View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DigitalCommons@CalPoly

A MULTI-CARRIER COLLABORATIVE SOLUTION TO MINIMIZE CONNECTIVITY-LOSS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Michael Wong

June 2016

© 2016
Michael Wong
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	A Multi-carrier Collaborative Solution to Minimize Connectivity-loss
AUTHOR:	Michael Wong
DATE SUBMITTED:	June 2016
COMMITTEE CHAIR:	David Janzen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Seng, Ph.D. Professor of Computer Science

ABSTRACT

A Multi-carrier Collaborative Solution to Minimize Connectivity-loss

Michael Wong

Nearly two-thirds of Americans own a smart phone, and 19% of Americans rely on their smartphone for either accessing valuable information or staying connected with their friends and family across the globe [15]. Staying always-on and always-connected to the Internet is one of the most important and useful features of a smartphone. This connection is used by almost every single application on the device including web browsers, email clients, messaging applications, etc. Unfortunately, the cellular networks on our smartphones are not perfect and do not always have cellular signal. Our devices often lose Internet connection when users are on the go and traveling.

This thesis presents a novel in-depth implementation and evaluation of what we can achieve when a user loses network connectivity. BleHttp, a library for Android, was developed that uses Bluetooth Low Energy to connect to other devices using a different carrier within close proximity of each other and make HTTP requests. In our results, we saw 100% success rates on HTTP requests with connected devices on a good connection. Average round trip times were tested to be as low as 1.5 seconds.

ACKNOWLEDGMENTS

I am extremely grateful for the incredible support system that has been with me throughout this entire process. Thank you so much for all the support:

- My parents, Norman and Cindy.
- My sister, Helen.
- My beautiful girlfriend, Melody.
- My advisors, Dr. Janzen, Dr. Kurfess, and Dr. Seng
- All my friends and family.
- The entire Computer Science Department at Cal Poly.

TABLE OF CONTENTS

Page

LIST OF FIGURES . . . viii

CHAPTER

1 Introduction . . . 1

2 Background . . . 3

2.1 Android . . . 3

2.2 Data Connectivity . . . 4

2.3 Bluetooth . . . 8

3 Related Work . . . 9

3.1 Bandwidth Aggregation Across Multiple Smartphone Devices . . . 9

3.2 Making Use of All the Networks Around Us . . . 16

3.3 Project Fi . . . 20

3.4 Off Grid Communications with Android . . . 21

3.5 Mobile Tethering: Overview, Perspectives and Challenges . . . 22

4 Implementation . . . 26

4.1 Overall Architecture . . . 26

4.2 BleService . . . 30

4.3 BleClient . . . 35

4.3.1 Sending Request . . . 37

4.3.2 Handling Response . . . 38

4.4 BleServer . . . 41

4.4.1 Handling Incoming Request . . . 41

4.4.2 Sending Response . . . 42

5 Performance Evaluation . . . 45

5.1 Experimental Testbed . . . 45

5.1.1 Testing Connectivity . . . 45

5.1.2 Testing Battery . . . 46

5.1.3 Testing Speed and Time . . . 47

5.2 Results . . . 47

6 Conclusion 51
6.1 Future Work 51
BIBLIOGRAPHY 53

LIST OF FIGURES

Figure		Page
2.1	AT&T Coverage along Highway 46 with bad to poor connectivity [12].	6
2.2	Sprint Coverage along Highway 46 with good coverage in the beginning of the highway [12].	6
2.3	Verizon Coverage along Highway 46 with mostly good coverage [12].	7
2.4	T-Mobile Coverage along Highway 46 with very good coverage [12].	7
3.1	Overall architecture of Cell Share [18].	10
3.2	Cell-Share system components [18].	11
3.3	Cell-Share system design [18].	12
3.4	Aggregate bandwidth using static scheduling method for loading a single, 10mb file [18].	14
3.5	Aggregate bandwidth using dynamic scheduling method for loading a single, 10mb file [18].	15
3.6	System diagram of prototype [17].	18
3.7	Throughput during seamless connectivity test [17].	18
3.8	Throughput of stitching two networks [17].	20
3.9	Throughput of stitching ten networks [17].	20
3.10	Battery lifetime of two different devices under test, Wi-Fi tethering various types of traffic [4].	24
3.11	Bandwidth comparison of two devices under test while tethering [4].	24
3.12	Social aspect questionnaire results [4].	25
4.1	Overview of library.	27
4.2	Overview of developer making an HTTP request using BleHttp.	28
4.3	Interactions between BleService, BleClient, and BleServer.	31
4.4	Overview of a packet.	38
4.5	BleClient flow diagram.	40
4.6	BleServer flow.	43
4.7	BleClient and BleServer flow combined.	44

5.1 Average Time for the BleHttp checkpoints during a request. 49

5.2 Average round trip time's for requests with different numbers of send and request packets. 49

5.3 Average RTT and success rate for different types of network conditions on the server. 50

5.4 Battery life of server and client over a course of five minutes. 50

Chapter 1

INTRODUCTION

Surfing the web or “going online” traditionally involved using a desktop or a laptop with a full set of devices including a monitor, keyboard, mouse, and a dedicated high-speed connection. In 2008, a study conducted by eMarketer found that 80% of adults in America accessed the Internet using a desktop compared to mobile (12%) [3]. Today, the online experience is substantially different. Mobile usage in America has increased to 51%; conversely, desktop usage has dropped to 42% [3]. Nearly two-thirds of Americans own a smartphone, and 19% of Americans rely on their smartphone for either accessing valuable information or staying connected with their friends and family across the globe [15].

Staying always-on and always-connected to the Internet is one of the most important and useful features of a smartphone. This connection is used by almost every single application on the device including web browsers, email clients, messaging applications, etc. In fact, Pew Research center reports that 10% of Americans own a smartphone but do not have a dedicated Internet connection at home and 15% own a smartphone but have limited options of going online other than their cellphone [15]. Internet connection on the smartphone has become one of the most dependent features for people around the globe.

Unfortunately, the cellular network on our smartphones is not perfect and does not always have cellular signal. Our devices often lose Internet connection when users are on the go and traveling. When our devices lose connection, we often lose access to almost all content available on the device, thus impacting the usefulness of the device.

Most smartphones are multi-homed, meaning the devices are able to connect to different networks and devices simultaneously through either cellular, Wi-Fi, or Bluetooth. Additionally, we as human beings tend to be in settings where other humans with smartphones may be nearby.

Consider the following use-case:

> A few friends are on a road trip, and each person in the car owns a smartphone with a cellular data connection from different carriers. John loses Internet connection which renders his smartphone useless. The other people in the car all have Internet connection. If they could share their Internet connection, John's smartphone will be usable again.

Features such as Wi-Fi tethering and sharing Internet through Bluetooth exist, but many of these features are restricted by the carrier. Additionally, Wi-Fi and Bluetooth tethering require user interactions which do not flow seamlessly for the user.

The goal of this thesis is to present an in-depth implementation and evaluation of what we can achieve when a user loses network connectivity. This thesis will explore how Bluetooth Low Energy can be used to seamlessly connect to the Internet via HTTP through another device when a user has no connectivity.

Chapter 2

BACKGROUND

2.1 Android

Android is a mobile operating system developed by Google and released in 2008 [9]. The operating system was developed primarily to be a smartphone operating system. Android is based on the Linux kernel and designed for touchscreen mobile devices such as smartphones and tablets. Founded in October 2003, the initial intentions for the company were aimed towards digital cameras. After a more detailed analysis of the market at the time, the company pivoted towards producing an operating system that would rival Symbian and Microsoft Windows Mobile.

As of 2015 Q2, IDC research has reported Android dominating the market share at 82.8% [14]. iOS, the closest Android operating system competitor, owns 13.9% of the market share. Statistic Brain Research Institute reported 1.6 billion Android devices sold worldwide and an average of 355,000 Android devices being activated each day [11]. The Google Play Store, Android's main application store, releases on average 15,000 new applications for users to download per day. The Statistics Portal reported, as of July 2015, Android users were able to choose between 1.6 million apps compared to Apple's App Store at 1.5 million available apps [13]. Android has seen tremendous success in the smartphone industry. Every day, Android touches billions of people on Earth to help their lives in one way or another.

AppBrain Statistics reported that travel and local apps were one of the top ten categories in the Google Play Store [1]. With over 80,000 apps in travel and location and over 28,000 apps in transportation, using a smartphone on the go while traveling is a very common use case in today's world. Travel applications such as Yelp provide

users the ability to find local businesses nearby and see how others have reviewed them. The Waze application empowers users to join one of the largest community-based traffic and navigation apps. Users traveling in a car can get real time alerts from others on traffic, accidents, hazards, and even police reports. The popular application, Gas Buddy, allows users to find the closest gas stations with near real-time gas prices crowd sourced by the community. Users are able to sort gas stations by distance and price to plan the closest and cheapest gas station for their next stop. Google Maps, the king of travel apps with over one billion installs, provides real time route planning for cars, public transportation, biking, and walking. All these applications that are used by millions of people every day have one thing in common: they require an Internet connection.

2.2 Data Connectivity

The need for data connectivity is greater than ever. Applications with social networks have become the norm. Yelp, Waze, Gas Buddy, and Google Maps are all applications that are deemed must-haves when traveling on the go. However, Internet coverage on the go is not perfect. Consumers tend to only have one telecommunications carrier that they associate with and often one carrier will have Internet connection in an area that another carrier won't have. Discrepancies in coverage between these different carriers happen everywhere in the world.

RootMetrics, a leader in coverage reports, provided an in-depth detailed assessment of coverage in the United States for 2015 [12]. Over 6.1 million tests were performed, 237,000 miles driven, and 7,300 indoor locations were tested in this report. Network Reliability for the top four carriers in United States were:

- Verizon - 94.5%

- AT&T - 91.8%
- Sprint - 87.5%
- T-Mobile - 82.0%

With T-Mobile rating 10% less reliable than Verizon, the top carrier, reliability discrepancies are clear and areas of improvement can be made. We can see examples in the real world with highways that have poor network coverage for one carrier but significantly better for different carriers. An example of this is on Highway 46 in California near Blackwell's Corner:

RootMetrics reported AT&T with bad or poor coverage on Highway 46 near Blackwell's Corner (Figure 2.1). We can see in Figures 2.2, 2.3, and 2.4 that other carriers with significantly better coverage.

Sprint and Verizon showed coverage that was better than AT&T on Highway 46 near Blackwell's Corner but had areas that were either bad or untested. T-Mobile had complete coverage along the stretch of the highway with all areas showing good to fair coverage. Examples of network discrepancies like this can be found all over the world. In our use case, people tend to travel with other passengers in the car and often passengers do not share the same carrier coverage. Technologies such as Bluetooth enable us the power to transfer data between phones and then share Internet connectivity from whichever device has carrier connectivity.

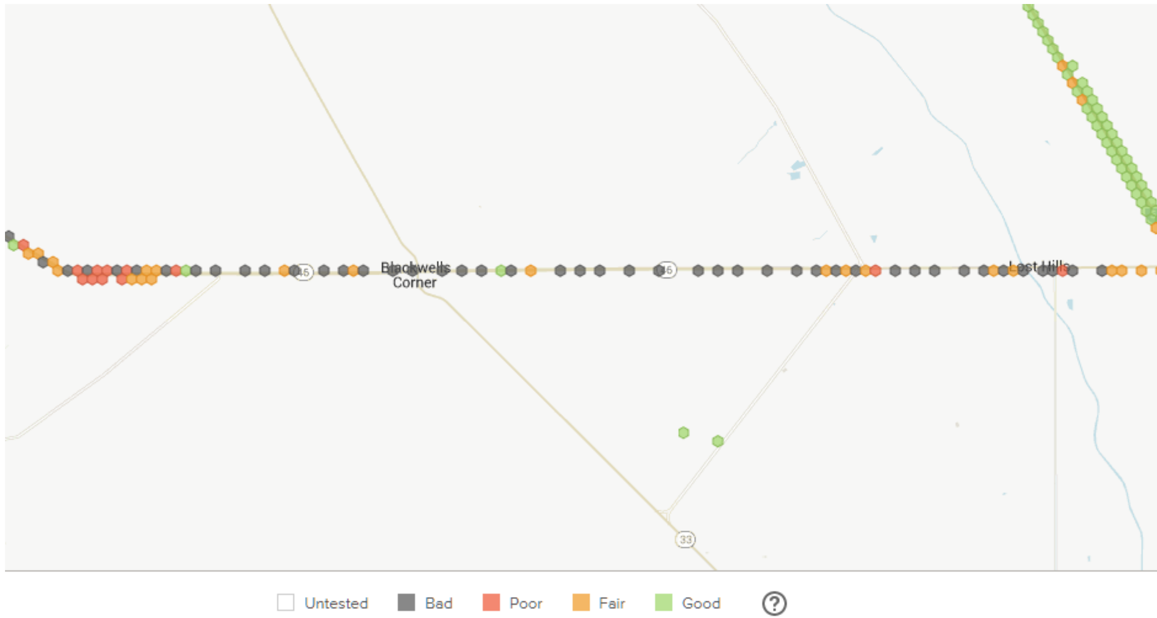


Figure 2.1: AT&T Coverage along Highway 46 with bad to poor connectivity [12].

