Exploring MANETs to Provide Connected Transportation Services at Crosswalks

Jacob Rodal
jr6ff@virginia.edu
University of Virginia
Charlottesville, Virginia

ABSTRACT

Connected and autonomous vehicle technologies are expected to greatly increase the safety and efficiency of surface transportation. However, some challenges need to be addressed before the connected transportation systems become effective. Solutions that rely on infrastructure-dependent wireless will be a major impediment to the development of an effective system due to the need for significant resources and maintenance. Another challenge that must be addressed is figuring out how to support all forms of travelers, including pedestrians, cyclists, and vehicles. In this paper, we propose a MANET based solution that uses Google's nearby connections API to provide connected transportation services to all types of travelers at crosswalks. Based on our performance evaluation, the MANET application could potentially be useful in rural settings, though more extensive testing will need to be performed.

1 INTRODUCTION

Connected and autonomous vehicle technologies are expected to greatly increase the safety and efficiency of surface transportation. A challenge with adopting these systems is ensuring that all types of travelers, including pedestrians, cyclists, and vehicles, can communicate. Relying on the large-scale, dense deployment of wireless network infrastructure in order to provide connected transportation services is not necessarily an effective solution. Constructing and maintaining a network devoted to transportation will require significant resources, which are not readily available in today's transportation environment. This has led to increased interest in commercial wireless networks, particularly as they gain capability (for example 5G). However, there will certainly persist significant concerns regarding latency and availability when relying on commercial networks that support a wide range of applications to effectively and reliably support transportation safety applications. These challenges have led to our research group investigating the use of MANETs (Mobile ad Hoc Networks) to enable connected transportation services. MANETs are peer-to-peer, decentralized wireless networks that don't rely on a pre-existing infrastructure, such as routers in wired networks or access points in infrastructure-dependent wireless networks [14]. Each device in a MANET moves independently and can change its links to other devices as necessary. Each device forwards traffic unrelated to its own use, just like a router. Since there is no infrastructure, devices can create and join ad hoc networks anywhere and anytime.

This paper will specifically focus on using MANET technology to develop an Android mobile application that enables connected transportation services at crosswalks, but the mobile application we developed will also be useful for the development of other MANET applications. Our mobile application is developed using Google's Nearby Connections API, which enables nearby devices to exchange data in real-time, regardless of network connectivity. Connections are established automatically for the user, so all they have to do is open the application and wait. Once connections are established, they can press a button to send a message to every other device in the MANET. The other devices in the MANET then display the message in the center of their screen. Our paper will first discuss peer-to-peer Android APIs that could possibly be used to create MANETs, which will also include overviews of related work to bring MANETs to mobile phones. Next, the paper will go over the system design for our mobile application based on Google's nearby connection API, followed by a performance evaluation of the mobile application. The paper will end with a discussion on the limitations of the mobile application and potential improvements.

2 PEER-TO-PEER APIS

2.1 IP-level MANETs

An Internet Protocol (IP) design for a MANET application would be the preferred method of connecting devices, as this would offer the greatest performance. An issue with trying to create an IP-level MANET on Android devices is that, since 2013, the Android API does not enable the creation of an ad-hoc network. Android devices do not ship with any tools, even for the root user, that can be used to configure the wireless interface, so an IP-level Android MANET would be forced to use custom firmware and Android kernel builds [15]. AdhocDroid [20] is an Android mobile application that sets up an IP-level 802.11 ad-hoc network with multi-hop capability for root users by using their own API. In order to use the application and similar solutions, the user would have to "root" their phones, which is the process of enabling superuser features on the phone. Rooting Android phones is generally discouraged by service providers and phone manufacturers because of the potential security risks associated with enabling administrator privileges on phones. Some applications, such as Google pay, check if the phone has root privileges and will refuse to function due to the risks. Unless Google decides to officially support the creation of an ad-hoc network, an IP-level MANET isn't an appropriate choice for our crossroad application, as the average user wouldn't be able to use the application.

2.2 Bluetooth

Bluetooth is a wireless technology standard for exchanging data over short distances and building personal area networks (PANS). Bluetooth communicates using radio communications, so devices do not have to be in visual line of sight of each other. Bluetooth is widely adopted and exists in a devices such as mobile phones,

watches, and laptops. There are various different Bluetooth versions, with Bluetooth 4.0 and 5 being the latest major releases. Later versions of Bluetooth 5 offer enhancements such as options that connectionless services such as location-relevant navigation [19] while still remaining backward compatible with Bluetooth 4.0. There have been successful attempts to create MANETs using Bluetooth technology, such as Kargl et al. who create a MANET using Bluetooth-enabled cellphones for voice transmission and Reyes et al. who propose a MANET auto configuration system based on Bluetooth technology [12][16], indicating that Google's Bluetooth API could potentially be used to develop our MANET application.

A very brief, high-level overview of how Bluetooth works is as follows: Bluetooth connections are operated in a master-slave architecture. Client devices (the slaves), discover and connect to a host (the master) to create what is known as a piconet. A piconet can consist of up to seven slaves, though hardware typically supports fewer than seven. Data is transferred between the master and one slave at a time; masters typically switch between what slave it should address in a round-robin fashion. A slave can be part of multiple piconets, thereby creating a scatternet as depicted in Figure 1. A device that is the master in one piconet can simultaneously be a slave in another piconet, meaning that a device can be a master of seven slaves and a slave of multiple masters. The network created by Bluetooth connections enables multihop wireless traffic and can be used as an ad hoc network. For an in-depth, low-level explanation of how Bluetooth works, see [11].

