adapters. Each adapter was configured to capture traffic on one of the social channels as defined by the Wi-Fi Peer-to-Peer specification. This allowed for the wireless frames related to device and service discovery to be captured no matter which of the social channels the participating devices were operating on. Of the four devices we tested we found that the two running 4.x versions of Android always listened on channel one. The other two devices changed which listen channel they used when Wi-Fi was toggled. As long as Wi-Fi remained on, the listen channel did not change even when peer discovery was stopped and restarted.

4.2 Device and Service Discovery

In order for peers to be discovered, device discovery must be running on both devices. The application can start device discovery but the Android OS will stop performing device discovery after two minutes. The application is notified of this state change and may chose to start device discovery again.

The application also initiates service discovery. Device discovery must be running in order for service discovery to start. Service discovery runs for a much shorter and variable period of time, approximately a few seconds. This may be related to the number of peers discovered. The application receives service responses as they arrive but is not notified when service discovery is complete. The API does not provide any means of determining the current state of service discovery. When a device has no matching services it responds with a zero length response. This empty response is not sent to the application. Therefore, a lack of a response can not be interpreted as an error.

When two devices attempt service discovery at the same time they both fail to discover each other. From this, it is assumed that the starting of service discovery puts the device into the searching state. Device discovery requires a device to be in a listening state in order to be discovered. Having peers perform service discovery at the same interval risked devices becoming synchronized. Some form of randomization is needed to avoid this such as implemented in device discovery by the Wi-Fi Peer-to-Peer specification [7].

4.3 The Service

To create a service that can be discovered by other devices, a format must be selected. The Android OS supports two formats, DNS Service Discovery (DNS-SD) also known as Bonjour, or Universal Plug and Play (UPnP) service discovery which is based on Simple Service Discovery Protocol (SSDP) [12, 8]. No matter what format is used, Generic Advertisement Service (GAS) and the Access Network Query Protocol (ANQP) are used to transmit the messages. The choice of UPnP or DNS-SD only affects the formatting of the service description strings. However, this formatting ultimately does affect the amount of data that can be in a service response due to the different limitations of the formats.

The DNS-SD format requires a type and an instance string in addition to an object with key-value pairs [13]. In this object both keys and values are strings. On the Android 4.x devices tested the combined length of all parts was limited to 106 characters. When trying to register the service with the OS a failure would occur if the combined length was longer than this. On all of the devices tested an exception would occur if the combined length of any key-value pair was over 252 characters, resulting in the application crashing.

The UPnP format requires a Universally Unique Identifier (UUID) as a string, a device string, and a list of service descriptions as strings [14] On the Android 4.x devices tested, the combined maximum length of any one of the service strings and the UUID is 220 characters. If the combination of the UUID and any one of the service description is over 220 characters, attempting to register the service information with the OS will fail. The other devices tested did not have a practical limit. A service string of over 100,000 characters could be added. This is above the 64 K of data that the Wi-Fi P2P specification allows to be sent as a service response.

The UUID is checked for validity by splitting it on the hyphen characters and checking that there are five parts. Each part is then parsed as a hexadecimal positive long (four byte) value. Leading zeros are not required. Therefore each part of the UUID could be a hexadecimal string from '0' to '7fffffffffffffff', with or without leading zeros. This could be useful for extending the UUID to encode additional data, or for shortening the UUID to save space for more characters in the service description.

4.4 Service Request

In order to obtain the services that other devices have, a device has to initiate a service request. This service request contains a query that will be used to match against the services registered on the peer. The query is fulfilled using a sub-string match, any service description containing the query will be returned. The query can be an empty string that will match all services. Multiple service requests can be sent in a single service request frame. However, the Android 4.x devices tested produced an error when the combined length of the queries registered with the OS was over 64 characters. When attempting to add the service request that would result in exceeding the limit, a failure would occur. Additional attempts to add new service requests would appear to work. However, service discovery would fail to start until all service requests were cleared.

The Android OS assigns a service transaction ID to each new service request. This value is incremented by one each time a new service request is created. Testing showed that there is a bug in the way Android implements this transaction ID. Per the specification the transaction ID field is one octet [7]. After the service request that is assigned the transaction ID 127, the following new requests all result in malformed frames. This is because where the single octet value 128 is expected, a four octet value possibly representing -128 is inserted into the frame. Table 4.2 shows an an example of the observed progression of transaction ID values. This bug could be caused by a signed data type being used where an unsigned one was expected.

Table 4.2: Description of malformed service request packet

Length (2 Octets)	Protocol (1 Octet)	Transaction ID (1 Octet)	Query (Length - 2 Octets)
0x02 0x00 (2)	0x00 (0)	0x7e (126)	-
0x02 0x00 (2)	0x00 (0)	0x7f (127)	-
0x02 0x00 (2)	0x00 (0)	**0xff 0xff 0xff 0x80 (-128?)**	-

Unfortunately the Android API does not provide any means of detecting these malformed frames. Nor is there a way to find out what the current service transaction ID is. In addition, the ID is incremented globally. Meaning that other applications that use service discovery can cause it to increase. Therefore, the application can not make any assumptions about the current value of the ID or when it might reach this threshold.

