Wi-Fi Direct P2P Messenger-Browser App

Anand Sekar
anand272@cs.washington.edu

Michael Flanders
mkf727@cs.washington.edu

Source Code

Can be found in this GitLab repository.

Abstract

Modern internet and cell networks offer little privacy and anonymity. These infrastructures also cannot always be relied upon. During natural disasters, protests, or other emergencies, or when cellular data and access points are not available, communication needs to rely on something else. Peer-to-peer (P2P) networks have emerged as a viable alternative. For our project, we addressed these privacy and availability issues by creating a P2P messenger and HTML server application for Android devices. Our application allows users to chat 1-on-1 with other users and offers limited web browsing through a dedicated peer that acts as a downlink from the Internet. Although our prototype only works on Android, our implementation uses Wi-Fi Direct and a JSON API for communication between clients, so our protocol can run on any device using Wi-Fi Direct.

1 Introduction

Much of today's communication is held up by and routed through the Internet backbone - an enormous, interconnected network provided by Internet Service Providers (ISPs) and monitored by several governmental and corporate entities. As a result, tracking an individual's interactions with the Internet is performed by entities like advertising agencies; simply talking about or searching for certain products or brands will result in corresponding ads being displayed throughout sites which use Google AdSense or similar services [4].

A host of cryptographic security paradigms have been implemented at each and every layer of internet communication [14] in order to preserve the confidentiality and integrity of important interactions - such as logging into a company's website to access sensitive information or into one's bank account. Anonymity and privacy, however, are extremely difficult to achieve, and are sometimes reasonably traded for accountability. Virtual private networks (VPN) and onion routing (such as Tor) have become popular methods for preserving anonymity, but have their vulnerabilities [9] [8].

For instance, suppose one is attempting to maintain anonymity by connecting to the internet via a VPN while using a browser app on their phone in private/ incognito mode. Their identity is potentially exposed when push notifications are initiated by apps, such as email and social media, which contain identifying information related to their accounts.

For another example, suppose Alice and Bob are attempting to communicate without anyone else knowing that they are. Even when using end-to-end encrypted communication, ISPs can see their traffic. Authorities can deploy devices called StingRays which act as a small cell tower to identify and even track phones within its range; it can allow them to read unencrypted metadata. The aforementioned example involving push notifications also apply here: devices connected to the internet exchange a complex wealth of information which can be used to establish one's identity, even if all one wants to do is communicate with one other person or view a web page [13] [17].

Ultimately, it's difficult to trust complex networks and services when all one wishes to do are simple, isolated actions. One approach to preserve privacy is using proven cryptographic key exchanges and encryption - which works; however the unnecessary complexity of the interaction is left untreated. Another approach - which can be done in tandem with cryptography - is simply reducing the complexity of the interaction, making the possible security vulnerabilities few and apparent. If all Alice wants to do is talk to Bob, then Alice should be able to do that without routing that entire process through the backbone. Direct communication is lacking. In this

project, we explore establishing a direct connection between two devices independent of existing Internet/cell networks.

2 Background

Before we describe the app and its architecture, this section summarizes some relevant background information on Wi-Fi Direct.

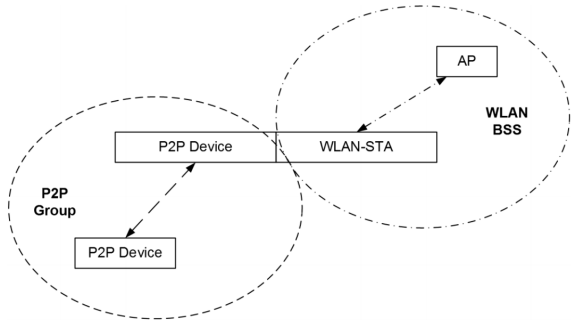

Figure 1: Diagram of a P2P concurrent device taken from the Wi-Fi Direct specification v1.8 [3]

2.1 Wi-Fi Direct

Wi-Fi Direct allows devices to set up a P2P wifi network without needing a wireless router or other access points. In a home network that can contain printers, smart televisions, gaming consoles, and Internet-of-Things (IoT) devices, such devices are usually predetermined to be a P2P Group Owner (GO) [3]. In our app, however, the group owner is just the application client that another client is trying to connect to. The GO devices are responsible for being an "AP-like" entity that provides basic service-set functionality and services for their connected clients [3]. The GO devices may also provide communication between their clients and access to a simultaneous WLAN connection like we do in our application and as shown in Figure 1.