2.3 Wi-Fi Direct

Wi-Fi Direct (also known as Wi-Fi Peer-To-Peer) is a Wi-Fi standard for establishing peer-to-peer wireless connections without the need for network infrastructure [1]. It is often used to connect mobile phones, cameras, printers, PC's gaming devices, and many other devices. An advantage of Wi-Fi direct is the ability to connect devices from different manufacturers, so long as the devices all implement the Wi-Fi direct standard. Connections can either be direct connections between two devices (1-1) or between multiple devices with a star topology (1-n). When establishing Wi-Fi Direct connections, one device, known as the group owner (GO), hosts a Wi-Fi Direct group that other devices, known as group clients (GC), can discover and connect to. Google's Wi-Fi Direct API examines each device's power management, UI, and service capabilities and uses this information that can handle server responsibilities most effectively [3].

A disadvantage of using Wi-Fi direct is that the entire network will collapse if the group owner leaves the network. Additionally, Wi-Fi direct only supports single-hop networking, meaning the range of the created network would be limited. Since vehicles and pedestrians will often be leaving the network, Wi-Fi direct would not be an appropriate technology stack to develop our MANET application with.

2.4 Google's Nearby Connections API

In their paper on Ad hoc networking on mobile devices, Jesse van den Ende et al. investigated using Google's nearby connections to share and simultaneously playback audio files among peers [21]. Their resulting application succeeds at constructing a network automatically as well as sharing files among connected devices, showing that the nearby connections API is another feasible choice for creating ad hoc connections.

Nearby Connections is a fully offline peer-to-peer networking API that allows applications to easily discover, connect to, and exchange data with nearby devices in real-time, regardless of network connectivity [8]. Connections are high-bandwidth, low-latency, and fully encrypted between devices. The API uses a combination of classic Bluetooth, Bluetooth low energy (BLE), and Wi-Fi hotspots to discover and advertise to nearby devices. Using a combination of different technologies allows the API to circumvent each of the technologies' weaknesses while leveraging their respective strengths. For example, Bluetooth has low connection latency but also has low bandwidth, whereas Wi-Fi hotspots have slightly higher connection

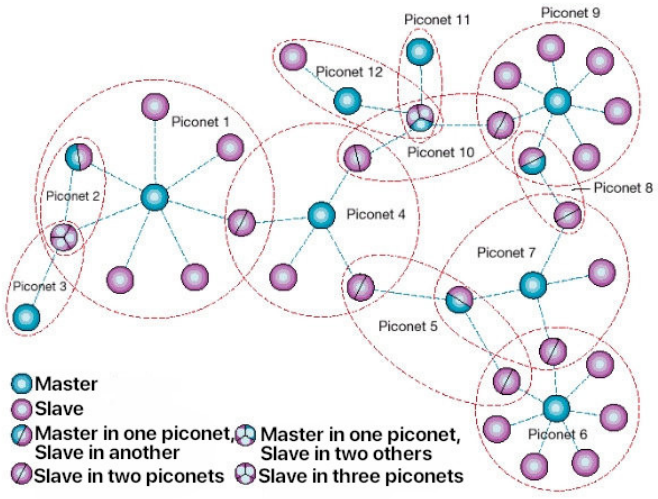

Figure 1: Twelve piconets interconnected into one scatternet [2]

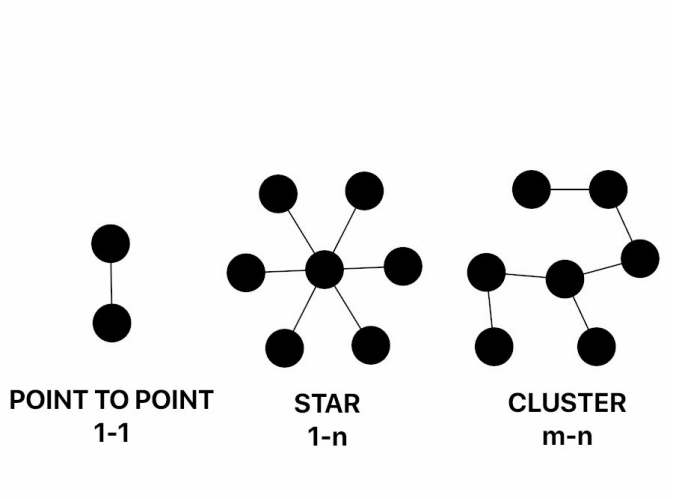

Figure 2: Nearby Connections Strategy Topologies

latency but also have much higher bandwidth. Nearby connections will connect devices via Bluetooth and start transferring data instantly. In the background, the API will start a Wi-Fi hotspot, and when it's ready, the API will seamlessly transfer the connection from Bluetooth to Wi-Fi with no work required by the application developer [22]. Once connections are established, devices communicate by sending each other payloads. The connections are full-duplex, which means that the devices can simultaneously send and receive payloads. The payloads are as follows:

- **Bytes**: byte arrays of up to 32K
- **Files**: files on the device's storage
- **Streams**: used for sending data on the fly, such as with video or audio

Meftah et al. [13] analyzed the apks of many applications in the Google Play store and found that the nearby connections API is being used in various popular applications, as seen in table 2. The adoption of the API in such popular applications helps provide credibility to the robustness of the API.