The Transaction ID is only reset back to zero when Wi-Fi is to switched off and back on. This would of course disrupt other services using wireless connectivity. Determining when this should be done is also problematic due to the lack of information provided by the OS.

4.5 The Service Response

The Wi-Fi Peer-to-Peer specification states that a response can be up to 64 kilobytes [7]. The limit on the size of a response within a single frame was observed to be 1400 bytes. The GAS protocol provides fragmentation in order to transmit responses above this size. Monitoring the exchange of service requests and responses shows that the GAS protocol has not been fully implemented on all the devices tested. When the response can not be fulfilled within a single frame a fragmentation process is initiated by sending an empty initial response frame with a fragmentation flag set. The device that sent the request is then suppose to resend the request in a comeback request frame. However, two of the devices tested would not send the comeback request, the LG Nexus 5 and the HTC Desire 500. When a response is above 1400 bytes these devices will not be able to receive them.

There is another issue which limits the size of service response frames even further. If the response is over a certain size it will not be provided to the application even though it appears to be received by the OS. The size observed was typically about 1000 bytes. This limit was not consistent and varied at times by as much as 10 bytes even on the same device. As the framework does not provide any notification to the application on either of these situations, the application has to ensure ahead of time that responses will not exceed this limit through query construction and limiting the amount of data in a service response.

Moreover, there is no function in the Android API to let the application know that another device has sent a service request frame. This means that there is no indication that data has been sent to a peer and another method of coordinating the update of available data must be implemented.

4.6 Byte and Character Encoding

The Wi-Fi Peer-to-Peer API provided by the Android OS requires that the service descriptions be entered as a string data type. A challenge of passing binary data by this means is how to encode the binary data as a string. When the data is converted to a string it will be interpreted as characters and some characters may receive special treatment. The safest way to encode binary data as a string is typically by encoding it as a hexadecimal string. A hexadecimal string uses a relatively safe set of alphanumeric characters to encode the values. However, this provides a relatively low bit density as each character represents only 4 bits of the original data. Another method is to use

Base64 encoding, this provides 6 bits of information per character but does include two non-alphanumeric characters.

The one printable character found to receive special treatment in service descriptions was the comma character. When a comma was used in a service description the service description was split on the receiving device. When there were two consecutive commas, the sub-strings returned included an empty string. An empty string was also returned when a comma was the last character of the service description. Table 4.3 shows examples of the received strings when a comma is used. Since the number of splits is always equal to the number of commas, with some additional logic the receiving peer can reconstruct the original string.

Table 4.3: Results of using a comma in a service description.

Original String	Received Strings
"And,roid"	"And" + "roid"
"Android,,roid"	"And" + "" + "roid"
"Android,"	"Android" + ""

The delete character also received special treatment, it was converted to a null character. This can not be rectified on the receiver alone as it would not be able to distinguish between null characters that were originally null versus null characters that were originally delete characters. The sender would need to implement some way of marking these characters in order to compensate for this.

There are other considerations when attempting to use characters beyond the standard US-ASCII character set. The Android devices tested all used UTF-8 as the default character set. This is determined by the OS and is not configurable by the application [15]. UTF-8 is a variable length encoding scheme [16]. When characters are converted to bytes to be transmitted each character can be represented by up to four bytes. The characters corresponding to the standard US-ASCII character set are all represented as a single byte [16]. When using characters outside the standard US-ASCII character set those characters will be converted to multiple byte values when transmitted. Since there is a limit on the response size that the receiver will accept this must be avoided. Table 4.4 shows examples of how different characters will be expanded when using UTF-8 string.

Table 4.4: Example of variable length UTF-8 encoding [16].

Character	Encoded Bytes
$	0x24
¢	0xc2 0xa2
€	0xe2 0x82 0xac

Lastly, attempting to interpret arbitrary binary data as a UTF-8 string presents another challenge. Not all byte sequences are valid in UTF-8 and the behavior when trying to decode an invalid sequence is undefined [17]. This could lead to errors or data corruption.

5. Implementation

Given the limitations imposed by the Wi-Fi Peer-to-Peer specification and the API provided by the Android OS, data transport over Wi-Fi Peer-to-Peer Service Discovery was implemented using a post/pull concept. Data will be provided by encoding it into a service description and registering the service with the Android OS (posting). Creating this service description does not cause any data to be sent, it just makes it available to be queried by peers. When a peer sends the appropriate service request it will result in the data being sent in a response (pulling). The Android OS fulfills service requests automatically without notifying the application that a request has been made or fulfilled. Due to this it was decided to implement a means of acknowledging the data received by a peer. This lead to the decision to implement a reliable delivery model. Data up to the maximum allowed is provided in a service description of the service(post). Only when the destination acknowledges that it has received the data will it be removed. Several concepts from Transmission Control Protocol (TCP) were used within this implementation. With this implementation the two factors most important to performance are the maximum post size and how often data can be pulled.