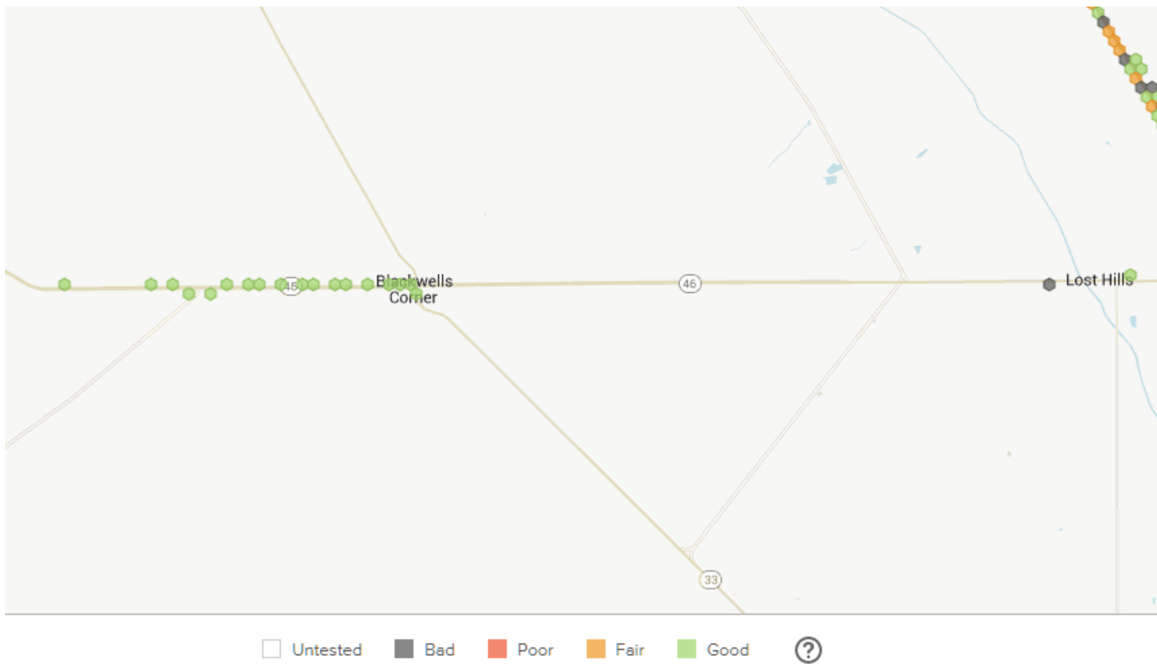


Figure 2.2: Sprint Coverage along Highway 46 with good coverage in the beginning of the highway [12].

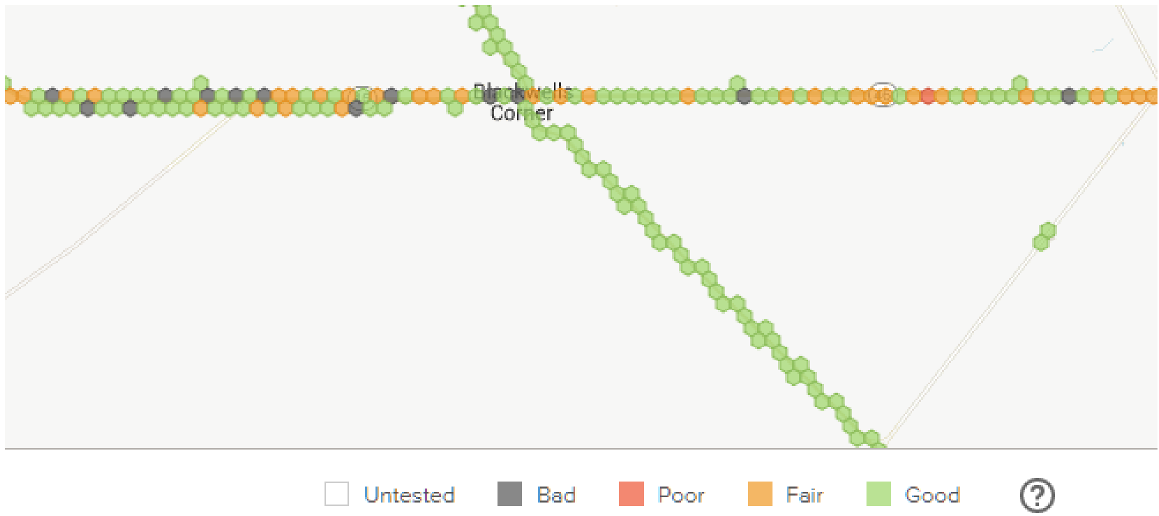


Figure 2.3: Verizon Coverage along Highway 46 with mostly good coverage [12].

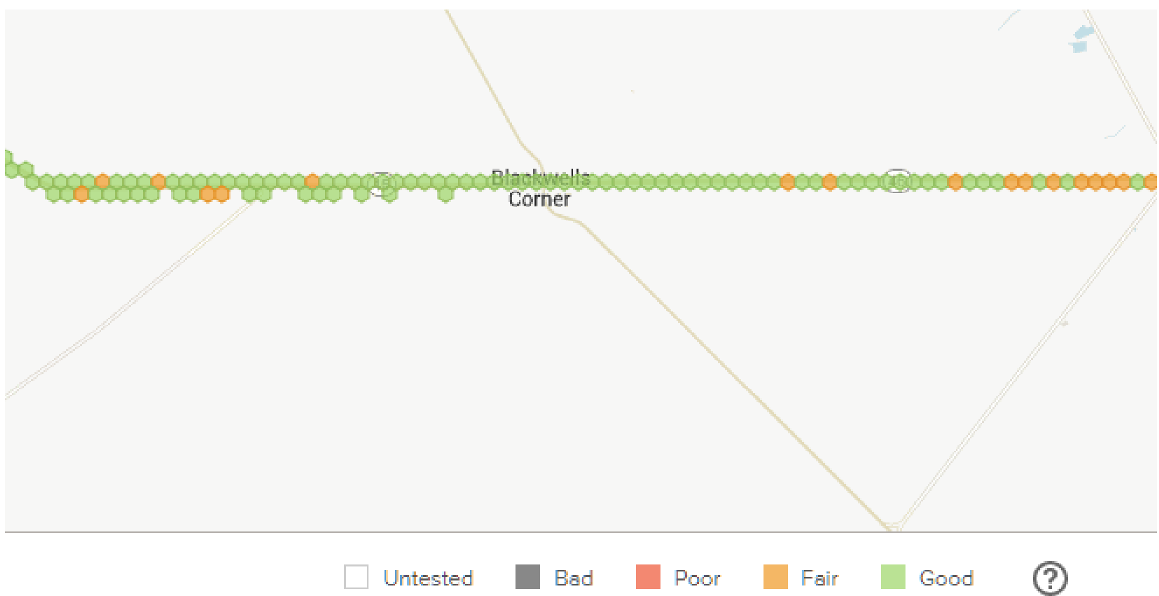


Figure 2.4: T-Mobile Coverage along Highway 46 with very good coverage [12].

2.3 Bluetooth

Created in 1994, Bluetooth is the standard for exchanging data between nearby devices [2]. Created as an alternative to data cables, Bluetooth exchanges data by using radio transmissions. One of the most popular applications that Bluetooth empowers is using wireless audio headsets. Bluetooth allows high quality streaming that is optimized to send even high quality streams of data such as music.

In Android 4.3, Bluetooth Low Energy (BLE) introduced built-in platform support in the central role and provides APIs that apps can use to discover devices, query for services, and read/write characteristics [7]. In contrast to Classic Bluetooth, BLE is designed to provide significantly lower power consumption. This allows Android apps to communicate with BLE devices that have low power requirements, such as proximity sensors, heart rate monitors, fitness devices, etc.

In Android 5.0, BLE was updated to allow an Android device to act as a Bluetooth LE peripheral device [6]. Apps can now use this capability to make their presence known to nearby devices. For instance, developers can build apps that allow a device to function as a pedometer or health monitor and communicate its data with another Bluetooth LE device.

Using BLE we can have a device with no Internet connectivity enter central mode to scan for a device with connectivity in peripheral mode. Because the device is in peripheral mode and automatically makes its presence known to nearby devices, we can create a seamless user experience for both users to share their Internet access. It is important to note that a device looking to connect to a device nearby needs to be on Android version 4.3 or greater to support scanning or central mode. To support a connection from a device, a device needs to be on Android version 5.0 or greater.

Chapter 3

RELATED WORK

As this thesis presents a novel idea in minimizing connectivity loss, I will discuss some works that are related to networking and sharing data connectivity through smartphone devices.

3.1 Bandwidth Aggregation Across Multiple Smartphone Devices

Zeller [18] focuses on the problem that bandwidth offered by cellular networks is often much lower than ones that we typically experience on our standard home networks. This unfortunately leads to a less-than-optimal user experience making it challenging and frustrating to access certain types of web content such as video streaming, large file downloads, loading large web pages, etc.

Given that most modern smartphones are multi-homed and capable of accessing multiple networks simultaneously, Zeller's thesis attempts to utilize all available network interfaces to achieve the aggregated bandwidth of each to improve the overall network performance of the phone. Zeller implements a bandwidth aggregation system for the iOS operating system that combines the bandwidths of multiple devices located within close proximity of each other. In summary, Zeller's results saw speedups of up to 1.82x for downloading a single, 10MB file, and web page loading speedups of up to 1.55x when deployed on up to three devices.