We decided on using Wi-Fi Direct for the project because it was easy to get a prototype working and tested; we both have 2 Android devices to test with, and the Wi-Fi Direct development API on Android is pretty easy to use. Wi-Fi Direct also uses WPA2 which helps meet our need for privacy and security talked about in §3.

3 Goals

Our goal was to create an accessible, peer-to-peer communication network that does not does not rely on the Internet or cellular infrastructure. We made a checklist of features in our project proposal that our P2P network was supposed to implement.

Here is the list sorted descending by order of importance:

1. Enable at least two devices to communicate with each other independently from the Internet/cell networks
2. Provide some user interface
3. Use end-to-end encryption for communication
4. Be accessible through different devices (phones, laptops, etc. . .)
5. Enable multiple devices to communicate with each other simultaneously
6. Be long range
7. Store private and deniable data on the network via pings between devices
8. Access the internet anonymously through a centralized "peer" server

The first two features we made a necessity since a P2P client needs to be able to communicate with another peer and in an accessible way. Then we focused on features 3-5 as the second most important group of features, and we left features 6-8 as more exploratory ideas that would be fun to get around to if there was time.

We fully accomplished goals 1, 2, and 8 and partially accomplished goal 4. Since Wi-Fi Direct runs on many devices and we use a JSON API for messages, anyone can write a client for our app that will run on any device using Wi-Fi Direct. This was tested by forming a P2P connection with Microsoft's Wi-Fi Direct demo, without changing any code. We thought we had accomplished goal 3, "Use end-to-end encryption for communication," since Wi-Fi Direct uses WPA2, but we learned during the Q&A of the project presentation that this is not the same thing as end-to-end encryption. §6 contains a further discussion of this.

The remaining goals (5 and 7) were not implemented due to time and resource constraints. The parts of the client we finished during this quarter serves as a prototype and foundation for this future work.

4 Application Overview

We implemented the solution using an Android application (minimum API 26) and its P2P APIs [5] for Wi-Fi Direct.