5.1 Serval Peer Discovery

In order for Serval peers to distinguish each other from other Wi-Fi Direct devices the device name is set to a common format. The device name consists of a predetermined prefix of no more than six characters known by all Serval devices. The prefix is followed by the 8 byte identifier of that device encoded as a hexadecimal string. When a device is discovered the name is checked for the correct formatting. If the device is a new Serval peer, state tracking will be initialized and it will be added to the list of known peers. If a known Serval peer is not discovered again for a configurable amount of time, all state information will be deleted and it will be removed from the peer list.

5.2 Interaction with Serval

Between the transport and Serval, complete packets are exchanged. Therefor, unlike TCP, message boundaries must be maintained. Multiple packets can be inserted into a single post to increase bandwidth but to ensure that a frame can grow to its maximum size a way of inserting partial packets was implemented. Packets are fragmented by placing them in a send buffer with their length encoded as the first two bytes. When the service (post) is generated, bytes are copied from the send buffer up to the maximum response size. On the receiving end, the received bytes are copied into a receive buffer that checks the first two bytes for the length of the packet. If the entire packet is in the receive buffer the packet is delivered to Serval.

Serval requests that a broadcast packet be sent by providing a null address. When this occurs the transport copies the packet into the send buffer of all known peers.

5.3 Service String Format

Of the two service description formats supported by Android, UPnP allowed larger amounts of data to be inserted into a single service description. The service responses are returned to the application as a collection of services containing a UUID and the service description. The UUID is required by the API but what UUID is used can be specified. In order to allow more data to be encoded into the service description the acknowledgment, sequence and sort order numbers are encoded into the UUID along with the destination identifier, figure 5.1. This is followed by the service prefix and the Base64 encoded data. The sub-string starting with the destination identifier and ending with the service prefix is constant for data destined for a specific peer. This is the sub-string that will be used as a service query string to retrieve data.

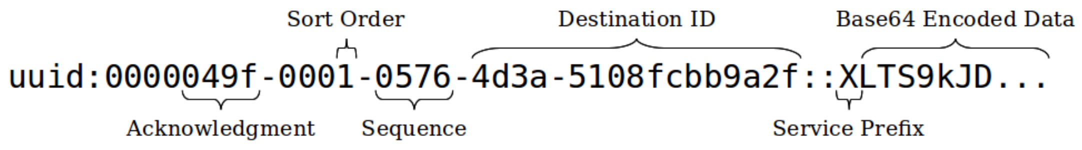

Figure 5.1: Format of service string.

5.4 Sequence Numbers and Acknowledgments

When a service is generated a sequence and acknowledgment number are encoded within it. The acknowledgment number is used to notify the sender that those bytes less than the acknowledgment number have been received and can be removed for the send buffer. This also allows for new data, if available, to be added to the response. The sequence number lets the receiver know where the first data byte in a response fits into the sequence. In this way the destination can tell if they have received a response with duplicate data. This could occur when a peer pulls data before the sender has received the latest acknowledgment.

5.5 Description Length Limit

In order to keep the response size under the limit of 1000 bytes, the service description length must be limited to 932 characters. Due to the Base64 encoding of the binary data this results in 699 bytes of data in a single service description. On the devices tested running Android version 4.x, there is a limit to the length of a single service description string. Due to this, in order to achieve the maximum service response size the data must be split into multiple service descriptions. This creates additional overhead which results in a limit of 764 characters spread over five service descriptions. This results in a maximum of 573 bytes of encoded data on these devices.

$$\text{Character Limit} \times \frac{6 \text{ bits}}{1 \text{ char}} \times \frac{1 \text{ byte}}{8 \text{ bits}} = \text{Byte Limit}$$

The splitting of data over multiple service descriptions leads to the requirement for a way to ensure that the receiver concatenates the pieces of the Base64 encoded data back together in the correct order. As the split occurs after the data has been encoded into a Base64 string the byte boundaries and character boundaries are not necessarily the same, figure 5.2. Meaning that it is not appropriate to use the sequence number to try to determine where a split occurred in a sequence of characters representing base64 data. A separate order number is encoded into the service of each split. This number allows the destination device to sort the services in the correct order before joining them.

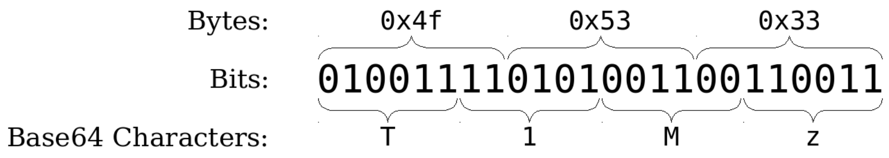

Figure 5.2: Comparison of byte and Base64 character boundaries.