2.4.1 API overview. The application developer must declare the following permissions if they wish to use the nearby connections API:

- BLUETOOTH
- BLUETOOTH_ADMIN
- ACCESS_WIFI_STATE
- CHANGE_WIFI_STATE
- ACCESS_COARSE_LOCATION
- ACCESS_FINE_LOCATION (Only on devices running Android Q and onwards)

The nearby connections framework provides three different strategies for advertising and discovery: `P2P_CLUSTER`, `P2P_STAR`, and `P2P_POINT_TO_POINT` [9]. `P2P_CLUSTER` supports an M-N connection topology. It allows each device in the network to initiate outgoing connections to M other devices and accept incoming connections from N other devices. This is the most flexible strategy, but it results in lower bandwidth connections. `P2P_STAR` supports a 1-M, or star-shaped, connection topology. It allows devices to either accept incoming connections from N other devices or initiate a connection to a single device, but it can't do both at the same time. The topology is more limited than the cluster, but it enables higher bandwidth connections. `P2P_POINT_TO_POINT` supports a 1-1 connection topology. It allows a device to connect to exactly one other device, which, while limiting, allows for the highest possible throughput. The topologies of the strategies can be seen in Figure 2.

The sequence of events that take place for two devices to communicate with each other is as follows: an advertiser calls `startAdvertising` and a discoverer calls `startDiscovery`. The discoverer is alerted to the advertiser's presence by the `onEndpointFound` callback. The discover can then call `requestConnection` if it is interested in connecting. After a request is sent, both the discoverer and the advertiser get an `onConnectionInitated` callback that provides an authentication token that the devices can use to verify they are indeed talking to each other. Both sides can then either accept or reject the connection with `acceptConnection` or `rejectConnection`. When each side gets the other's response, the `onConnectionResult` callback is invoked. If the connection was sucessfully established, then either side can send payloads with the `sendPayload` method, which results in the other side

Table 2: applications in the Google Play Store using the nearby connections API [13]

Application	Downloads
WhatsApp Messenger	5,000,000,000+
WeChat	100,000,000+
BBM - Free Calls & Messages	100,000,000+
Hola Free VPN Proxy	50,000,000+
Email application for Mail.Ru	50,000,000+
AirDroid	10,000,000+
XShare	10,000,000+
iPair-Meet	5,000,000+
DataBot 1A	1,000,000+
FITAPP	1,000,000+

Table 1: Key methods of Google's Nearby Connections API [4] [5] [6] [7]

Class	Method	Description
ConnectionsClient	startAdvertising	Starts advertising an endpoint for a local app.
ConnectionsClient	startDiscovery	Starts discovery for remote endpoints with the specified service ID.
EndpointDiscoveryCallback	onEndpointFound	Called when a remote endpoint is discovered.
ConnectionsClient	requestConnection	Sends a request to connect to a remote endpoint.
ConnectionLifecycleCallback	onConnectionInitiated	A basic encrypted channel has been created between you and the endpoint.
ConnectionsClient	acceptConnection	Accepts a connection to a remote endpoint.
ConnectionsClient	rejectConnection	Rejects a connection to a remote endpoint.
ConnectionLifecycleCallback	onConnectionResult	Called after both sides have either accepted or rejected the connection.
ConnectionsClient	sendPayload	Sends a Payload to a remote endpoint.
PayloadCallback	onPayloadReceived	Called when a Payload is received from a remote endpoint.
PayloadCallback	onPayloadTransferUpdate	Called with progress information about an active Payload transfer
ConnectionsClient	disconnectFromEndpoint	Disconnects from a remote endpoint.
ConnectionLifecycleCallback	onDisconnected	Called when a remote endpoint is disconnected or has become unreachable.

getting the `onPayloadReceived` callback and both sides getting `onPayloadTransferUpdate` callbacks until the transfer is completed. Any device can disconnect can disconnect from any other device with the `disconnectFromEndpoint` method, which results in the connected device(s) getting the `onDisconnected` callback. This sequence can be seen in Figure 3, which was shown in the Google IO session "How to Enable Contextual application Experiences."

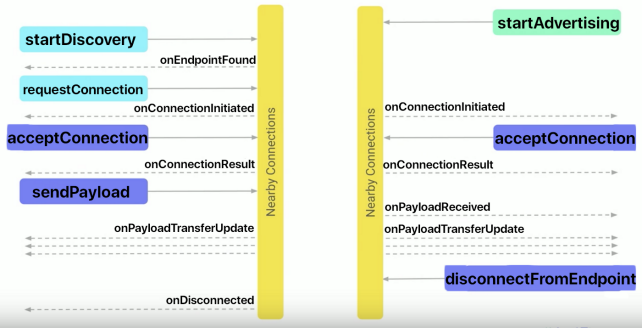

Figure 3: Nearby Connections API sequence of events [22]

3 SYSTEM DESIGN

We chose to develop our application using the nearby connections API, as the abstractions over different technology stacks make the API easy to use, compatible with various versions of Android, and performant. The API supports multi-hop relays and there is no notion of group leader, so devices can leave at any time without destroying the network. It abstracts over the Bluetooth API and uses better technologies where possible, making it better than using the Bluetooth API. The application is currently named "Connected Crossroad" and is hosted on Github [17]. The usage for the application is as follows: a pedestrian approaches a crosswalk and hits a button on the application that sends a signal to all vehicles in the MANET. The application tells the pedestrian whether or not their message was sent successfully.