To explore bandwidth aggregation methods, a system called Cell-Share was implemented on top of the iOS platform. This application allows users to form local communities among nearby peers. With this community of peers, Zeller is able to pool Internet connections together. The main goal of the thesis is to determine the

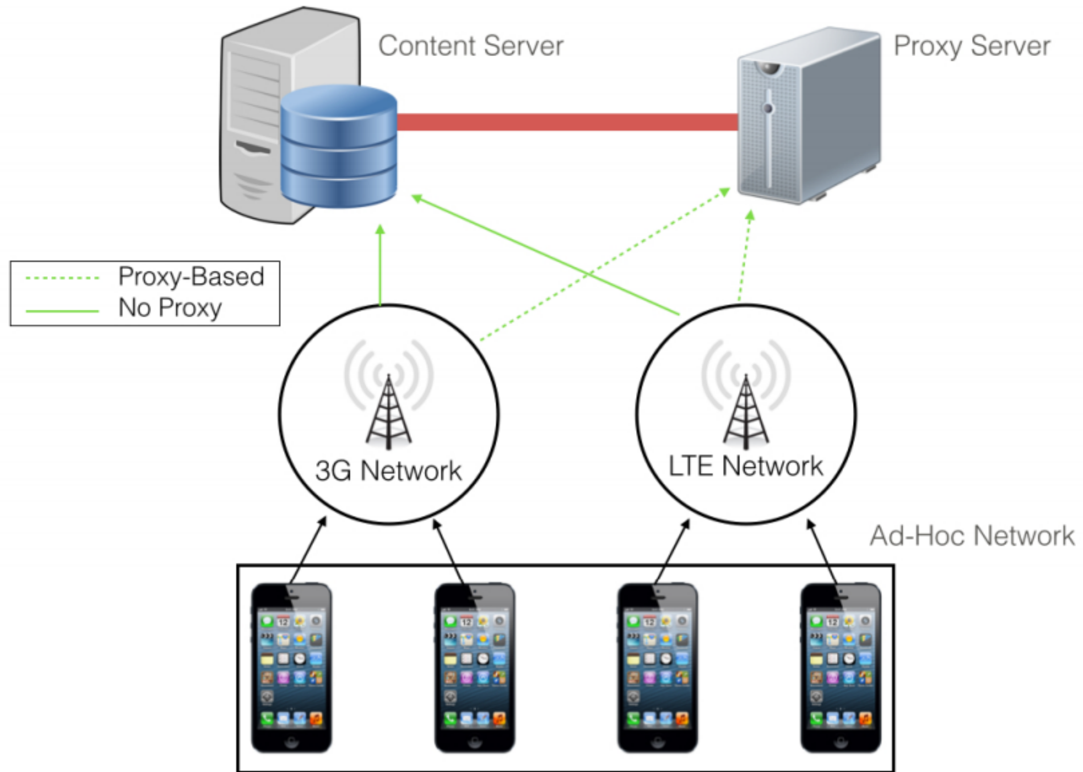


Figure 3.1: Overall architecture of Cell Share [18].

maximum performance capability of iOS devices.

The overall architecture of Cell-Share is depicted in Figure 3.1. Once a community of nearby devices is formed, each member of the community establishes a connection, over their cellular interface, with the proxy server. When a member makes a request for data, the request is routed through the proxy server. The proxy server makes the request to the content server and when the response arrives, the proxy splits the downstream traffic among each connection belonging to the corresponding community. Each member of the community that received data on behalf of another member forwards their segment of data through the ad-hoc network. This effectively load balances the response data across all available links, rather than being limited by just one device.

In Cell-Share, each smartphone device sharing bandwidth belongs to an ad-hoc

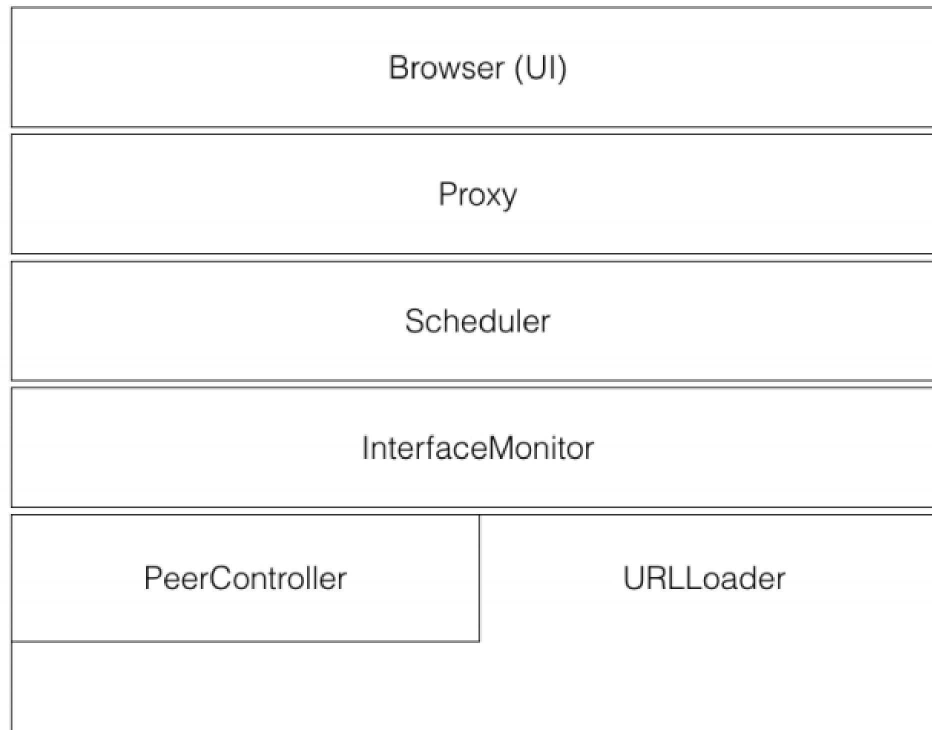


Figure 3.2: Cell-Share system components [18].

network device and each device is exactly one hop away from any member. The ad-hoc network is connected through either Bluetooth or Wi-Fi and can be used to manage collaboration as well as relay response data, effectively eliminating the need for a proxy server.

Figure 3.2 illustrates the Cell-Share system components and how they work together. Figure 3.3 illustrates Cell-Share's system design and shows an example of what a request and response may look like.

The browser represents the user interface and imitates a simple browser with an additional function to allow the user to connect to other nearby neighbors. The proxy layer enables the system to intercept each request that is generated by the browser. When a request is intercepted, it is sent to the scheduler. The Scheduler's main responsibility includes handling the scheduling logic and deciding to which devices

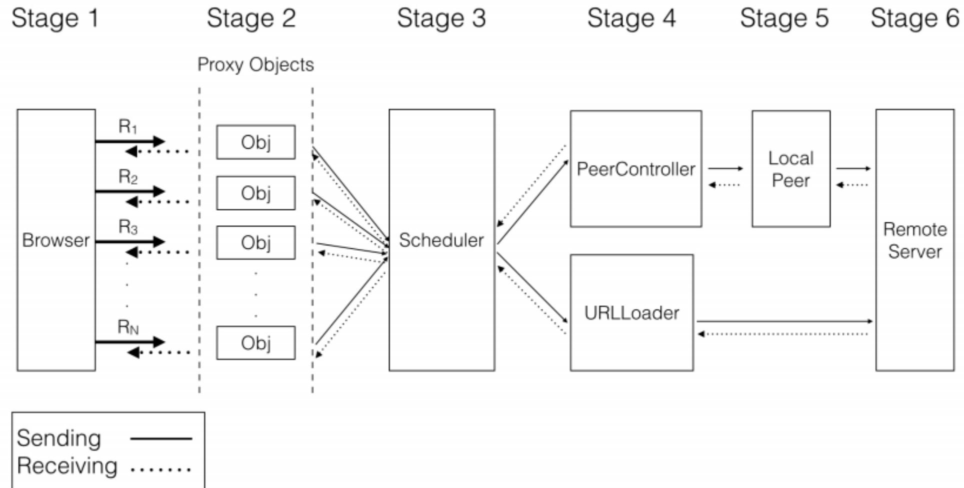
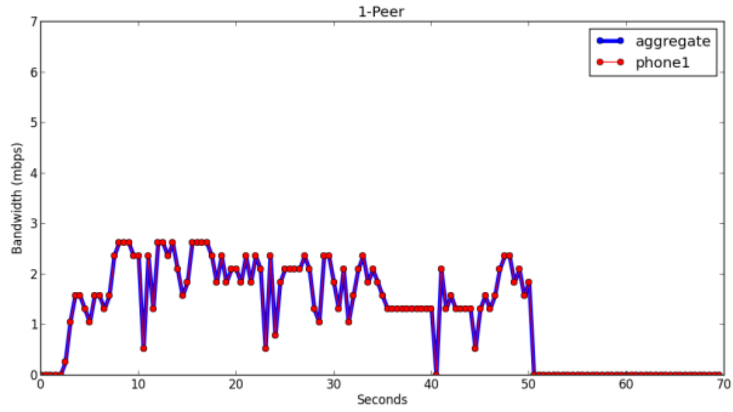


Figure 3.3: Cell-Share system design [18].

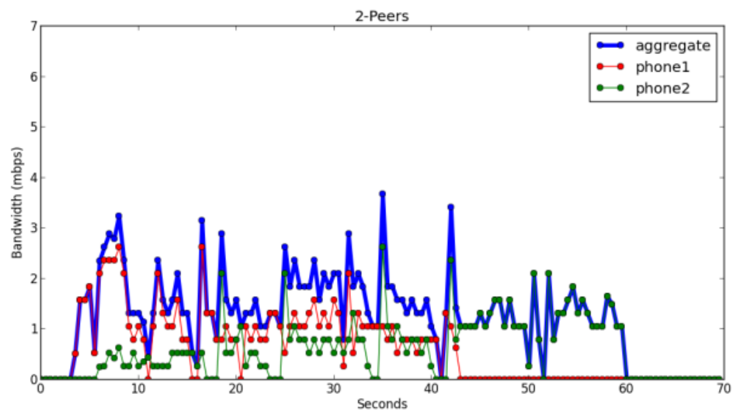
to off-load the device. In order to make informed scheduling decisions, all interfaces need to be monitored. InterfaceMonitor is responsible for recording several details on each interface. The Scheduler may talk to the InterfaceMonitor for more intelligent scheduling algorithms. Once a decision has been made, the request under consideration is either passed to the URLLoader if it is to be loaded on the current device's own cellular connection, otherwise it is passed to PeerController. The URLLoader is quite simple and simply fetches data from the network asynchronously. The PeerController represents the local, ad-hoc network. To generalize, the PeerController provides the same programming interface as the URLLoader and is used to relay data payloads and manage collaboration between members.

To test Cell-Share, three devices were used, one iPhone 5, and two iPhone 5s devices. All phones were on Verizon's 3G or LTE network and placed within several inches of each other indoors. Four different web pages were used to test web page loading: apple.com, surfline.com, maps.google.com, and theverge.com. To test single file downloads, a file of 10MB was downloaded from Dropbox. Two main types of scheduling algorithms were used in testing: static and dynamic scheduling. Dynamic

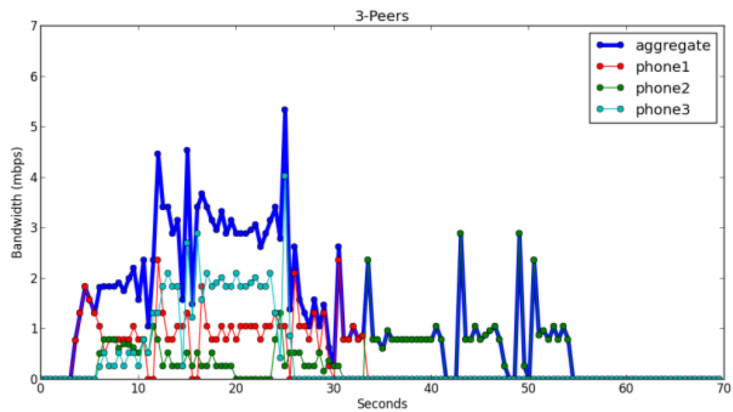
scheduling offloads the workload for each interface proportional to its capabilities. We can see the results of Cell-Share in figure 3.4 and 3.5. The aggregate bandwidth performs much better for both two and three devices in session using a static and dynamic scheduler.



(a) Bandwidth for 1 device.

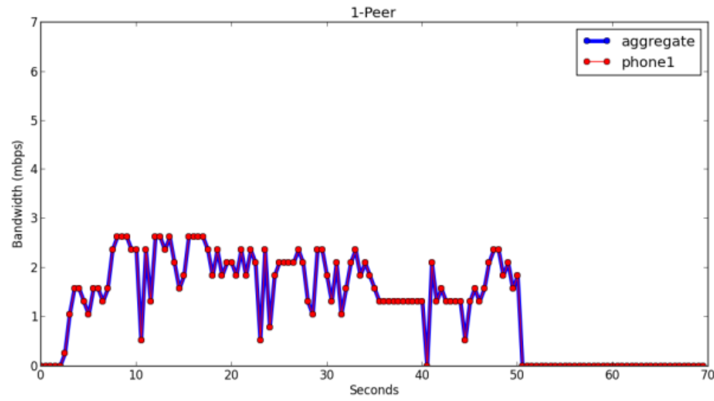


(b) Aggregate bandwidth for 2 devices.