5.6 Request Response Matching

To ensure that a destination device receives only the messages intended for it, it will send a service request with a query string encoding its own identifier. When the data source creates the service the sender will encode the destination device's identifier into it. This will cause the service request to only match the service created for that device. However, when using UPnP service discovery, a number of default service are created. Without additional steps the default services generated by UPnP service discovery would still be returned. As there is a limit on the total size of the service discovery response, this reduces the amount of data that can be provided in each response. In order to ensure that the service request issued will not match any of the default service, a specific character known by all peers is added to the service descriptions. By adding this character to the service request's query string, the default services are not returned.

5.7 Service Request Interval

Once the service request has been registered with the Android OS, data is pulled by asking the Android OS to start service discovery. As two devices performing service discovery at the same time will cause it to fail for both devices, this needs to be avoided. There needs to be some idle between starting service discovery to ensure that peer devices have an opportunity to pull data. Also devices must avoid becoming synchronized with a peer where they are performing device discovery at the same time every time despite the interval. This implementation avoids these issues by scheduling each service discovery interval at a randomly chosen time from a configurable range. Adding randomization to the process should ensure that peers do not become perpetually locked in synchronization. Using an appropriately high interval allows other peers the opportunity to pull data.

6. Testing and Results

Various tests where performed in order to produce data and analyze them. Using those data the throughput can be determined depending on different variables, latency, as well as jitter. By analyzing those data the performance of this implementation can be measured as well as how useful it can be to Serval project and what improvements it might need.

6.1 Throughput

The following data where gathered by having all the other variables fixed and only change the interval that service discovery was called. This test was performed using only the service discovery interval as it is the major variable that changes the throughput. If the process is called to often then the devices will not have enough time to communicate, if it is called after a long period of times then the throughput is lowered. In addition in our implementation we try to fill the frame to the MTU by implementing byte streaming which will utilize the bandwidth as it reduces the overhead.

The test was performed using four intervals for service discovery:

1. 5s-10s
2. 10s-15s
3. 15s-18s
4. 18s-20s

From the throughput graphs it can be seen where the service discovery interval creates a peak in the five second average followed by periods where no data is transferred. When there is a consistent distance between peaks, service discovery is occurring regularly. When there are irregular gaps between peeks, service discovery is failing. The 60 second average shows how consistently successful or failing service discovery impacts the overall average throughput.

The figure 6.1 shows the throughput when the service request interval is set to 5 to 10 seconds. The average throughput is 31.2 bytes/second. From the graph it can be seen that when service discovery is repeatedly successful this interval has a relatively high throughput. At several points the 60 second average reaches as high as 58.3 bytes/second. Unfortunately due to the instability there are also periods where service discovery repeatedly fails. This causes a 60 second average of 0 bytes/second at one point and several other points where throughput is below 20 bytes/second.

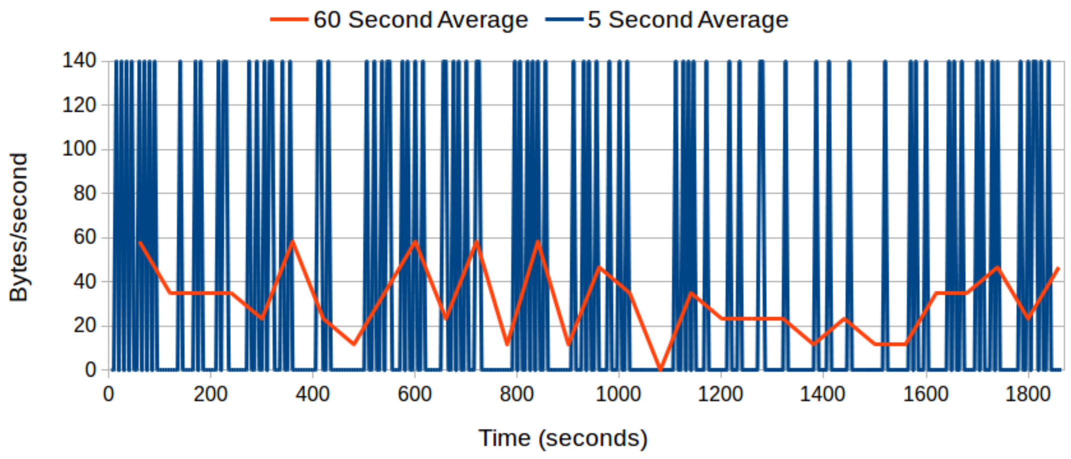

Figure 6.1: Throughput of the interval 5s-10s

The second test shows the results when an interval of 10 to 15 seconds is used, figure 6.2. While there are still periods of consecutive service discovery failures, this interval is more stable than the previous one. The 60 second average has the same maximum of 58.3 bytes/second but only reaches it once. However, it only drops below 20 bytes/second on a couple of occasions and never drops to 0 bytes/second. As a result it achieved a better average throughput of 34.1 bytes/second, which is the highest of any of the tested intervals.