3.1 MANET creation strategy

When designing the application, careful consideration went into developing a strategy for creating a large MANET. First, a nearby connection strategy had to be chosen. The `P2P_POINT_TO_POINT` strategy is not suited for MANETS, as it can only establish 1-1 connections. The `P2P_STAR` strategy isn't well suited for the task either because the 1-n topology is too limiting. It would require one phone to act as a host to all other phones, which would be difficult to coordinate ahead of time. Additionally, if the phone went out of range, the entire network would collapse, since all devices in the network were connected directly to the same host phone. Due to this limitation, the `P2P_CLUSTER` topology was chosen. While more flexible, the `P2P_CLUSTER` method has less bandwidth than the `P2P_STAR` method because it maintains connections with primarily Bluetooth. Since a Bluetooth piconet only supports up to seven slaves, a carefully constructed method for creating a larger MANET with the cluster topology is necessary. One way to do this is to create a network similar in structure to the scatternet shown in Figure 1, where each device in our network will be connected directly to up to seven devices but connected indirectly to many other devices. An additional restriction is that a device already in the network cannot connect to a device that is also already in the network. This prevents scenarios where "islands" form, which prevents new devices from connecting to the network. To track whether a device is already in the network, each device will maintain a list of all of the devices connected to it and exchange this information when establishing a connection. This allows each device in the network to know every other device in the network. A visual representation of information exchange can be seen in Figure 4. A node in the blue network sends a connection request to a node in the green network. Once the connection is established, the node is the green network sends its data to the node in the blue network and the node in the blue network sends its data to the node in the green network. This transfer of data is represented by the colored arrows. Each node in the blue network is forwarded the data from the green network and each node in the green network is forwarded the data from the blue network. The result is that every node knows about every other node, meaning the blue and green networks have effectively combined into a single network. Disconnection is handled in essentially the same manner as connections. Assume that the blue and green networks in Figure 4 disconnected. The blue node that was directly connected from the green node would know every device in the old network that is connected to the green node. The blue node would send this list of nodes that should be removed to all of the other blue nodes and then it would remove that list of nodes from its memory. The green network would do the same thing.

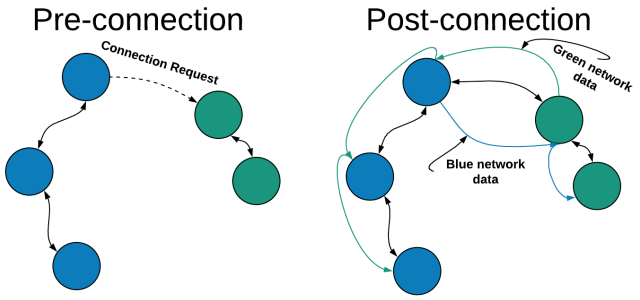

Figure 4: Merging two networks

3.2 Implementation Details

Note: To read the source code for this application, visit our Github repository [17]. In our implementation, each device in the MANET is limited to at most two direct connections instead of the more performant seven direct connection limit so that we could more easily evaluate multihop performance with fewer devices. This results in a chain of devices shaped like a linked list, where each node in the list is a device and each edge in the list is a connection to another device. In an implementation released to end users, this restriction should be lifted so that sending messages require fewer hops.

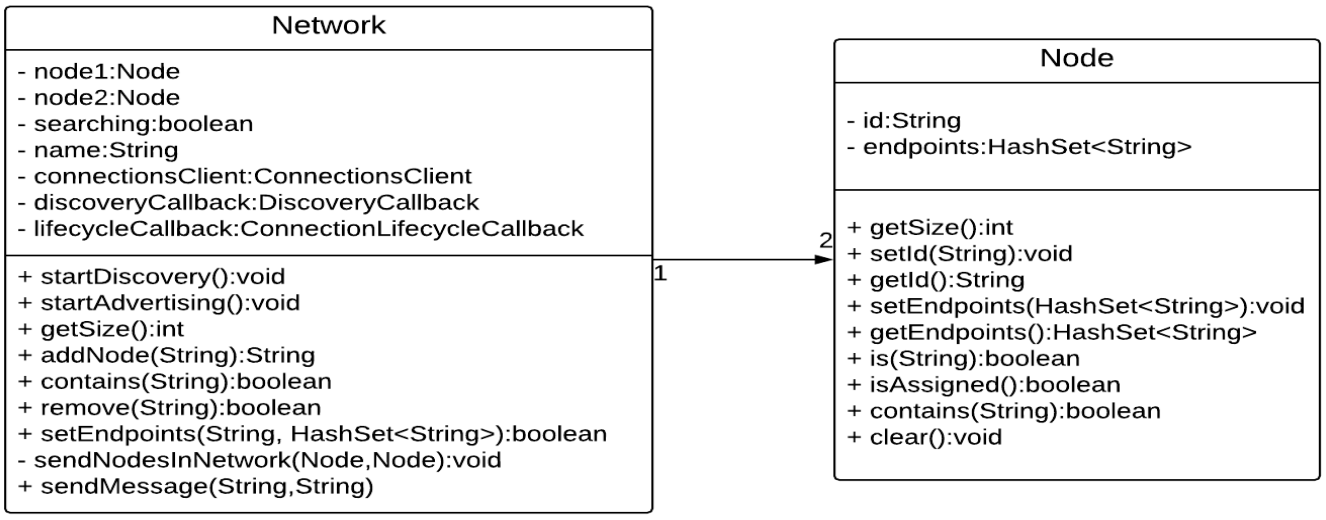

Figure 5: UML Diagram for Network and Node

3.2.1 Classes and Helper Methods. A `Node` object stores useful information about a device, including the device's endpoint id and a set containing the endpoint ids of all the devices connected directly or indirectly to the node. The class also provides a few helper methods for working with nodes, such as a method that checks if the node has been assigned and a method that checks if a discovered endpoint is the same as the node's endpoint id, which helps prevent the node from trying to connect to itself. A `Network` object is a wrapper around node objects for tracking which devices are in a network and for sending messages to devices in the network.