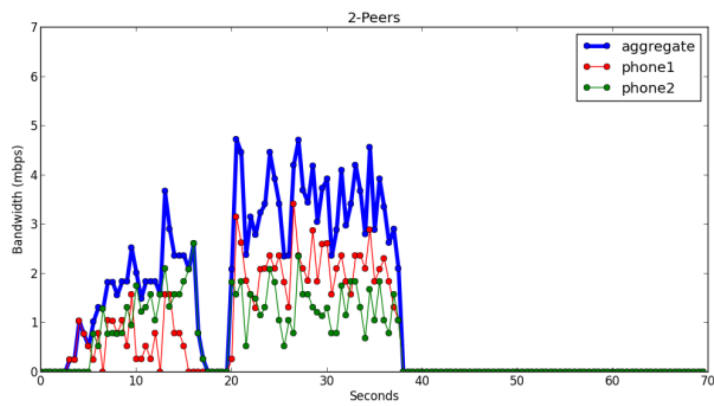


(c) Aggregate bandwidth for 3 devices.

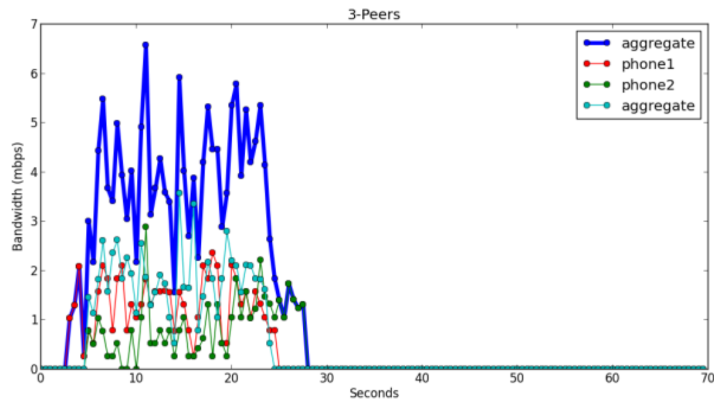
Figure 3.4: Aggregate bandwidth using static scheduling method for loading a single, 10mb file [18].



(a) Bandwidth for 1 device.



(b) Aggregate bandwidth for 2 devices.



(c) Aggregate bandwidth for 3 devices.

Figure 3.5: Aggregate bandwidth using dynamic scheduling method for loading a single, 10mb file [18].

3.2 Making Use of All the Networks Around Us

Yap et al. [17] discusses a solution to poor connectivity when using wireless networks on the go. They present a solution by taking advantage of the multiple networks around us. Using multiple networks at a time makes it possible for faster connections, seamless connectivity, and potentially lower usage charges. They explore how to make use of all the networks around us with a prototyped solution on an Android phone.

The idea behind the paper is that smartphones are armed with multiple radios capable of connecting to several networks at a time. Today's phones commonly have four or five radios (e.g. 3G, 4G, LTE, Wi-Fi, Bluetooth) and in the future they will have more. We typically only have one or two active radios, either Wi-Fi or LTE, whichever offers us a more stable wireless connection. However, if a smartphone can take advantage of multiple wireless networks at a time, then the user can experience seamless connectivity, faster connections, lower usage charges, and lower energy usage.

To implement this solution, the paper modified the Android and Linux operating system to allow a mobile device to use multiple interfaces. Android by default only allows one network interface to be active at a time. It chooses which interface to use according to a preference order. If a device is connected to a Wi-Fi network, Android will automatically disconnect from WiMAX. The authors modified Android's Connectivity Service to allow the capability of using multiple interfaces simultaneously.

Next, to spread traffic from one application over multiple interfaces, an application was written that sends traffic using one IP source address. The networking stack takes care of spreading the traffic over several interfaces, each with its own IP address. A virtual Ethernet interface is used to connect to the application with its local IP address to a special gateway inside the Linux kernel. This gateway stitches multiple interfaces together without the application knowing (similar to a load-balancer). Figure 3.6

illustrates the system diagram of the prototype.

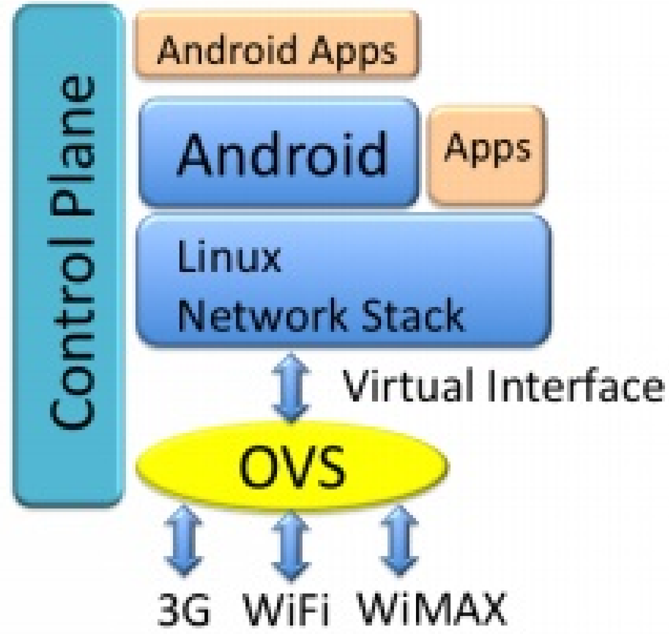


Figure 3.6: System diagram of prototype [17].

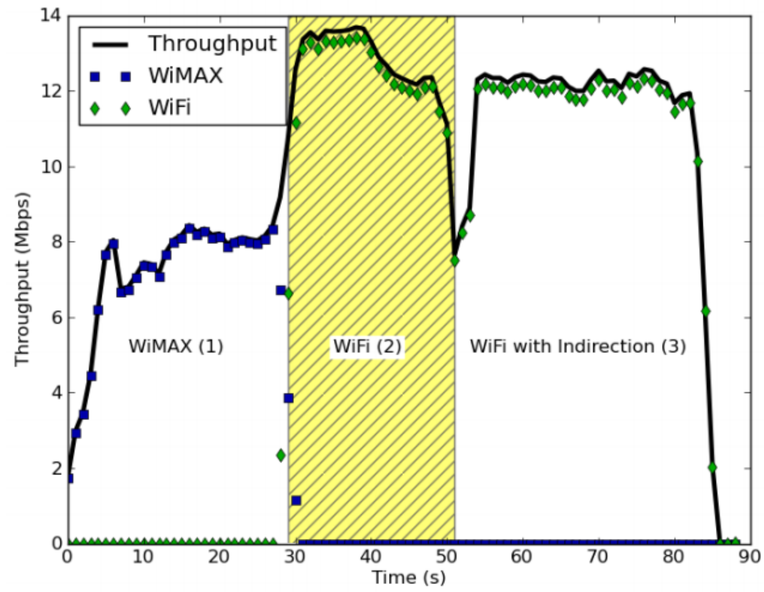


Figure 3.7: Throughput during seamless connectivity test [17].

To test seamless connectivity of this prototype stack, a simple model was created: A user arriving to work who wishes to migrate an ongoing video stream from a public WiMAX network to a corporate Wi-Fi network. Figure 3.6 shows the throughput of the user as he is migrated through the flow. Throughput is never dropped, and in fact, increases immediately after the migration.

The prototype allows multiple networks to be used simultaneously. To test how this works, data was streamed while varying the number of interfaces and throughput was measured. In the experiment, a 100MB file is downloaded using five parallel connections. Figure 3.8 shows the test results. Figure 3.9 shows the results of ten different network interfaces stitched together. From these results, we can clearly see a higher throughput as more networks are stitched.

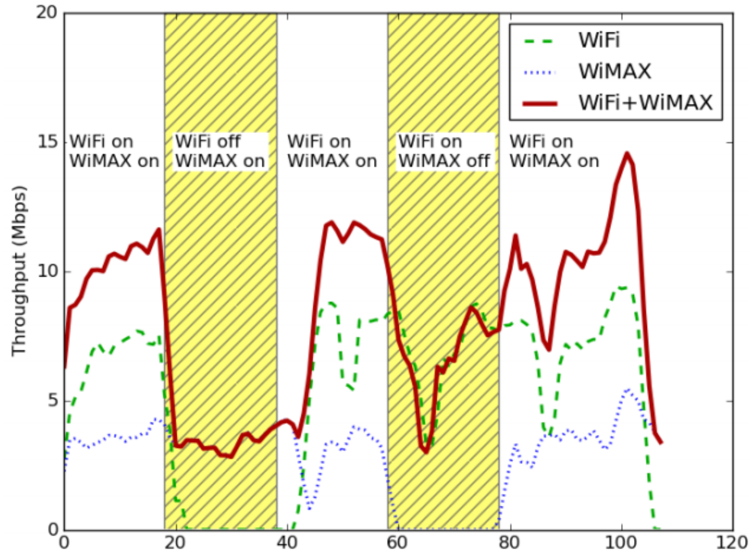


Figure 3.8: Throughput of stitching two networks [17].

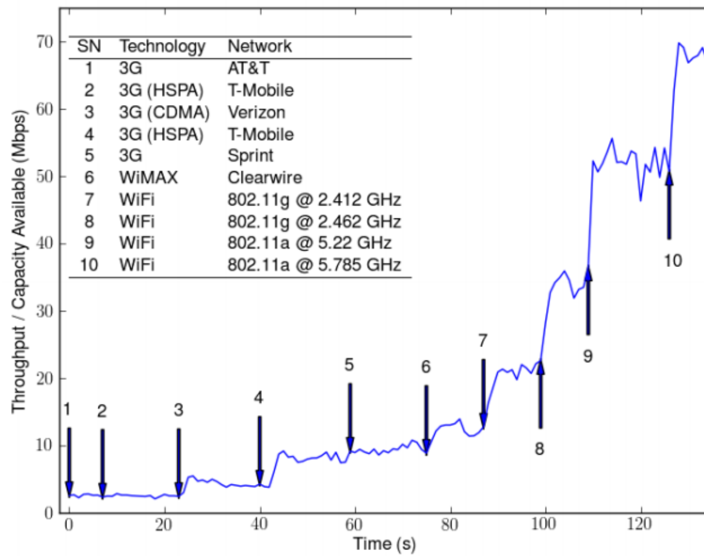


Figure 3.9: Throughput of stitching ten networks [17].

3.3 Project Fi

As of 2016, smartphone devices typically support only one LTE network. What is unique about Project Fi is its ability to pick the best of two 4G LTE networks on a

smartphone device [8]. Project Fi has partnered with Sprint and T-Mobile to launch its cellular service. If a person uses a Project Fi cellular plan with a Project Fi compatible phone, Project Fi will automatically determine which of the two networks is faster. Additionally, Project Fi has been renowned for their seamless connectivity. If a phone call is started over Wi-Fi and the connection is lost, Project Fi seamlessly transitions the call to a cellular network, keeping the phone call in session.

3.4 Off Grid Communications with Android

Off Grid Communications with Android written by Josh Thomas [16], presents a project named Smart Phone Ad-Hoc Networks (SPAN), an open source implementation of a generalized Mobile Ad-Hoc Network framework (MANET). SPAN's goals are to bring dynamic mesh networking to smartphones and to explore the concepts of Off-Grid communications. A recent incident in Fukushima (2011) required the need for a ubiquitous mobile mesh network. During and soon after the disaster, world media reported numerous stories regarding the lack of connectivity and viable communication channels for rescue and energy response personnel. If a ubiquitous mobile mesh network had been in place as a backup, the situation could have been drastically different. For example, friends and family members could have stayed in contact through text messaging and voice over IP phone calls. The SPAN project aims to solve these issues by utilizing MANET to provide a resilient backup framework for communication between individuals when all other infrastructure is unavailable or unreliable.

This paper utilizes the SPAN project which has implemented an application that allows developers to enable Ad-Hoc mode on Android hardware. Using this application, the paper conducts a field test and evaluates effective range, limits of simple multi-hop outing, and node density limitations of Ad-Hoc on Android.