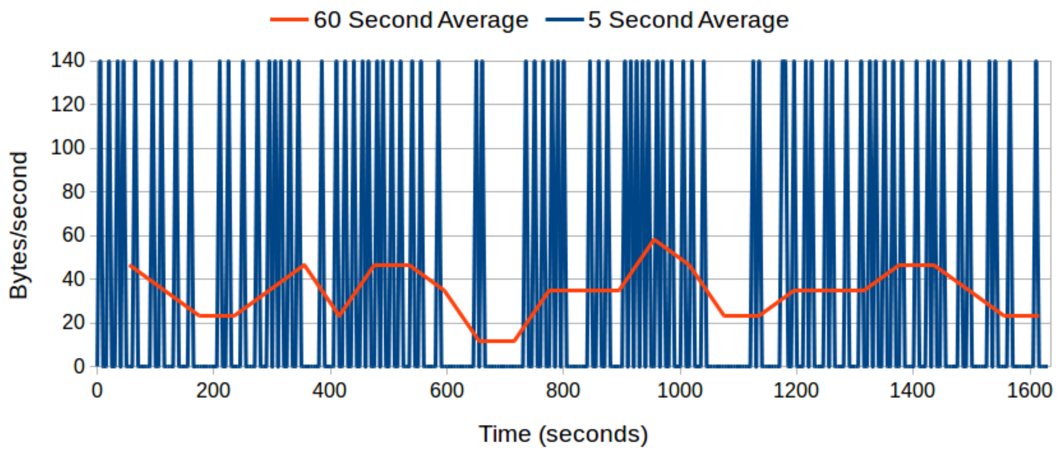

Figure 6.2: Throughput of the interval 10s-15s

The third test was performed using the interval 15 to 18 seconds. Figure 6.3 shows that the 60 second average is more consistent but at a noticeably lower rate. The 60 second average is only above 40 bytes/second at two points. However, there are a number of points under 20 bytes/second including one at 0 bytes/second. Having a longer average time between service request, it is expected that it would have a lower potential for throughput. However, the instability was not expected. This could be due to the narrower range used with the difference between the maximum and minimum interval being only 3, compared to 5 with the previous tests. The overall average was 29.1 bytes/second.

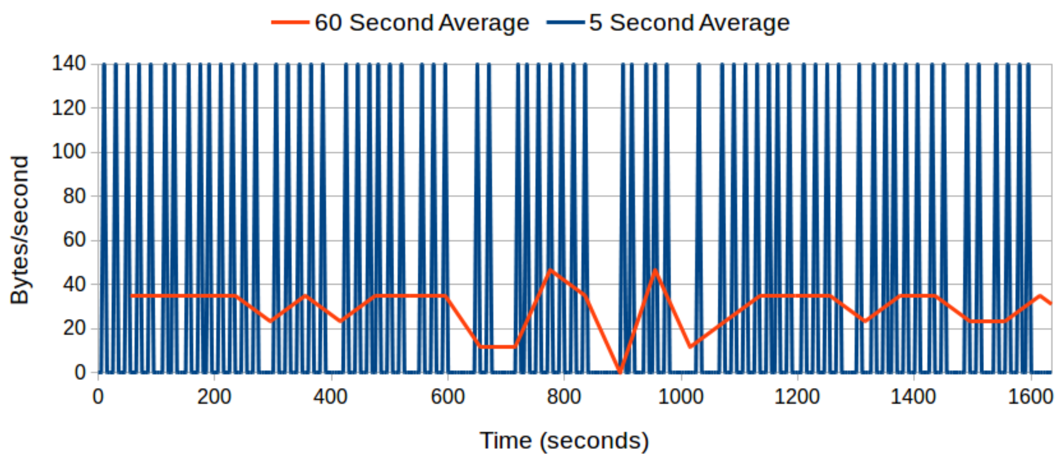

Figure 6.3: Throughput of the interval 15s-18s

Four the fourth test an interval of 18 to 20 seconds was used. This test is similar to the previous one in that it uses a relatively long average interval with a narrow range. Figure 6.4 shows that the results are also similar. The peaks of the 60 second average are constrained to just below 40 bytes/second. There are a few periods where service discovery was particularly unreliable. This test also drops to 0 bytes/second at one point. The service discovery failures in this test also suggests that the range between minimum and maximum needs to be greater to increase stability. The overall average throughput was 26.5 bytes/second, the lowest of any of the tests.