A `SerializationHelper` class provides helper static methods for serializing and deserializing objects into bytes that can be sent as payloads. The implementation is fairly standard and is based on the answer to a popular question [18] on Stackoverflow.

A `CodenameGenerator` class provides a helper method that generates a random 32-bit integer to identify each device in the network by. Its implementation details are trivial and therefore will not be discussed.

3.2.2 Application Setup. The views for the application are set up and an empty network is initialized. The `CodenameGenerator` is used to generate an id for the device. The device simultaneously registers itself as an advertiser and starts discovering other devices.

3.2.3 Discovery and Advertisement. When a device is discovered, the discoverer gets a `onEndpointFound` callback. If the discovered device isn't already in the network, a connection request is sent. Sometimes, two devices will discover each other and both send connection requests. This is known as a simultaneous connection clash and prevents the devices from establishing a connection. Nearby connections handles this by randomly failing one of the device's requests before triggering `onConnectedInitiated()` on both devices [23]. Sometimes, this hangs and results in neither device being able to connect to the network. To handle this edge case, we created an `onFailureListener` that listens for a `STATUS_ENDPOINT_IO_ERROR` status code, which is the code that is returned when the devices clash. The listener method instructs one device to retry sending the connection request to the other device. This ensures that a connection between two discoverers doesn't fail. The current advertisement simply uses the default nearby advertising functions and therefore its implementation details won't be discussed here.

Once `onConnectedInitiated` has been triggered on each device, the application checks again to make sure that the devices aren't both already in the same network. This is necessary, even though this check also occurs before sending a connection request, because multiple devices in the same network A can try to connect to the same device in network B. If the devices aren't in the same network, then each side will accept the connections request by calling `acceptConnection`. Once each device accepts the connection request, and adds the new node to the network with the `addNode` method, which adds the node to the network by either assigning the new device to `n1` or `n2`. Then, the device calls the `network.sendNodesInNetwork` method to merge the networks tracked by `n1` and `n2`. The algorithm for the `sendNodesInNetwork` method fairly simple. The device that calls the message serializes the set of nodes contained in the network, which is obtained by calling `n1.getEndpoints()` or `n2.getEndpoints()`, depending on which nodes are being sent. Once the message is serialized, the message is sent to the other node using the `connectionsClient.sendPayload` method.

3.2.4 Sending, receiving, and forwarding messages. An action listener is connected a button on the application. When the button is pressed, the `network.sendMessage` is called. This serializes the message into a payload and sends the message to the devices tracked by `n1` and `n2` with the `connectionsClient.sendPayload` method. When a device receives a message, it deserializes the message and checks if the message contains data about network merging or if it

contains a regular text message. If the message contains network merging information, the device calls the `network.setEndpoints` method to merge the network and forward the data throughout the rest of the network with the `sendNodesInNetwork` method described earlier. If the message received was a regular text message, the device forwards the message to the rest of the network with the `network.sendMessage` method and then displays the message on the phone screen.

3.3 Challenges Faced During Development

Overall, the development process was fairly straightforward. The documentation Google provides on the nearby connections API is thorough and gave us a good idea of where we needed to start when developing the application. The biggest challenge was lack of community support; we were unable to find solutions on websites such as Stackoverflow to most problems and bugs we would encounter and instead had to rely on sifting through the documentation to find answers. Additionally, some of the sample code Google provides is a bit out of date and doesn't run without a bit of modification, which was confusing when we were determining whether or not nearby connections would be feasible for our purposes.

4 PERFORMANCE EVALUATION

Table 3: Description of relevant parameters

Parameter	Description
Message Size	Size of the sent message
Connection Distance	Distance between two devices
Number of Hops	Number of hops to get to the target
Maximum Range	Maximum connection distance
Connection Speed	Time for a device to join the network
Latency	Time for a message to get to target

Testing applications using the nearby connections API is challenging. The most challenging aspect is that mobile applications using the nearby connections API can only be tested on physical devices that have the hardware components that the nearby framework uses. In other words, emulators don't work, meaning testing cannot easily be automated, and testing the scalability of the application requires access to a large number of Android phones. Due to this restriction, testing thus far has been limited to just three phones: a Pixel 3 running Android 10, a Pixel 1 running Android 8, and a Nokia 3.1 running android 9. Three of the metrics we evaluated were maximum range, connection speed at various distances apart, and latency at various distances apart and with different message sizes.

The testing zone we used was a 100-meter open field in a neighborhood with a slight downwards incline. Distances were determined in 5-meter increments by using a tape measure. Testing was performed on a cloudy and windy day.