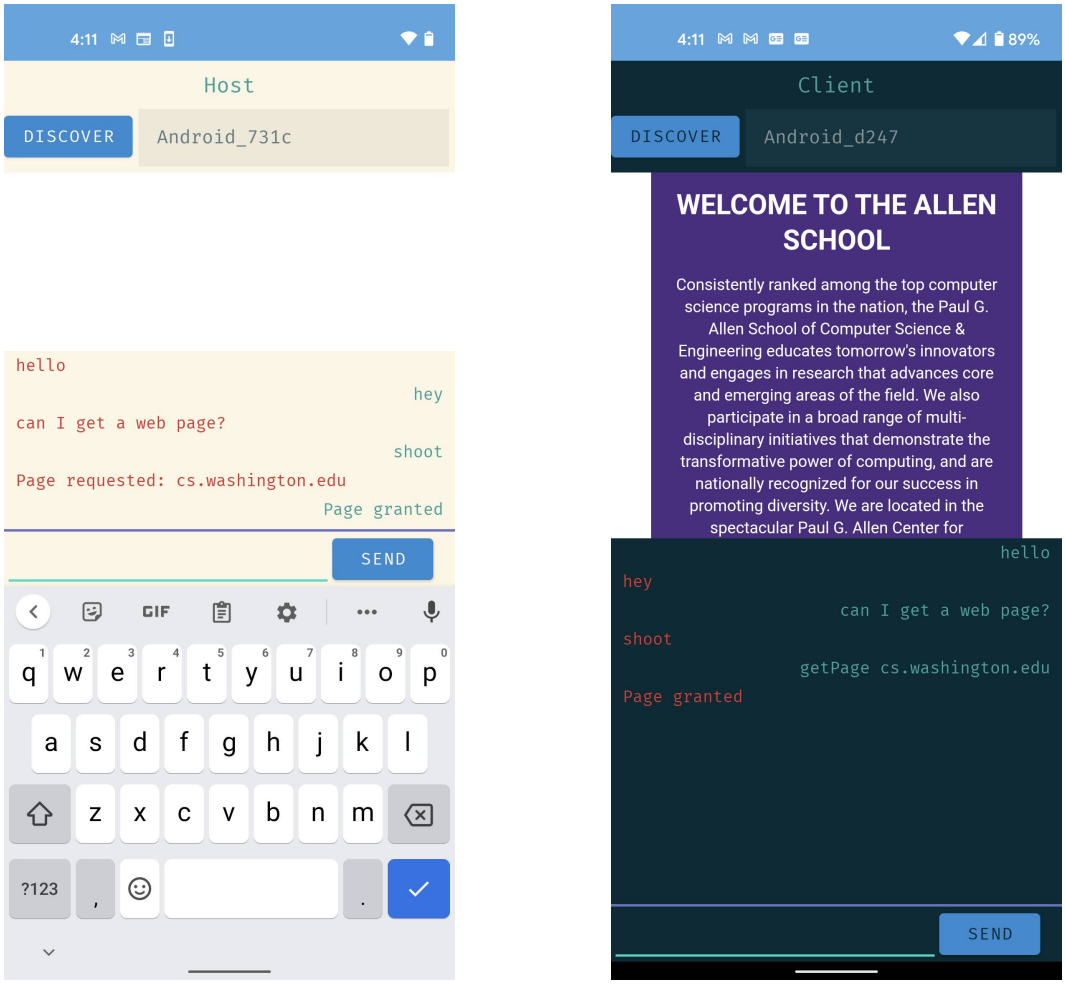

Figure 2: The interface is a single screen; both light and dark modes can be used.

The application lets users connect to each other, send messages to each other, and ask for webpages using the other's internet connection. Figure 2 shows what the app looks like on two phones communicating with each other.

The user begins by pressing the "Discover" button to advertise their device and populate the peers list on its right; the user then selects the desired other Android device on the list to connect to it. Once either "Host" or "Client" is displayed at the top, they can begin sending messages to each other. A message of the form "getPage url" where url is replaced with a simple url (such as of the form google.com or cs.washington.edu) will result in being sent that page from the other phone (e.g. if the client says "getPage cs.washington.edu", the host will download that page through their internet connection, then deliver it to the client through the Wi-Fi Direct connection. That page will then populate the client's WebView. When viewing a webpage, a user can click on hyperlinks to other urls, which will result in "getPage url-link" in their text field, which they can send to effectively "browse" internet pages. Throughout this browsing, the client's identity (IP/ MAC address, other internet activity) is never exposed to the rest of the internet - only to the connected server/ WiFi-Direct group. The server takes on that exposure instead.

5 Implementation and Architecture

Every component of the application communicates with the Main Activity, as shown in Figure 2. The WiFi P2P Manager object is provided by Android's Wi-Fi Direct API. It notifies our activity of the status of the Wi-Fi Direct connection(s) through intents - Android's version of software interrupts or signals which indicate that some action needs to be performed. Upon initializing the Activity, we instantiate our WiFiDirect Broadcast Receiver and delegate it to respond to specific intents which notify us of WiFi being enabled/ disabled and peer connections changing; we respond by notifying the user of the changes through the peers list.

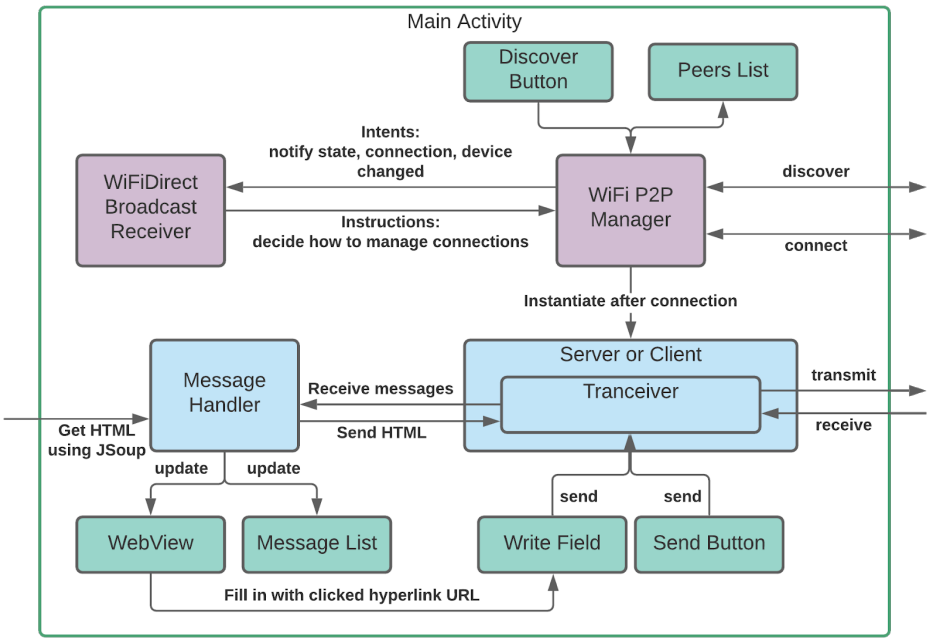

Figure 3: System diagram of how the application code is structured.