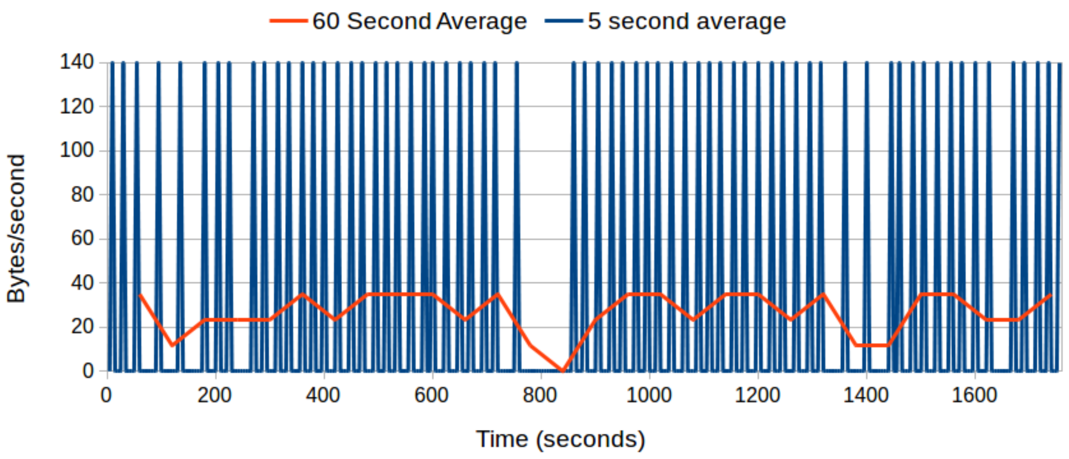

Figure 6.4: Throughput of the interval 18s-20s

6.2 Latency

The most stable service discovery interval, 18 to 20 seconds, was used in order to measure the latency of packets being transferred. Figure 6.5 shows a histogram of the latency in seconds for all packets transferred. The measurement was done by timing how long it took each packet to be received after entering the frame. This test was implemented by placing a timestamp in each packet (8 bytes) when added to the buffer and checking the timestamp against the clock of the receiver. The device clocks where synchronized by updating them from the same local provider. The data rate was set to ensure that no packets would be queued waiting to enter the frame. During this test no packets were fragmented. In total 1343 packets where transferred of which 85.63% had a latency of under 20 seconds while 14.37% had a latency of 21 to 43 seconds.

From the histogram it can be seen that even when service discovery is reliable, a packet's latency could fall anywhere within the time of a single interval. In addition we can see that there is a limited number of packets that are transferred on the second interval due to a miss in the service discovery. We can also conclude that due to the low number of data missing the first discovery service it would be unreasonable to increase our interval to longer than 18 to 20 seconds as it would also decrease the throughput.

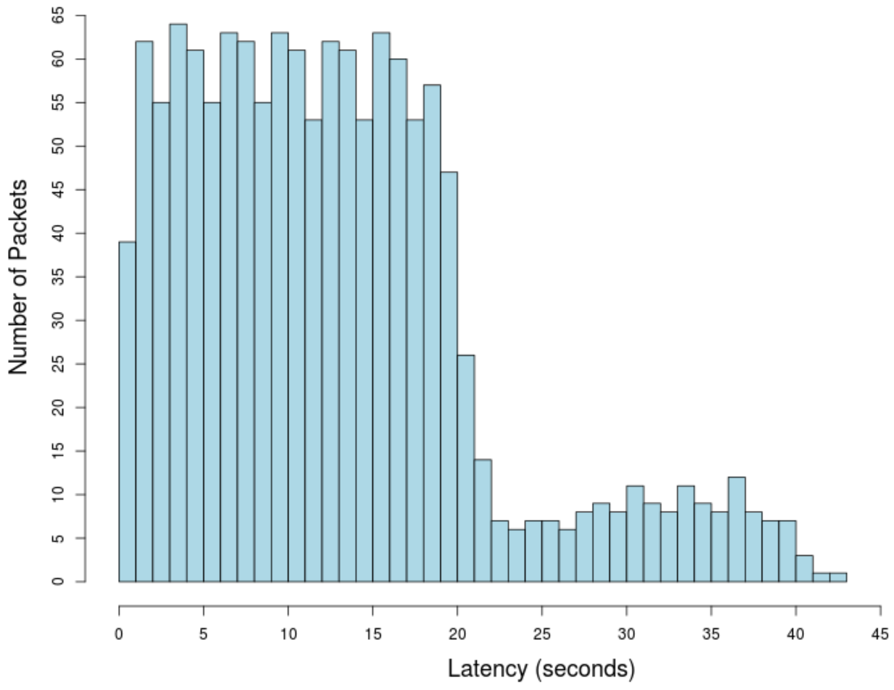

Figure 6.5: Histogram of Latency on the interval 18s-20s

6.3 Unreliable Transport

An unreliable transport could be implemented by posting data for a limited amount of time and then removing it without waiting for an acknowledgment. The time the data is posted for could be thought of as the time to live (TTL). Given the latency data collected for the previous experiment, by calculating the percentage of packets under a given TTL, an estimate of packet loss can be made. The results are presented in figure 6.6.

Using those results, an estimate of throughput can be calculated based on TTL and the percent of packet loss, $Throughput = \frac{Framesize \times Loss}{TTL}$. A frame size of 699 bytes is used for this calculation. The results of this calculation can be seen in figure 6.7.