At the time of the paper, it was uncommon to see a device offering configuration of the built-in wireless chip. Devices that support configuration of the wireless chip were found to share similar wireless chips manufactured by Broadcom. Thus devices with the Broadcom BCM4329 and BCM4330 wireless chips were chosen to be test devices. Some of these devices include the Samsung Nexus S, Samsung Galaxy Tab 10.1, Samsung Galaxy Nexus, iPhone 4S, and Nokia Lumia 900.

It was observed that each node utilizing a Broadcom BCM4329 Wi-Fi chipset could be a maximal distance of 106 feet from its closest neighbor and still maintain MANET connectivity. With devices using BCM4330 chipset, the maximal distance was observed to be 98 feet.

Initial testing did not reveal an upper limit on multi-hop communications. A simple chat conversation was able to traverse a 5 hop network with minimal delay and throughput problems. The paper expects more testing to be done in the future with 10-25 node traversals.

Because of the nature of 802.11 specifications, it is expected that there is an upper limit of devices that can exist in the same peer to peer MANET enclave. However, this limit was not reached during an initial test of 30 devices. The paper expects more testing to be done in the future.

3.5 Mobile Tethering: Overview, Perspectives and Challenges

Constantinescu et al. [4] analyzes the challenges of mobile tethering from a technological and social perspective. Although the technology is ready and has promising outcomes, service providers and users still keep their distance. Incentives for users and service providers should be identified.

To analyze challenges in terms of energy and bandwidth consumption, an appli-

cation was specifically developed for mobile tethering. Figure 3.10 tells us video and radio streaming are high battery consumption tasks, but when there is no traffic or low network sync tasks, the impact on battery life is minimal. Figure 3.10 depicts the bandwidth while tethering is exceptional. Figure 3.12 shows usage issues studied through a conjoint analysis in which technical and social aspects were analyzed.

In summary, this paper presents findings that show that although energy, bandwidth and security are important, users are mainly concerned about social aspects. Decisions such as with whom the connection will be shared are the most important challenges to tethering. This paper deduces that mobile tethering is a viable cooperative service, but only when users are familiar with the person with whom the data connection is being shared.

Mobile tethering performance results have indicated that it is a viable solution and allows more people access to the Internet while they are mobile even if they do not have a cellular data subscription. Additionally, more Internet-based services can be offered to people when they are traveling or roaming in other countries. Service providers have shown the ability to satisfy security and privacy concerns for the users, but service providers tend to block mobile tethering technology as they do not have control and do not expect to gain revenues.

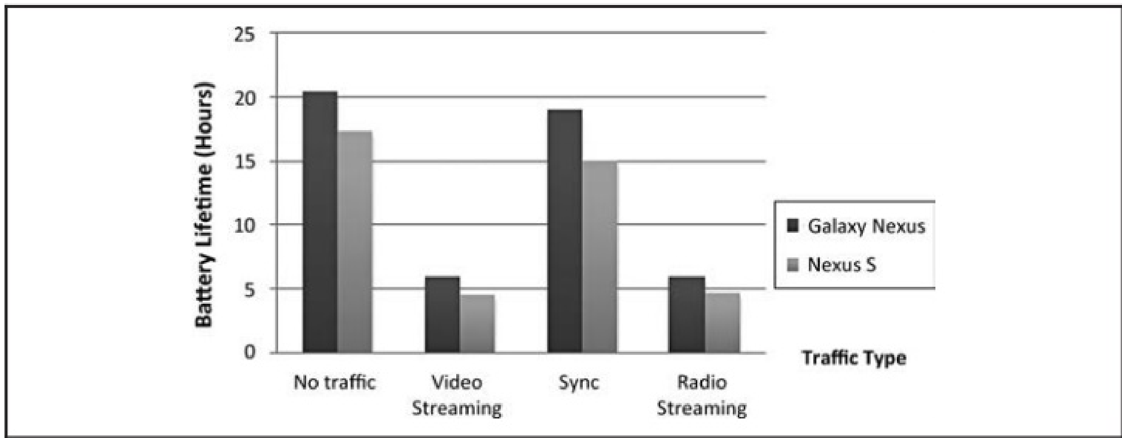
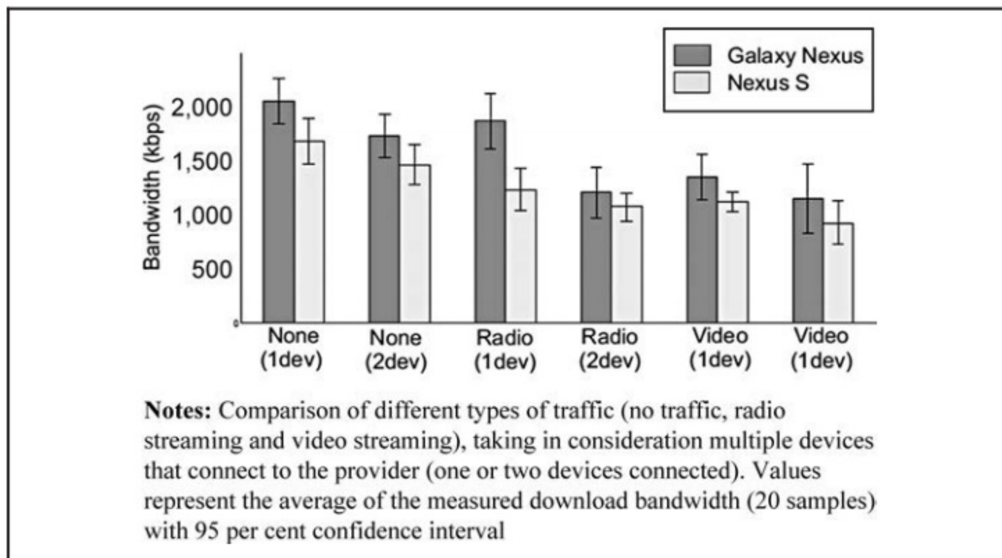


Figure 3.10: Battery lifetime of two different devices under test, Wi-Fi tethering various types of traffic [4].



Notes: Comparison of different types of traffic (no traffic, radio streaming and video streaming), taking in consideration multiple devices that connect to the provider (one or two devices connected). Values represent the average of the measured download bandwidth (20 samples) with 95 per cent confidence interval

Figure 3.11: Bandwidth comparison of two devices under test while tethering [4].

Table V Conjoint results for the dependent variable questions (consumer)

Attributes	*Levels*	*Q1: Sharing for free* *Utility*	*Q1* *Importance (per cent)*	*Q2: In exchange of financial compensation* *Utility*	*Q2* *Importance (per cent)*	*Q3: In exchange of virtual currency or reputation* *Utility*	*Q3* *Importance (per cent)*	*Q4: How concerned would you be regardi ng your privacy?* *Utility*	*Q4* *Importance (per cent)*
Costs	No connection	0.276	36	0.046	12	0.108	19	0.186	13
	Higher	−0.276		−0.046		−0.108		−0.186	
Quality of service	Normal	0.022	3	−0.038	10	−0.004	*1*	0.203	14
	Lower	−0.022		0.038		0.004		−0.203	
Battery lifetime	Longer	−0.004	*1*	0.029	*8*	0.050	9	0.225	16
	Shorter	0.004		−0.029		−0.050		−0.225	
Persons involved	Familiar (family, friend, co-worker)	0.338	*44*	0.154	*40*	0.246	*44*	−0.647	*45*
	Not Familiar (unknown person, public)	−0.338		−0.154		−0.246		0.647	
Subscription Type	Unlimited	0.123	16	0.117	30	0.154	27	−0.175	*12*
	Limited	−0.123		−0.117		−0.154		0.175	
Pearson's r		0.904, $p < 0.001$		0.916, $p < 0.001$		0.952, $p < 0.000$		0.927, $p < 0.000$	
Kendall's tau		0.764, $p < 0.004$		*0.643*, $p < 0.015$		0.857, $p < 0.001$		*0.593*, $p < 0.022$	

Note: Italic values in table show the most and the least important attributes according to the respondents

Figure 3.12: Social aspect questionnaire results [4].

Chapter 4

IMPLEMENTATION

A library named BleHttp was implemented on top of the Android platform. BleHttp is an Android library that allows developers to make network requests through Bluetooth Low Energy (BLE). The library automatically handles peer discovery, peer connection and forwarding and receiving of the network request.

The reason BLE technology was chosen over other technologies such as Wi-Fi, Wi-Fi direct, and standard Bluetooth was because of the need to provide a seamless experience to the user. With BLE, we are able to setup devices to automatically advertise to other nearby devices and setup a connection without user interaction. From a user perspective, there is no dialog or prompt that the user needs to enter before allowing this to happen.

This library supports networking between two identical applications. Because this library is implemented on the application level, both devices need to have the same application running in order for it to work. For example, if both devices have the application Yelp installed and Yelp implemented BleHttp, both devices would support forwarding and receiving network requests. However, if device one had Yelp installed, and device two had a different application named Waze installed which also implemented BleHttp, this would not work.

4.1 Overall Architecture

Figure 4.1 shows a high level overview of the library. Assuming that there are two devices, device one and device two, Figure 4.1 presents how the devices will interact with each other to make a network request. We begin with device one requesting

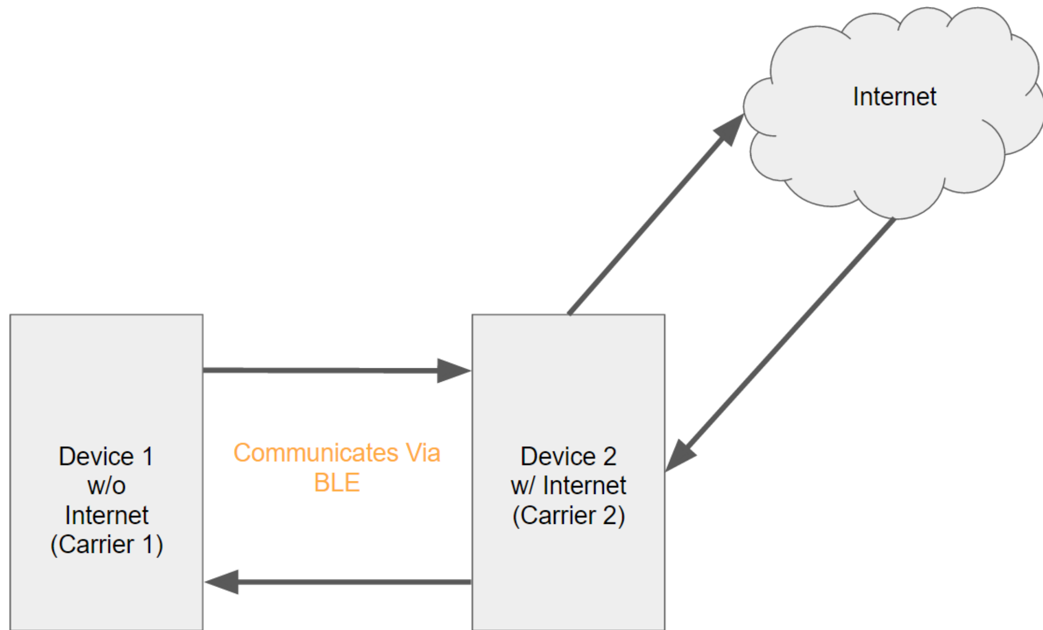


Figure 4.1: Overview of library.

to make a network request, but has no data connectivity at present. By using BLE, device one can search and connect to device two. Device one proceeds to sends the request through BLE to device two. Once device two has received the request, the phone will process the network request through its own data connection. After the network request is complete, device two will forward the response through BLE back to device one.

Because of the complex nature of BLE, the main goal of the library was to make it as easy as possible for the developer to implement the library into their application. I wanted to ensure that developers did not need any knowledge of Bluetooth prior to using the library. The developer only needs to perform three tasks to use the library:

1. BleService must be started with a universally unique identifier (UUID) for each application. This UUID allows the library to identify which application on the device is making the HTTP request.