4.1 Connection Range

The maximum distance at which two devices were able to connect from was 85 meters apart, but the connection was not stable and the devices would disconnect after a few seconds. Two devices were able to consistently maintain and send messages of up to 4KB from 65 meters apart without ever disconnecting but would fail to 4KB messages from 70 meters apart. Interestingly, two device were able to consistently send 100B messages from 75 meters apart without ever disconnecting, indicating that the size of the message plays a role in whether a connection will be maintained.

The network created supports multi-hop routing, meaning that adding more devices to the network can increase its range. We found that adding a third device to the network increased the range of the network to at least 100 meters without ever disconnecting. The actual range of the network is likely greater, but we were unable to determine the upper bound due to limited testing space.

4.2 Connection Speed

We defined connection speed as the time it took to establish a connection upon opening the mobile application. Connection speed was essentially equal across all distances, and connecting to a network containing only one other device or two other devices took roughly the same amount of time. The vast majority of connections would take between 4-8 seconds, with some connections taking as little as 2 seconds and some connections taking as long as 10 seconds. Upon looking at logging data from the application, it appears that devices that connect in about 2-5 seconds didn't experience simultaneous connection clashing, and devices that connected in more than 5 seconds usually experienced simultaneous connection clashing. The devices that took the longest time to connect always experienced simultaneous connection clashing at least twice.

4.3 Latency

Latency was measured by logging the time at which a message was sent and logging the time at which the message was received and then taking the difference between these two values. When testing with two devices, the Pixel 3 would send the Nokia 3.1 a message 20-40 times at each 5-meter increment. This enabled us to get a sense of what the average latency for a one-hop message is. When testing three devices, we wanted to get a sense of what the latency for a two-hop message might be. The Pixel 3 was put at the right end of the field, the Pixel 1 was put in the middle of the field, and the Nokia 3.1 was put at the left end of the field. The Pixel 3 connected and sent messages to the Pixel 1. The Pixel 1 would then forward these messages to the Nokia 3.1. A diagram showing different distance configurations for the three device test can be seen in Figure 7, where the Pixel 3 is device A, the Pixel 1 is device B, and the Nokia 3.1 is device C.

Average latency tests between two devices (one-hop tests) were performed with message sizes of 8 bytes, the minimum message size the nearby connection APIs supports, 100 bytes, and 4 kilobytes. Average latency tests between three devices (two-hop test) were performed with message sizes of 8 bytes and 100 bytes. The results can be seen in Figure 6. Special note should be taken to carefully observe the y-axis on each plot, as the plots have different scales. Average latencies between two devices with one-hop stayed under 100 ms at up to 60 meters when small message sizes were sent. The average latency was around 200 ms up to 60 meters when a large message was sent. Two-hop latency was less predictable when