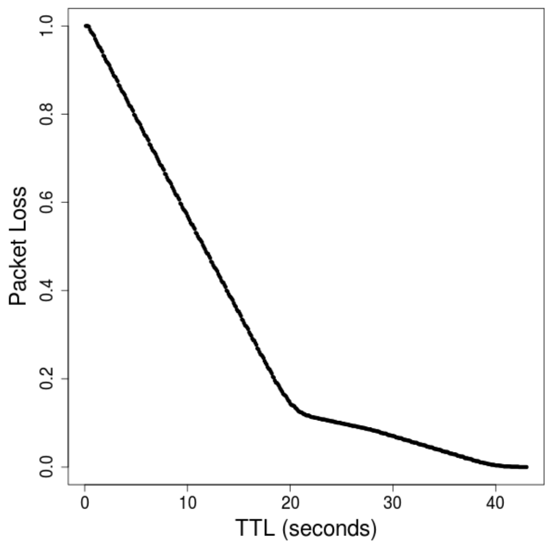

Figure 6.6: Unreliable transfer

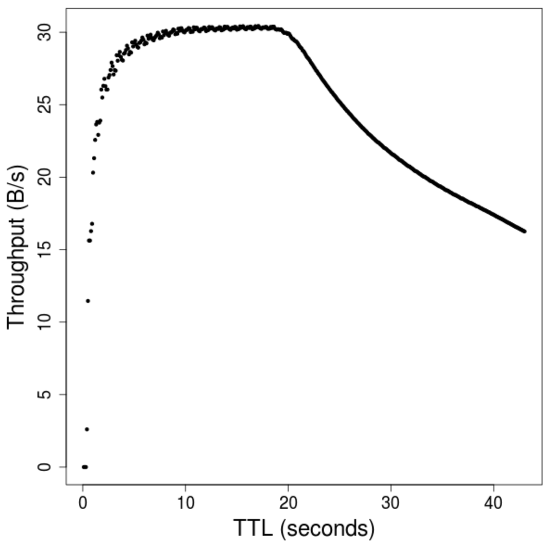

Figure 6.7: Unreliable transfer throughput

The graph shows that throughput it capped at about 30 bytes/second for TTL values between 5 and 20 seconds. The best throughput, 30.4 bytes/second is projected to be at a TTL of 17.1 seconds which would result in a packet loss of 25.6%. While this is higher than the reliable transport received with the same interval settings, depending on the use case, the high packet loss might not be acceptable. However, it could also be the case that the use case requires enforcing a maximum latency in which case the unreliable transport could provide this while providing greater throughput.

7. Conclusion

7.1 Integration with Serval

The interface that was created using Wi-Fi P2P service discovery was also integrated into the Serval application. It was tested using various configurations in order to see what services it can support. Serval peers where formed using this interface and data where sent and received as the logs where indicating. Unfortunately due to the low throughput that the created interface was able to provide messaging and calls using the Serval application could not be performed. By looking at the data using the logs we could determine that the Serval application was exchanging far more data than the interface can handle as it also tries to manage encryption and mesh routing.

The channel that was created might be useful to the Serval project for coordination and configuration, e.g., as a side-channel to provide hints as to when it may be advantageous to establish Wi-Fi P2P connections, and to pass the necessary information to do so. It may also make sense to use the Wi-Fi direct discovery transport that we have implemented together with LBARD to provide a low-bandwidth Rhizome transport for Serval, which would allow MeshMS and other services, despite the limitations of the interface. Given that LBARD has already been demonstrated with network links that are only slightly faster, but that can suffer much higher rates of packet loss, it would seem that the prospects are good for running LBARD over this new Wi-Fi direct discovery transport.

7.2 Throughput

The throughput of the implemented interface is currently low to be used on Serval app as explained above but this can change in the future depending on the changes that might be performed at the Android API. The main reason that throughput is so low is the bug that was found in the transaction ID. By fixing that then the interface will be able to place multiple requests with different sequence numbers allowing a device to retrieve multiple frames from a single device rather than just one as the current implementation does.

Another factor that limits throughput is the frame size. According to specification service responses where suppose to be able to transfer up to 64K of bytes but Android does not allow that. By increasing the size of the data that can be transferred using the GAS messages and by implementing fragmentation the interface's throughput can be increased dramatically with just a simple change in our code.

7.3 Service Discovery

As was already discussed service discovery to be successful needs the devices to be in complimentary state. Experiments where performed by changing the interval of calling the service discovery as well as randomizing the variables to make it more stable. Unfortunately as we can see in the testing and results section 6 there where calls of the discovery services that where not successful and increasing the interval would also result in a lower throughput.

A better solution would be to synchronize the devices and have them in complimentary states in order to make the service request successful in shorter intervals which would also increase the throughput. The functionality to perform such synchronization is not implemented in the API and it also does not provide the necessary functions to implement it. On the other hand such synchronization should be implemented in the framework and Android OS rather than the application level.