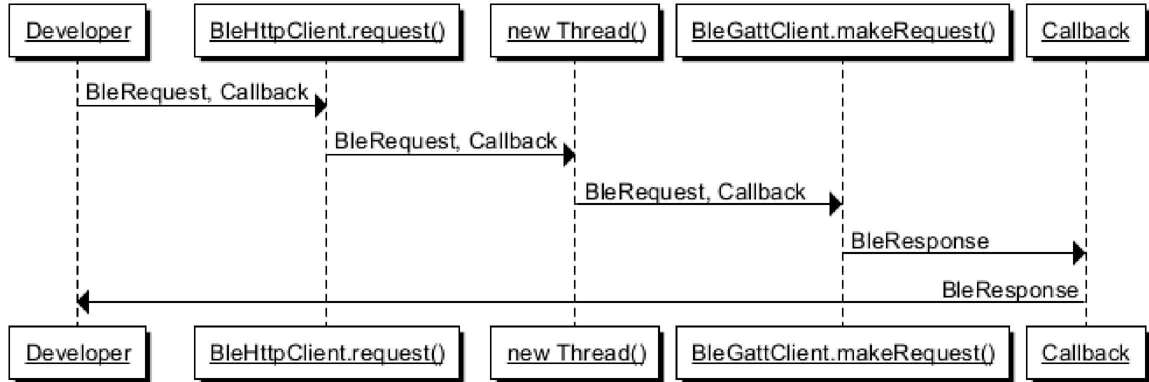


Figure 4.2: Overview of developer making an HTTP request using BleHttp.

2. A BleRequest object must be constructed to make an HTTP request. This includes the HTTP request information: URL, method, and body.

3. The developer must call BleHttpClient.request(), passing in the BleRequest object and a callback object which will contain the response or the error when the request is finished.

BleService must be started because it contains the main bulk of the Bluetooth Low Energy logic. BleService initializes and starts many of the core services that BleHttp needs. These include initializing the BleServer and BleClient.

Assuming BleService has been started, Figure 4.2 depicts a high level overview sequence diagram of a developer making an HTTP request. A BleRequest object contains three strings: a URL, method, and body. The URL string is the network request that the developer wants to make. The method string can only be either GET or POST; support for other methods of HTTP request are not supported. A body to the HTTP request is typically added for POST requests to send data from the application to the server. To construct an instance of a BleRequest object, a static Builder class inside the BleRequest class is provided for the developer to build the BleRequest object for convenience.

Once the developer has constructed the BleRequest object, he/she can make a request through BleHttpClient passing the request object and a callback. This callback will later be called when the request finishes and is either successful or failed. Inside the BleHttpClient.request() method, an object named BleClient is obtained and a new thread is automatically created to handle the request off the main UI thread. Next, we pass the request to BleClient and it will handle processing the request. BleClient will interface and communicate with the BleServer object that was constructed in BleService. I will discuss in detail both BleClient and BleServer in later sections. Once BleClient has received the response from a nearby device, it will notify the developer through the callback object.

Inside our callback object, we have two methods: done() and error(). If the HTTP request was successful, then the done() method will be called passing an object named BleResponse. BleResponse includes the body and the HTTP code of the response. If the request failed then the error() method will be called passing an error string. It is important to note here that done() will be called as long as the HTTP request was successfully made and HTTP response was received. For example, if the HTTP request failed, resulting in HTTP code 400, done() will still be called. The error() method will be called for all other errors.

There are three main objects that are vital to making this process happen: BleService, BleServer, and BleClient. BleService acts as the middleman and initializes core BLE services and notifies BleServer and BleClient when devices have connected/disconnected and sending/receiving data. BleServer is responsible for handling the logic of parsing incoming requests and building outgoing responses. BleClient is responsible for handling the logic of building the request that BleServer will receive and is responsible for parsing the incoming request.

4.2 BleService

The BleService's main responsibilities include handling the overall connection and data communication with BLE devices as the server role, and the initialization of both the BleServer and the BleClient. Figure 4.3 shows an overview of the interactions between BleService, BleClient, and BleServer.

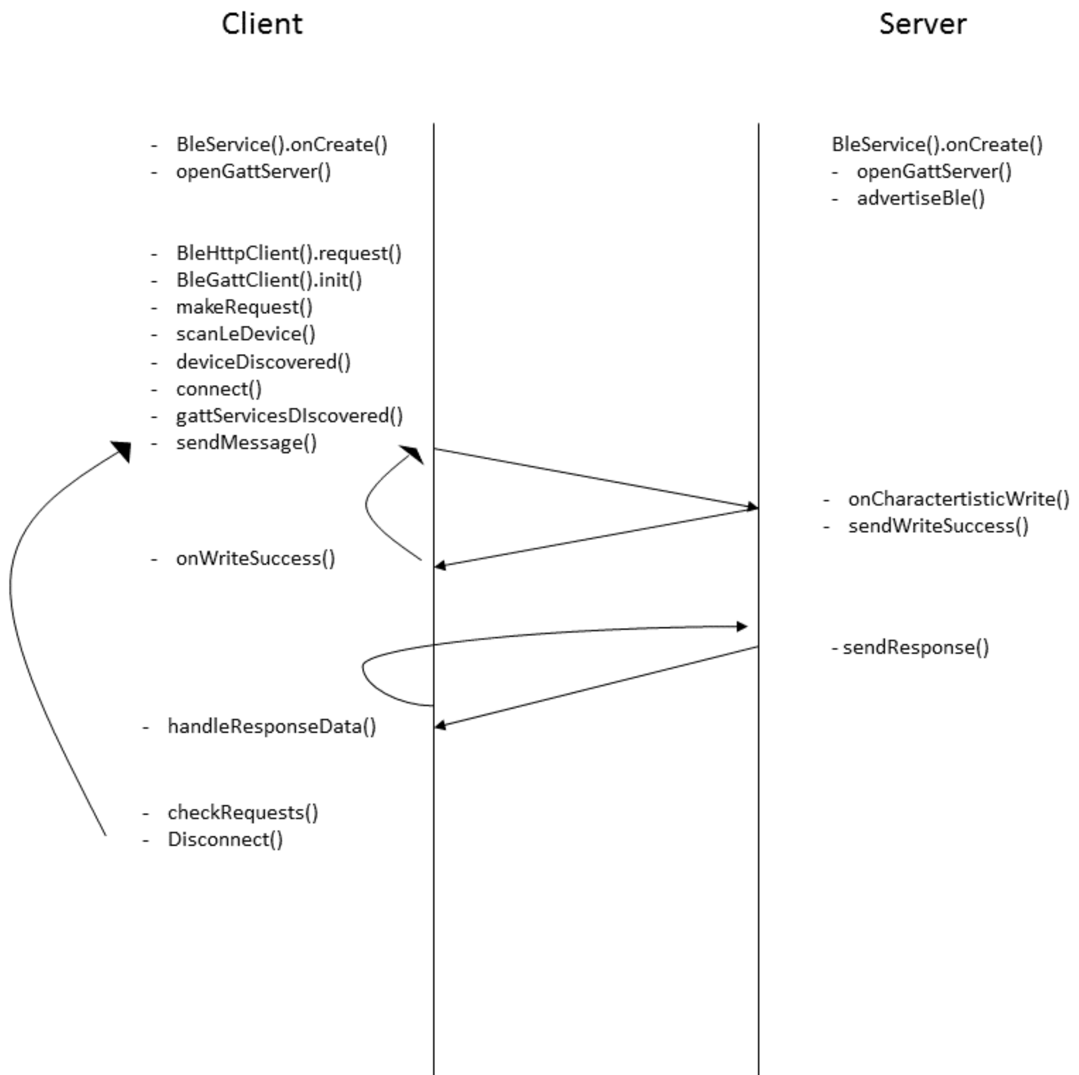


Figure 4.3: Interactions between BleService, BleClient, and BleServer.

To handle overall connection and data communication, BleService implements the following four Bluetooth GATT callbacks:

- OnConnectionStateChange(Gatt, Status, State): A callback indicating when GATT client has connected/disconnected to/from a remote GATT server.
- OnServicesDiscovered(Gatt, Status): A callback invoked when the list of remote services, characteristics and descriptors for the remote device has been updated, i.e when new services have been discovered.
- OnCharacteristicWrite(Gatt, Characteristic, Status): A callback reporting the result of a characteristic write operation.
- OnCharacteristicChanged(Gatt, Characteristic): A callback triggered as a result of a remote characteristic notification.

Note: GATT (Generic Attribute Profile) is a general specification for sending and receiving short pieces of data known as "attributes" over a BLE link. All current Low Energy application profiles are based on GATT [7].

To handle these callbacks, we have a local event bus running in our application. This was chosen for the ability to implement a publish-subscribe-style communication between components without requiring components to explicitly register with one another (and thus be aware of each other) [10]. The event bus in our library will support pushing five different events:

- DeviceDiscovered(Device)
- DeviceDisconnected
- ServicesDiscovered
- ResponseAvailable(Response)
- SendSuccess

These events will be used to notify other components in the library of the numerous stages at which the request is currently.

In OnConnectionStateChange, BleService will be notified whether we have connected to a device or disconnected. If a device is connected, BleService will attempt to call discoverServices(). DiscoverServices(), will allow the library to detect whether the connected device has support for BleClient. If BleService is notified that the device is disconnected, BleService will unregister and close the connection and push a DeviceDisconnected event.

In OnServicesDiscovered, BleService will check if it was successful in obtaining the list of GATT services and push a ServicesDiscovered event. This will later be handled by BleClient.

In OnCharacteristicWrite, BleService will check if the write request was successful and push a SendSuccess event. In our library, a write request is used to send data to the connected device. This will later be handled by BleClient.

In OnCharacteristicChanged, BleService will push a ResponseAvailable event. This will later be handled by BleClient.

Upon creation of BleService, it checks to see if the current device supports BluetoothManager. BluetoothManager is a high level manager used to obtain an instance of BluetoothAdapter and conducts overall Bluetooth management. If an instance of

BluetoothManager is obtained, then we attempt to retrieve the BluetoothAdapter. The BluetoothAdapter is important because it lets you perform fundamental Bluetooth tasks such as initiate device discovery, query a list of bonded (paired devices), instantiate a BluetoothDevice using a known MAC address, and create a BluetoothServerSocket to listen for connection requests from other devices, and start a scan for Bluetooth LE devices.

Once we have verified that Bluetooth is supported on the current device, we can begin the construction of BleServer. Once BleServer is created, we proceed to call openGattServer(); this allows the current device to support handling any incoming data requests. Next, BleService enables BLE advertisement.

BLE advertisement allows us to broadcast our device and a small amount of advertisement data. This allows other devices to search for us and send connection requests. It is important to note here that Bluetooth Low Energy advertisement is an API that was recently added in Android Lollipop [7]. Furthermore, phones that are on Lollipop or greater may or may not support advertisement if the hardware does not support it. Because of these reasons, not all Android devices will support peer discovery and receiving HTTP requests. In our BLE advertise settings, we set the advertisement mode to balanced. This gives us a balance between advertising frequency and power consumption.

Finally, we proceed to creating an instance of BleClient which will be used later to send requests and read responses.

BleService also includes a few additional methods which will be used by BleClient:

- Connect(Address): Connects to the GATT server hosted on the Bluetooth LE Device.
- Disconnect(): Disconnects an existing connection or cancel a pending connec-

tion. The disconnection result is reported asynchronously through the OnConnectionStateChange callback.

- Close(): This method to ensure resources are released properly after a BLE connection.
- WriteCharacteristic(Characteristic): Request a write on a given Bluetooth GATT Characteristic.
- SetCharacteristicNotification(Characteristic, Enabled): Enables or disables notification on a given characteristic. This is so the BleClient can listen for the response from BleServer.
- GetSupportedGattServices(): Retrieves a list of supported GATT services on the connected device. This is used to verify the connected device supports the BleHttp library.

4.3 BleClient

The BleClient's main responsibility is to handle sending the HTTP request to device two and receiving the response to forward to the developer.