When the discover button is clicked, the activity calls the manager to advertise its own device while gathering information of nearby Wi-Fi Direct devices to populate the peers list. When a device on the peers list is clicked, the activity calls the manager to form the connection and declare one node in the group as the group owner; we then deem the owner the "host" and the others as "clients" and notify the user which one they are. Once this is done, the activity respectively instantiates the server and client threads with their addresses. The server opens a server socket, the client creates a socket, and they both form a TCP connection with each other; both classes pass the socket to a transceiver thread which handles all the communication (so currently, host and client devices communicate symmetrically).

Type	Data
Message	A chat message
Page	HTML of a web page to be read
Request	URL of web page to be sent
Chunk	A segment of a larger JSON string
Chunk_Prefix	The number of chunks to expect for an incoming large JSON string

Table 1: Structure of messages communicated.

All communication is encapsulated by strings encoded in a modified UTF-8 format; UTF-8 includes a 2-byte size prefix field to ensure we stream whole strings like packets. We utilize Kotlin's DataInputStream and DataOutputStream which utilize that prefix to handle sending and receiving those strings. These strings are encoded in JSON format, with two fields: Type and Data. The forms of messages sent are shown in Table 1.

Type	Response
Message	Add message to the message list and update the view
Page	Unescape the HTML and load it in the Web View
Request	Update the chat log of the interaction; Download and send the web page's HTML.
Chunk	Decrement the chunk counter and add to the string builder. Once the counter hits zero, extract the whole string and handle it as a page; clear the string builder.
Chunk_Prefix	Set a chunk counter to the number of chunks and ensure the string builder is clear.

Table 2: Response to message types.

We use a library called JSoup to download a web page's HTML to be sent. In order to send web pages, we must escape the characters in the HTML to be compatible with JSON. Additionally, we must deal with the length of the string. Since UTF-8's size prefix is 2 bytes, the maximum size of a string is 65535 (or $2^{16} - 1$) characters. Most webpages' HTML is larger than that, so

we must split up messages into multiple chunks. In the write thread of the transceiver, if the string is longer than 50,000 characters, we split the string into chunks, each which are less than 50,000 characters long. We chose this size since it's less than the maximum length, and would account for any few additional JSON characters. We first send over a chunk prefix, which contains the number of chunks to be sent, then send over each chunk. The read thread of the transceiver delivers the message to a handler in the main activity thread. This handler deals with the received messages according to their type, as shown in Table 2.

When a link on the WebView which contains a hyperlink is clicked, we override the default response by instead populating the message field with the url prefixed with "getPage."

6 Discussion

The app robustly sends messages and webpages. There is some noticeable latency with requesting a web page, but only on the order of magnitude of a few seconds - which most likely comes from downloading the webpage from the server's internet connection.

As it is, the app would enable two trusted parties to communicate with each other privately - within WiFi range (less than 100 m) - without using any internet or cell infrastructures. Through their communication, no external entities can read their messages or associated metadata; the client can privately browse the web through the server, who serves as an exposed access point. We have achieved our main goals (1-3) (see §3), and mostly completed goal 7. The extension of goal 7 is to have one access point that is exposed, but service other nodes who wish to be anonymous on the internet - as basically a proxy.

The reason we did not just run the server node as a hotspot and have a client node connect to it is for the aforementioned complexity - data such as push notifications will be sent back and forth from the user's phone which exposes them. Ideally, we'd find a way to route just the browser's HTTP tunnel through the server node, which would act as an HTTP proxy by acting as an HTTP server to the client node and an HTTP client to the desired web server, routing HTTP messages back and forth and thereby shadowing the IP address of the client node. However, there is no permissible way to do this in Android, i.e. without gaining overprivileged, root access to the phone.

We thought we had accomplished goal 3, "Use end-to-end encryption for communication," since Wi-Fi Direct uses WPA2, but we learned during the Q&A of the project presentation that this is not the same thing as end-to-end encryption. WPA2 will protect and encrypt the messages but only while in transit; the messages will still be in plaintext at higher levels of the network stack, so someone could still intercept the messages. We made a serious error in not fully understanding WPA2 and end-to-end encryption, but saying that we provided this. If we had more time on the project, getting end-to-end encryption figured out would be our first task. WPA3 and WPA2-AES are more secure protocols to begin with.

In order to fulfill goal 6 to have communication be long-range, we'll have to switch from WiFi Direct to a protocol like LoRAWAN, which would have a range of around 5 kilometers. There is hardware out there, such as the FiPy board [12], which provides an API to use TCP/IP sockets like the ones from this project.