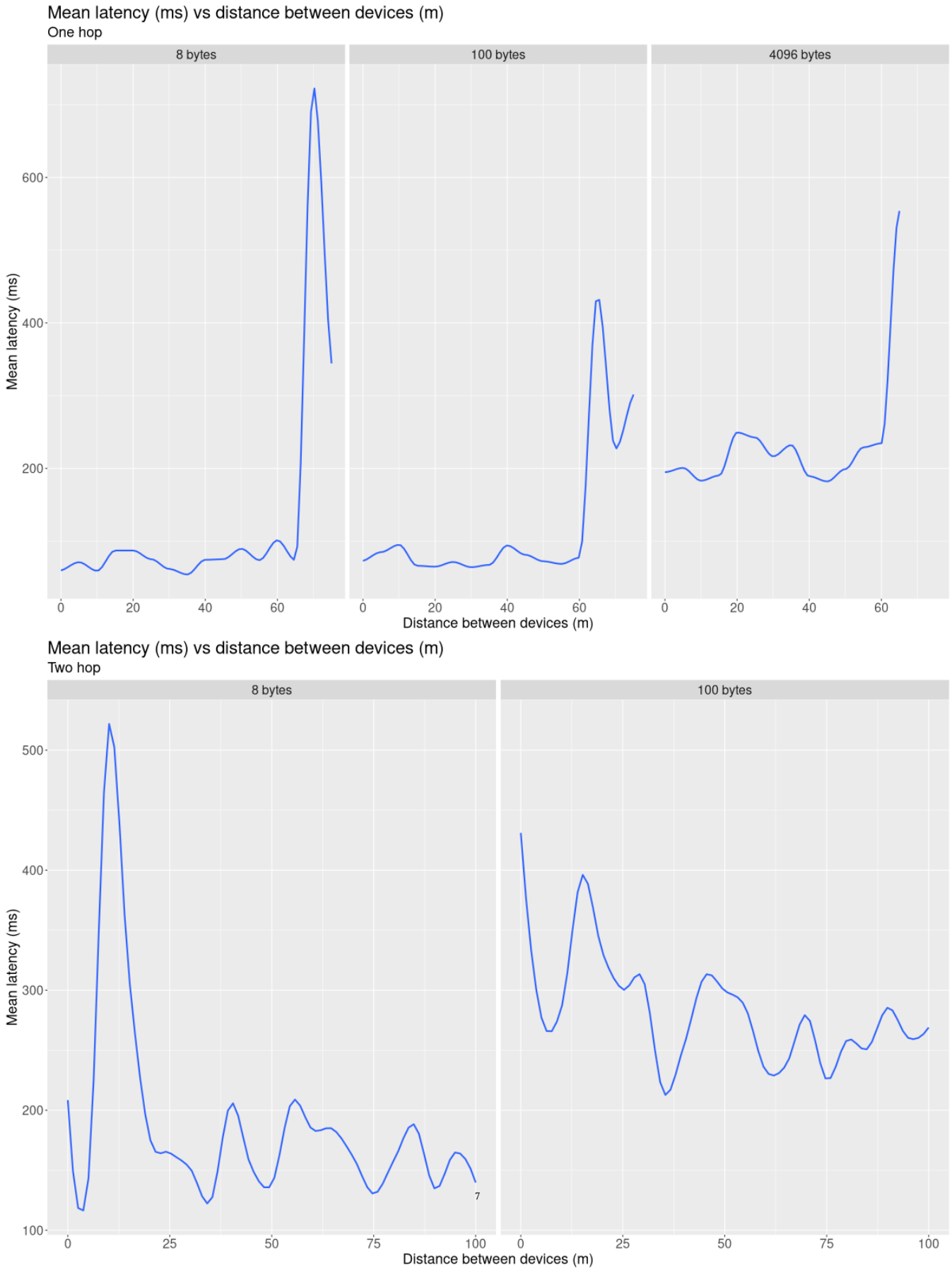

Figure 6: Average Latencies

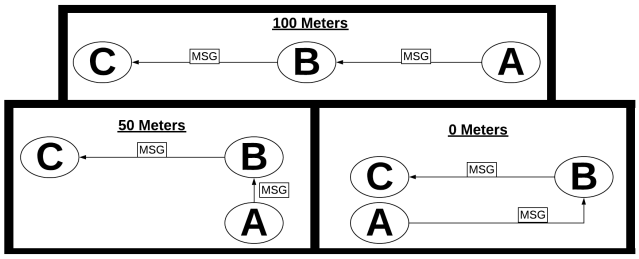

Figure 7: Three device testing configuration at 100, 50, and 0 meters

there were three devices, but overall tended to be higher than when there were two devices and a single hop. The 100 byte two hop test resulted in consistently higher latencies than the 8 byte two hop test.

5 DISCUSSION

In their paper on "Evaluating Bluetooth Low Energy for IoT," Furst et al. provide some metrics on BLE performance in Android phones [10]. To do this, they connected an Android phone to a CC2540 Bluetooth chip located 1 meter away from the phone. Once the phone discovered the CC2540 chip, they measured how long it took to transfer data to the chip using BLE, which supports a maximum message size of 20 bytes. This process can be broken down into three steps: First, the phone accepted the connection to the discovered chip. Then, it discovered the service of interest on the chip that it will write to. Lastly, it exchanged data with the chip (the write operation). They repeated this experiment with 10 different Android phones and found that the operating system implementation and the SoC implementation that schedules how multiple wireless modules share an antena impacted BLE performance on smartphones. Specifically, they found that the latency to perform the write operation ranged between 103-127 milliseconds, depending on the phone. Our one-hop, 8 byte write latency at 0 meters was roughly 60 milliseconds, which outperformed all of the phones Furst et al. tested. This is likely either because nearby connections performed better due to its ability to switch between Bluetooth, BLE, and Wi-Fi hotspots or because our phones are newer models (or possibly a combination of both).

Performance specifically on mobile application MANET applications was not readily available. Kargl et al., who create a MANET using Bluetooth-enabled cellphones for voice transmission, found that their latency for the Bluetooth transmission at 0 meters was about 28ms, or roughly half of our latency [12]. However, they tested their MANET using two computers instead of two phones. The latency for Bluetooth between computers is likely different than the latency between phones, as Bluetooth is accessed indirectly through several abstraction layers and drivers that are operating system dependent. The paper we found on using nearby connections to share and simultaneously playback audio files among peers did not share any performance evaluation for their application [21].

As far as our testing and evaluation goes, the application should be tested more throughly. All tests were static and between at most three devices. It would be useful to test the application with more than three devices and with vehicles to test the network behavior when some devices are moving. We also should explore why the two-hop 8 byte latency test performed significantly better than the 100 byte test.Overall, we found that connections between two devices of up to 60 meters were extremely consistent and could be considered the maximum stable connection range for the application. Anything beyond the 60 meter range risks network failure, with certain network failure in the 80 meter range. Adding more devices to the network will increase the range of the network at the cost of increased latency. In the application's current state, it would be well suited for rural environments where there is less traffic and thus less network latency.

Currently, connection speed is slow, so pedestrians using the application should open the application a few seconds before they arrive at a crosswalk and other participants in the network should have the application open at all times. Alternatively, we could look into running the application in the background so that devices can connect to the network without the application being opened. Device discovery should, in theory, take 2 seconds, but that number is heavily dependent on the devices being used, the stability of the Bluetooth stack that the nearby connections API is calling, and the radio environment. Our connection times are inconsistent and will likely need to be improved to the point where they almost always in the 2-7 second range. To do this, we will need to explore solutions that lower the likelihood of simultaneous clashing, as this was shown to contribute to longer connection times. Since simultaneous clashing only occurs when two devices are discovering and send each other connection requests at roughly the same time, preventing simultaneous clashing will require a solution that prevents a connection request from being sent to a device if that device has already sent it a connection request.

Other miscellaneous improvements could be made to the application. The user interface is currently designed to make testing the application easier, but wouldn't be visually appealing to an end user. To improve latency, we should loosen the restriction that devices can only connect to at most two other devices directly and instead try to connect to as many devices as the hardware will support. This would reduce the average number of hops messages require to get to a device, which would decrease the latency of the application. The architecture of the `Network` class will need to be changed slightly to accommodate this strategy, but it would be easily manageable. This would be a very useful avenue to explore, though it would require more devices to appropriately evaluate.