After the developer calls BleHttpClient.makeRequest(), BleClient receives the request. BleClient first checks to determine if there is an on-going request already. If a request exists, then the request will be added to a queue. Assuming that there is no on-going request, BleClient does the following steps:

1. Scan for a BLE device and connect to it.
2. After the device is connected, we must verify that it supports the BleHttp library. To do this, we call BleService.getSupportedGattServices() and look for a BLE service that has the same UUID as the UUID that was used to initialize

BleService. Additionally, we scan for a unique BLE characteristic with a UUID that is unique to our library. BLE characteristics are a basic data element used to construct a GATT service The characteristic contains a value as well as additional information and optional GATT descriptors [7].

3. If the BLE service and characteristic is found, we register for notifications on the characteristic. This allows device two to notify us when there is data available to be read. We will be using this to handle the incoming response of our network request.

4. Data is now sent.

5. Response is received and is forwarded to the developer through the callback.

6. Check for remaining requests, if so, process them.

7. Disconnect from connected device.

We will also note here that BleClient subscribes to the following events:

- DeviceDiscovered: A device has been discovered, BleClient will take the device address and call connect() from BleService.

- DeviceDisconnected: The connected device has disconnected. BleClient will search for new devices on the next request.

- ServicesDiscovered: BleService pushes this event after a device's services have been discovered. BleClient will proceed with the verification process detailed above when this event has been posted.

- SendSuccess: A send request was received successfully.

- ResponseAvailable: An incoming response is available.

4.3.1 Sending Request

Bluetooth Low Energy only supports sending and receiving data transmissions of up to 20 bytes per transmission. Because of the nature of Bluetooth Low Energy, sending the request is a complex matter requiring us to split the data into numerous "packets". The device on the other end must then be able to store and combine these packets before being able to fully read the request. In this section, we will focus on the details of sending a request.

Because a BleRequest contains three data structures: the URL, method, and body; it is converted into a JSON string to send. The JSON format is a simple array with three indices. Index one is used to hold the URL, index two is used to hold the method, and index three is used to hold the body. An array was chosen over a JSON object with keys and values because of the 20 byte limitation. We are able to save a few bytes by not having to store the keys and using index positions to imply the key in a JSON array.

Once the BleRequest is converted into a JSON string, we can begin the sending process. We must now define what a packet looks like during a data transmission For each packet during transmission, we reserve the first two bytes as the header. The first byte in the header signals whether the packet is a continuation packet or a finish packet. If it is a continuation packet,device two will know to continue waiting and if it is a finish packet, device two can now combine all the packets to form the BleRequest. The second byte in the header denotes the sequence number of the packet. Because BLE write requests occur asynchronously, there is a possibility that a later packet will be received earlier than an earlier packet. Additionally, due to the sequence number limited to one byte, there exists a limitation that only 256 packets can be sent for a BleRequest. Increasing this limitation is possible, but will reduce the amount of bytes a packet can hold for data thus increasing the number of packets to send per

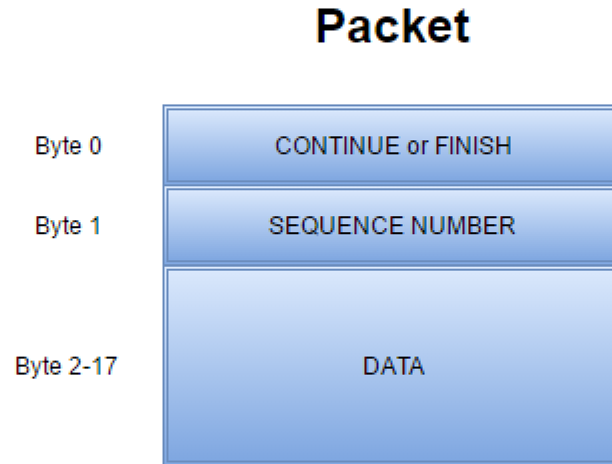


Figure 4.4: Overview of a packet.

request. The last 18 bytes are then reserved for the data of the split up JSON string. Figure 4.4 illustrates the data format of a packet.

Once a packet has been sent, we must wait for the connected device to acknowledge the transmission was successful. If the connected device acknowledges, our BleService from earlier will receive a OnCharacteristicWrite callback and consequently push a SendSuccess event. When we receive a SendSuccess event, BleClient will check and send any remaining packets.

4.3.2 Handling Response

Handling a response is similar to sending a request. Because of the same 20 byte limitation as sending, response handling must store and keep track of incoming packets before parsing and reading the response. When a connected device has received the BleRequest, successfully retrieved the HTTP response, and has started the response forwarding procedure, BleService's OnCharacteristicChanged callback will begin to start pushing ResponseAvailable events.

When a ResponseAvailable has been received, BleClient holds a two-dimensional array of size 256 x 18. We have 256 rows because there can only be 256 packets sent for any data transmission. There are 18 columns for each row to hold the data of each packet. Similar to a request packet, the response packets have the same header values, byte one as continuation or finish, and byte two as the sequence number. The sequence number of the response is used as the row index to the two-dimensional array to save the packet's data. BleClient checks byte one. If it is a finish byte, the two-dimensional array is parsed into a String which is then parsed into a BleResponse (covered in next section). The callback that was associated to the original BleRequest is then called with the response. To finish, BleClient will then check the request queue to process the next request. If there is no request, BleClient will disconnect from the connected device and the process is finished. If there is a request to process, BleClient will process the next request.

Figure 4.5 summarizes what we just covered.

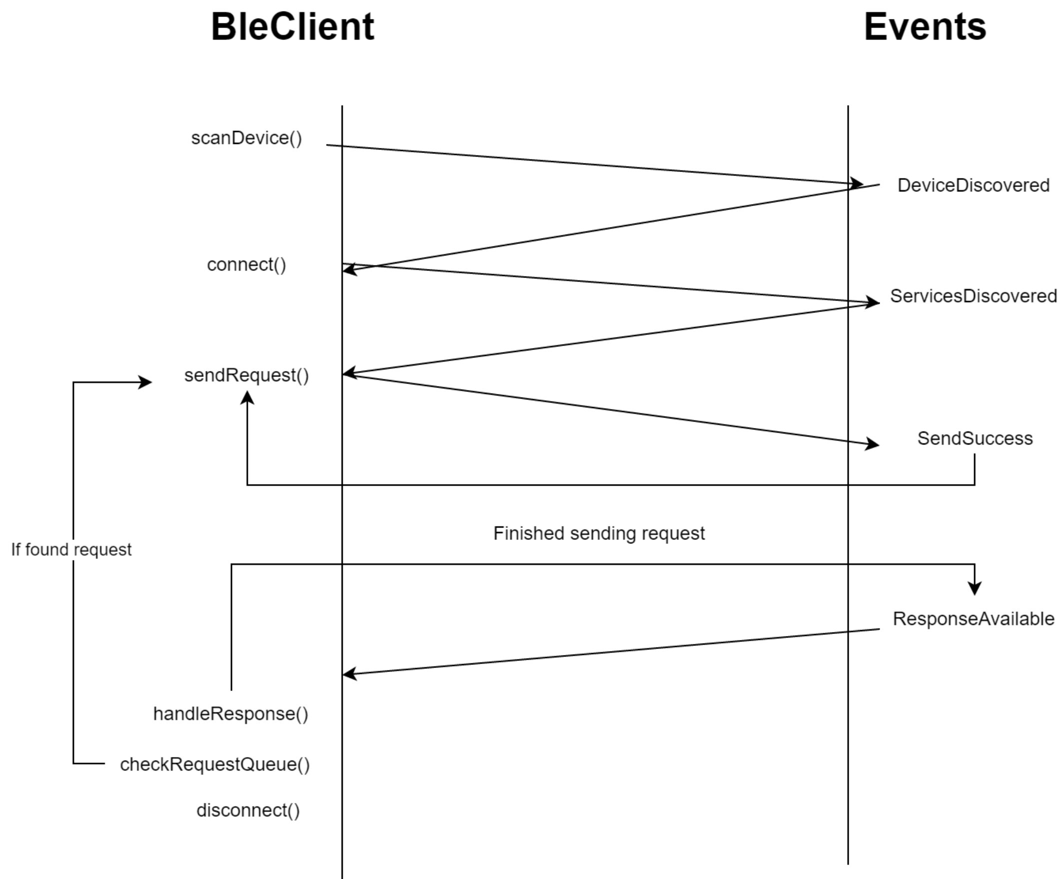


Figure 4.5: BleClient flow diagram.

4.4 BleServer

BleServer's main responsibilities are to open a GATT server to allow a connected device to send their BleRequest, interpret and fulfill the incoming request, and send the response.

Recall that one step of starting BleService was to construct BleServer and call openGattServer(). This method opens a GATT server by using BluetoothManager.openGattServer(). We provide a BluetoothGattServerCallback object which handles one callback - onCharacteristicWriteRequest. From BleClient, a request is sent to the connected device using writeCharacteristic(). When the request lands to the connected device, our BluetoothGattServerCallback will receive the data and BleServer will handle it. Handling response data will be covered in a later sub section.

After the GATT server is opened, we must add a GATT service and a GATT characteristic. Recall that BleClient scans for a device and needs to verify that the device supports our library. A GATT service is initialized with the UUID that the developer passed in to BleService. Next, a GATT characteristic is created with write permissions and a UUID: 247a6c1c-9fd2-42c7-8670-94af4b4ab754. This UUID was generated randomly and is guaranteed to be universally unique. This is the main characteristic with which both BleClient and BleServer communicates. The GATT characteristic is added to the GATT service which is then added to the GATT server and our GATT server is successfully setup at this point.

4.4.1 Handling Incoming Request

When BleClient sends a packet, onCharacteristicWriteRequest is hit in our BleServer. Recall that our packet includes a header with a continuation or finish byte and a sequence number. Additionally, Android BLE provides a few additional parameters,

one being the BluetoothDevice itself. This allows us mapping devices to requests to support multiple devices sending requests.

In BleService, we have a hash map of BluetoothDevice to a two-dimensional byte array, this will keep track of the request data as we are building it. When a request comes in, we pull the two-dimensional array from the map and, similar to handling a response in BleClient, we use the same logic to handle the request in BleService. The sequence number denotes which row the data will be placed in the array, and the continuation/finish byte will denote when the transmission is finished. OnCharacteristicWriteRequest is hit each time a packet is sent, which in return BleServer will store the packet data in the map.

Once we successfully have the request, we can now remove the request from our map and a network request is made to fetch the response. When the response arrives, we parse it into an object named BleResponse. BleResponse contains two data structures, a string for the body, and an integer for the HTTP code. BleResponse is then sent to the client.

4.4.2 Sending Response

Similar to sending a request, we use the same packet format with the two bytes. BleResponse is parsed into a JSON with a similar structure as BleRequest. Instead of three indicies, we have two indicies, one for the response body, and one for the HTTP code. After the packet is formed, BleService writes the data onto the shared characteristic and sends a notifyCharacteristicChanged(). When we send a notifyCharacteristicChanged(), we are given the ability to require confirmation from the client. It is interesting to see here that when the connected device receives the notification, Android will automatically handle the confirmation for us, eliminating the need to send an acknowledgement.