8. Future Work

During this project it was observed that different Android APIs had different behavior regarding Wi-Fi Direct and most of the variations had to do with service discovery. This leads us to the conclusion that Google is still working on improving the functionality of the Wi-Fi Direct and more changes are probably going to be seen in further implementations. In order to take advantage of those changes in future APIs a research will need to be contacted in order to see what has changed and how to utilize it.

In addition as discussed the currently implemented interface is not suited for actual communication transport for the Serval project. On the other hand data can be transferred without user interaction, at the distance of the Wi-Fi range, in a multi-hop wireless ad-hoc topology without the need of connecting to a network. Those are some aspects that are highly beneficial to Serval application and the transport that was create through this research can be used for other actions. A research can be contacted in order to find the home place of this interface.

An other piece of the puzzle that we did not try to find in this project and can be used for a new research is to find a method to connect the disjoint star topologies that Wi-Fi P2P creates. By finding the appropriate solution to this problem then the Serval application will be able to create reliable mesh networks with great range and the throughput that Wi-Fi has.

Acknowledgements

Our special thanks go to Dr Paul Gardner-Stephen for providing us the opportunity to work on adding unusual transport to the Serval project and act us our supervisor. We would also like to thank the Serval project developers and contributors for providing and helping us with their code and especially Jeremy Lakeman for providing us with in-depth information of how the code works and its functionality.

A. Project Code

Our interface was integrated with Serval's Android app. We cloned the repository of serval batphone and added our code there. The code can be found at `https://github.com/niemand2015/batphone`. In addition to modifying a few classes to integrate our code the following classes where added:

- WifiP2pControl.java (src/org/servalproject/system/wifidirect/WifiP2pControl.java)
- WifiP2pPeer.java (src/org/servalproject/system/wifidirect/WifiP2pPeer.java)

Bibliography

[1] Paul Wong, Vijay Varikota, Duong Nguyen, and Ahmed Abukmail. Automatic android-based wireless mesh networks. *Informatica*, 38(4):313, 2014.

[2] Serval Project. Bluetooth. `https://github.com/servalproject/batphone/tree/development/src/org/servalproject/system/bluetooth`. Online; Accessed on 12-06-2016.

[3] Serval Project. Low-bandwidth asynchronous rhizome transport (lbard). `https://github.com/servalproject/lbard`. Online; Accessed on 08-07-2016.

[4] Serval Project. The serval project. `http://www.servalproject.org/`. Online; Accessed on 30-05-2016.

[5] Serval Project. Serval mesh readme. `https://github.com/servalproject/batphone`. Online; Accessed on 30-05-2016.

[6] Thinktube Inc. *Why Wi-Fi Direct can not replace Ad-hoc mode*, 2016.

[7] Wi-Fi Alliance. Wi-Fi Peer-to-Peer (P2P) Technical Specification, Version 1.5. *Wi-Fi Alliance Technical Committee P2P Task Group*, 2014.

[8] Android Open Source project. WifiP2pServiceInfo - Android Developers. `https://developer.android.com/reference/android/net/wifi/p2p/nsd/WifiP2pServiceInfo.html`. Online; Accessed on 06-07-2016.

[9] Paul Gardner-Stephen, Romana Challans, Jeremy Lakeman, Andrew Bettison, Dione Gardner-Stephen, and Matthew Lloyd. The serval mesh: A platform for resilient communications in disaster & crisis. In *Global Humanitarian Technology Conference (GHTC), 2013 IEEE*, pages 162–166. IEEE, 2013.

[10] RFDesign. RFD900 UHF Packet Radio. `http://rfdesign.com.au/products/rfd900-modem/`. Online; Accessed on 08-07-2016.

[11] W. O. J. Moser Morton Abramson. More birthday surprises. *The American Mathematical Monthly*, 77(8):856–858, 1970.

[12] Wikipedia. Simple Service Discovery Protocol — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Simple_Service_Discovery_Protocol, 2016. Online; Accessed on 6-07-2016.

[13] Android Open Source project. WifiP2pDnsSdServiceInfo - Android Developers. https://developer.android.com/reference/android/net/wifi/p2p/nsd/WifiP2pDnsSdServiceInfo.html. Online; Accessed on 06-07-2016.

[14] Android Open Source project. WifiP2pUpnpServiceInfo - Android Developers. https://developer.android.com/reference/android/net/wifi/p2p/nsd/WifiP2pUpnpServiceInfo.html. Online; Accessed on 06-07-2016.

[15] Android Open Source project. Charset - Android Developers. https://developer.android.com/reference/java/nio/charset/Charset.html. Online; Accessed on 03-07-2016.

[16] Wikipedia. UTF-8 — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=UTF-8&oldid=728169679, 2016. Online; Accessed on 3-07-2016.

[17] Android Open Source project. String - Android Developers. https://developer.android.com/reference/java/lang/String.html. Online; Accessed on 03-07-2016.