6 CONCLUSION

In this paper, we present a MANET application that uses Google's nearby connections API to provide connected transportation services to all types of travelers at crosswalks. The multihop MANET supports stable connections at up to 60 meters per hop, with the ability to increase its range by adding more devices to the network. To increase the performance of the application, we should explore increasing the number of direct connections a device in the network can create and investigate ways to reduce the likelihood of simultaneous clashing. Based on our performance evaluation, the MANET application could potentially be useful in rural settings where network size remains low. Tests with more devices and other

measures of network quality should be performed to determine what type of environment the application is suited to supported.

REFERENCES

[1] WiFi Alliance. 2020. *Wi-Fi Direct.* https://www.wi-fi.org/discover-wi-fi/wi-fi-direct

[2] Françoise Bacquelaine. 2009. *La terminologie Bluetooth en Anglais, en Français et en Portugais : étude de néonymie comparée.* Ph.D. Dissertation.

[3] Google Developers. 2017. *Create P2P connections with Wi-Fi Direct.* https://developer.android.com/training/connect-devices-wirelessly/wifi-direct

[4] Google Developers. 2017. *EndpointDiscoveryCallback.* https://developers.google.com/android/reference/com/google/android/gms/nearby/connection/EndpointDiscoveryCallback

[5] Google Developers. 2017. *PayloadCallback.* https://developers.google.com/android/reference/com/google/android/gms/nearby/connection/PayloadCallback

[6] Google Developers. 2018. *ConnectionLifecycleCallback.* https://developers.google.com/android/reference/com/google/android/gms/nearby/connection/ConnectionLifecycleCallback

[7] Google Developers. 2018. *ConnectionsClient.* https://developers.google.com/android/reference/com/google/android/gms/nearby/connection/ConnectionsClient

[8] Google Developers. 2018. *Nearby Connections API Overview.* https://developers.google.com/nearby/connections/overview

[9] Google Developers. 2019. *Nearby Connections API Strategies.* https://developers.google.com/nearby/connections/strategies

[10] Jonathan Fürst, Kaifei Chen, Hyung-sin Kim, and Philippe Bonnet. 2018. Evaluating Bluetooth Low Energy for IoT. https://doi.org/10.1109/CPSBench.2018.00007

[11] P. Johansson, M. Kazantzidis, R. Kapoor, and M. Gerla. 2001. Bluetooth: an enabler for personal area networking. *IEEE Network* 15, 5 (2001), 28–37.

[12] Frank Kargl, Stefan Ribhegge, Stefan Schlott, and Michael Weber. 2003. Bluetooth-based Ad-Hoc Networks for Voice Transmission. 314. https://doi.org/10.1109/HICSS.2003.1174874

[13] Lakhdar Meftah, Romain Rouvoy, and Isabelle Chrisment. 2019. Testing Nearby Peer-to-Peer Mobile Apps at Large. In *MOBILESoft 2019 - 6th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, Denys Poshyvanyk and Ivano Malavolta (Eds.). Montréal, Canada. https://hal.inria.fr/hal-02059088

[14] Morteza Mohammadi Zanjireh and Hadi Larijani. 2015. A Survey on Centralised and Distributed Clustering Routing Algorithms for WSNs. *IEEE Vehicular Technology Conference* 2015. https://doi.org/10.1109/VTCSpring.2015.7145650

[15] Claes Nyberg. 2018. Adapting Android Devices for Multi-Hop Communication in Infrastructureless Ad-Hoc Networks.

[16] J. C. Reyes, E. Burgoa, C. T. Calafate, J. Cano, and P. Manzoni. 2006. A MANET Autoconfiguration System based on Bluetooth Technology. In *2006 3rd International Symposium on Wireless Communication Systems.* 674–678.

[17] Jacob Rodal. 2020. *Connected Crossroads Codebase.* https://github.com/jrodal98/connected-crossroad

[18] Nicolas Schirmer. 2017. *How To Send And Receive An Object In Nearby Connections Api.* https://stackoverflow.com/questions/46515029/how-to-send-and-receive-an-object-in-nearby-connections-api

[19] Alex Scroxton. 2016. *Bluetooth 5 standard brings range, speed and capacity boost for IoT.* https://www.computerweekly.com/news/450298598/Bluetooth-5-standard-brings-range-speed-and-capacity-boost-for-IoT

[20] Eduardo Soares, Pedro Brandão, Rui Prior, and Ana Aguiar. 2017. Experimentation with MANETs of Smartphones. arXiv:cs.NI/1702.04249

[21] J. van den Ende, E. Hamer, L. Hijfte, and D. Wind. 2018. Ad hoc networking on mobile devices: an experimental study on synchronised audio playback. (2018). https://larsvanhijfte.nl/Ad_hoc_networking_on_mobile_devices__an_experimental_study_on_synchronised_audio_playback.pdf

[22] Software Engineer at Google Varun Kapoor. 2017. *How to Enable Contextual App Experiences (Google I/O '17).* https://www.youtube.com/watch?v=1a0wII96cpE

[23] Nearby Engineer Xlythe. 2017. *Nearby Connections 2.0: Both sides request connections, but don't successfully connect.* https://stackoverflow.com/a/46717446/9063770