Figure 4: A closeup of the FiPy board

There are a host of improvements to be made on the Android app itself. First and foremost, the app should be split up into several fragments. Right now, everything is on a single screen activity. To make it more user-friendly, there should be an initial screen to discover and connect to o a peer, a chat screen, and a browser screen. All these fragments should be connected using Android's Navigation Host API. Furthermore, the app would work best with a Model-View View-Model architecture to make the data on View objects more persistent and life-cycle aware.

Goal 7 was an interesting idea inspired by something the professor mentioned earlier in the quarter about storing data"in-flight," i.e. using delay line memory. There have been a few explorations into this idea but this space is relatively unexplored [11][6][16]. It wouldn't have worked with the Wi-Fi Direct example, since there need to be at least one of two properties:

1. There should be sufficient latency between nodes for delay line memory to take effect. This wouldn't work with the close-range, low-latency transmissions in Wi-Fi Direct, but could work well with LoRaWAN.

2. There should be multiple nodes in the network routing and juggling around fragments of data. This wouldn't work with only two nodes, but can work with several.

Figure 5: A set up of the FiPy board, along with a solar panel and antennae, which a friend used. They prototyped their protocol over WiFi, then transferred over to LoRa relatively smoothly.

It's certainly something to explore for the future.

Note: At the project checkpoint, we had implemented sending basic messages through the TCP connection, but had no proper or robust way to check for the beginning/ end of messages - we simply sent data through, and the transmission was fast enough/ the data was short enough to pull all of it from the buffer at once. Since the checkpoint, we implemented sending large web pages, which forced us to make messaging proper by creating a packetized protocol. We also changed around the UI as we added features.

7 Related Work

There are a lot of other apps and class projects that use Wi-Fi Direct to run a P2P chat app. There is a pretty simple tutorial on the Android development website that shows how to use the Wi-Fi Direct API and get connected to another device [5] which probably explains the popularity.

There are also a lot of peer-to-peer messaging apps for phones [2]. Some of the more popular apps include Silence and TwinMe [1, 15]. Silence has a downside of requiring a phone number and only sends messages. On the other hand, TwinMe allows message, video, photo, and music sharing as well as group chats. The main difference between our app and these apps is our ability to use the P2P connection as a network and browse the web through a central peer. There is also a lot of other work on doing networking over a P2P connection such as IPFS [7] and libp2p [10]. IPFS is a file system over P2P networks intended for the sharing of hypermedia [7] and is similar to what we wanted to achieve with serving HTTP over the P2P connection. libp2p is a bunch of specifications, libraries, and protocols that help developers write P2P networking applications [10].

8 Conclusion

References

[1] Silence app.

[2] Peer to peer app survey, 2008.

[3] Wi-fi direct specification, 2009.

[4] COMMISSION, F. T. Online tracking.

[5] DEVBYTES. Create p2p connections with wi-fi direct, 2019.

[6] EKMAN, E. pingfs - "true cloud storage".

[7] JUAN BENET, PROTOCOLLABS. IPFS - content addressed, versioned, p2p file system.

[8] LING, Z., LUO, J., YU, W., FU, X., XUAN, D., AND JIA, W. A new cell counter based attack against tor. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, Association for Computing Machinery, p. 578–589.

[9] PERTA, V. C., BARBERA, M. V., TYSON, G., HADDADI, H., AND MEI, A. A glance through the vpn looking glass: Ipv6 leakage and dns hijacking in commercial vpn clients. *Proceedings on Privacy Enhancing Technologies 2015*, 1 (01 Apr. 2015), 77 – 91.

[10] PROTOCOLLABS. libp2p.

[11] PURCZYNSKI, W., AND ZALEWSKI, M. Juggling with packets: floating data storage.

[12] PYCOM. Fipy board.

[13] STROBEL, D. Imsi catcher, 2007.

[14] SUO, H., WAN, J., ZOU, C., AND LIU, J. Security in the internet of things: A review. In *2012 International Conference on Computer Science and Electronics Engineering* (2012), vol. 3, pp. 648–651.

[15] TWINLIFE. Twinme app, 2012.

[16] WIKIPEDIA CONTRIBUTORS. Delay line memory — Wikipedia, the free encyclopedia, 2020. [Online; accessed 19-December-2020].

[17] WIKIPEDIA CONTRIBUTORS. Stingray phone tracker — Wikipedia, the free encyclopedia, 2020. [Online; accessed 19-December-2020].