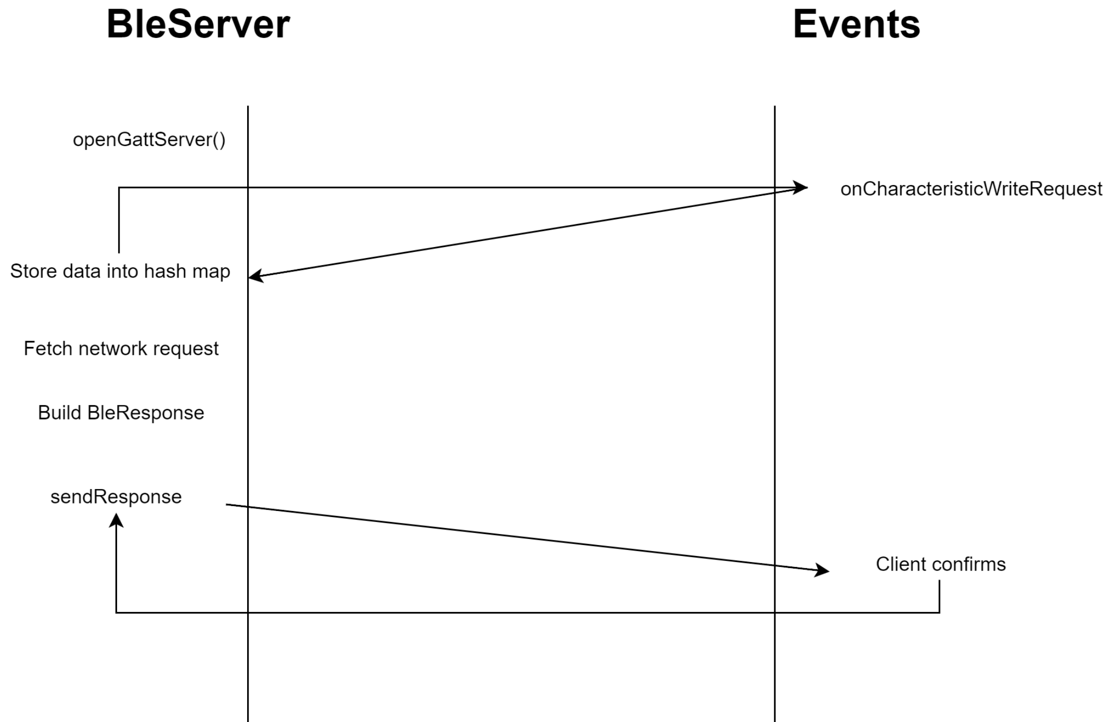


Figure 4.6: BleServer flow.

Recall, BleService implements the OnCharacteristicChanged callback and pushes a ResponseAvailable event. BleClient subscribes to the event and handles the response as described in the earlier section.

Figure 4.6 summarizes what we just covered. Figure 4.7 summarizes both the client and the server flow.

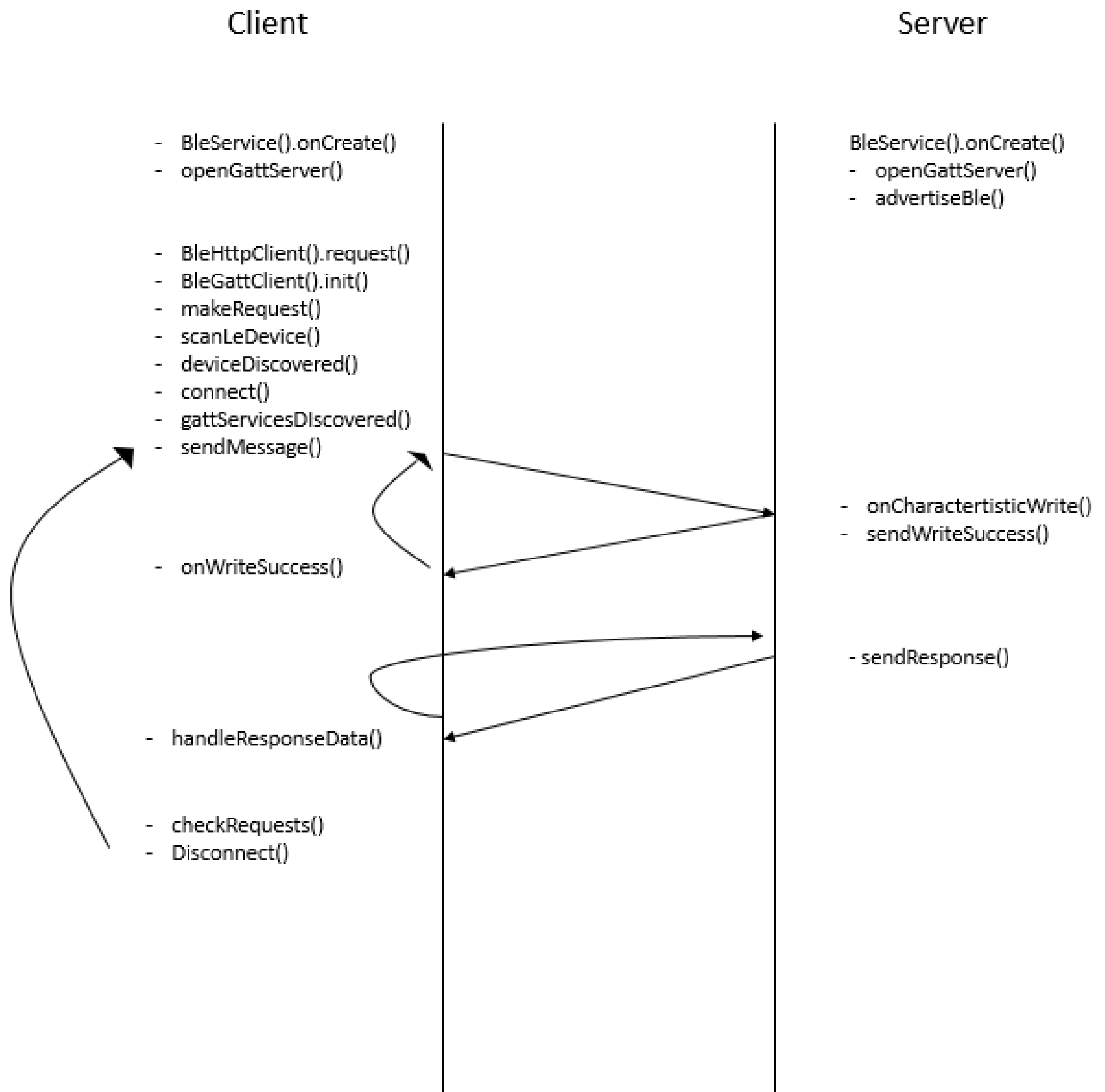


Figure 4.7: BleClient and BleServer flow combined.

Chapter 5

PERFORMANCE EVALUATION

Validation values collecting quantitative data on the success of obtaining connectivity through the connection of two smart phones. As the main goal for this thesis is to obtain connectivity when a phone loses Internet connection, the hope is that connectivity is regained in most circumstances when a nearby phone has an available connection to share.

5.1 Experimental Testbed

Validation will include two parts. The first part will consist of measuring the success of obtaining connectivity between two devices. The second part of the test will measure the effects on each device when running the tool. Battery, speed, and time will be measured to see how well the tool performs.

5.1.1 Testing Connectivity

To measure connectivity, we have created a benchmark to quantify the success of obtaining connectivity. To run the benchmark, a small simple Android application is created that will periodically post data. A small REST service is also created to allow the application to post data. Phone one will be connecting to phone two through various different scenarios to test BleHttp.

This application and REST service will run under a series of different scenarios:

- Phone one has connectivity through Wi-Fi, phone two is not necessary (this is the base line).

- Phone one has no connectivity. Phone two is connected through Wi-Fi.
- Phone one has no connectivity. Phone two is connected through cell tower, is stationary, and has spotty connection.
- Phone one has no connectivity. Phone two is connected through cell tower, is stationary, and has good connection.
- Phone one has no connectivity. Phone two is connected through cell tower, is mobile, and has spotty connection.
- Phone one has no connectivity. Phone two is connected through cell tower, is mobile, and has good connection.

The application will periodically post a time stamp data to the REST service every second. Once the data has been captured, the REST service will then add another time stamp field that will record exactly when the service received the request. By capturing the method the device used to post, we can calculate exactly when the device lost Internet connection and how many times it worked.

This benchmark will run in a simulated environment (stationary, indoors), and in a real-world environment (mobile) by taking a long road trip passing by areas of no connectivity.

5.1.2 Testing Battery

Battery testing is difficult in Android. We will test through power profiles that the Android operating system provides. The Android framework automatically determines battery use statistics by tracking how long device components spend in different states. Additionally, as components change states, such as Bluetooth turning off and on, the service will automatically report to the Android framework's battery

statistics. To use this framework for validation, we simply clear the battery profile before running a test scenario and pull the profile data once a scenario has been tested for a period of time. We can then analyze and compare how many mAhs the application consumed through each scenario.

5.1.3 Testing Speed and Time

To test speed and time, we can leverage the data that we recorded in our REST service - the two timestamps that were recorded into each record in the database. By saving a time stamp from the device and a time stamp when the server sees the request, we can measure how long the total trip time the request took. We can also measure how much impact using a second device takes compared to normal circumstances when the phone has network connection.

5.2 Results

In figure 5.1, we show the average time it takes for BleHttp to complete the three different tasks: searching and connecting to a device, sending the request to the connected device, and receiving the request/response data. This was done by configuring the application and the REST service to send varying amounts of data (3, 50, and 100 packets). Each test was averaged from five runs. From the graph, we can see that sending and receiving packets that are small (three packets), only take up about 200 milliseconds. As we increase the amount of packets we send, the time it takes increases linearly, about 80-90 milliseconds per packet. Sending or receiving 50 packets averages about five seconds and sending or receiving 100 packets takes up to almost ten seconds. These results show that BleHttp is exceptional at sending small chunks of data such as GPS coordinates to the server, and receiving a small response back such as an acknowledgement response. As we start to send more and more data, the

delay starts to become unreasonable.

Notice that sending three packets and receiving two packets together took about 400 milliseconds. For a request of that size, searching and connecting to a device takes up the majority of the round trip time. Recall BleHttp contains a request queue that is checked before disconnecting from a connected device. When requests are queued up, we can effectively eliminate the delay generated from searching and connecting to a device.

In figure 5.2, we show the total average round trip time for requests with different numbers of send and request packets. This included waiting for the HTTP request to finish on the connected device. We started with simply making a request through the phone's Wi-Fi to establish a baseline. Sending and receiving small amounts of data, i.e. three packets, only took about 1.5 seconds, but as we start to send large chunks of data, i.e. 100 packets, we soon reach to about 20 seconds per request.

In figure 5.3, we tested the average round trip time in different types of network conditions as described in the previous section. In this test, we configured the environment to send 25 packets and receive 25 packets, totaling to 50 packets for the entire request. When Phone two was connected to Wi-Fi indoors, the average round trip time (RTT) was about five seconds and had a success rate of 100%. Testing with a mobile connection under spotty and good signals continued to yield a 100% success rate, but average RTT increased. When we switched to a more real-world scenario, we saw similar average RTT's for both spotty and good signals, but the spotty signal saw success rate drop about 20%.

In figure 5.4, we tested battery life of both the client and the server running over a course of five minutes. We set the client to make a network request every second. Results indicate that the client consumes more battery through CPU cycles than Bluetooth cycles compared to the server.

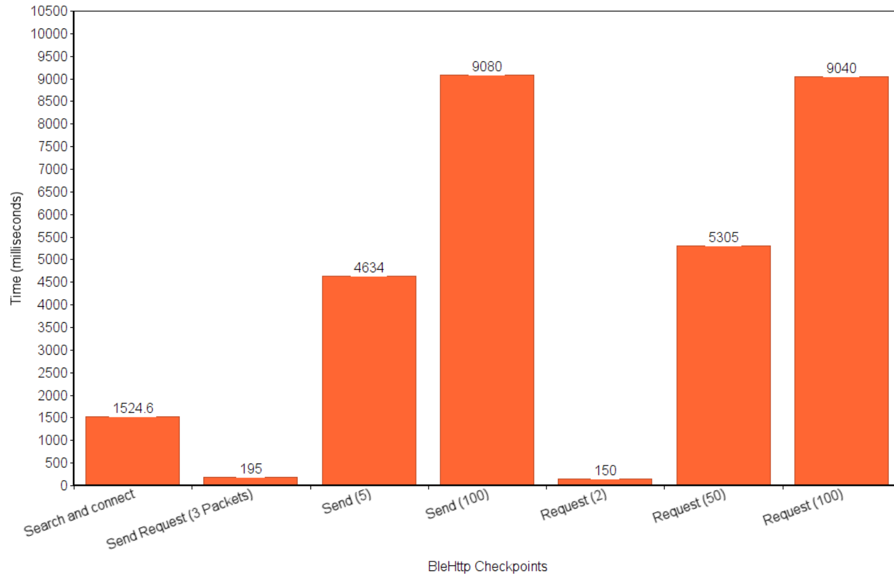


Figure 5.1: Average Time for the BleHttp checkpoints during a request.

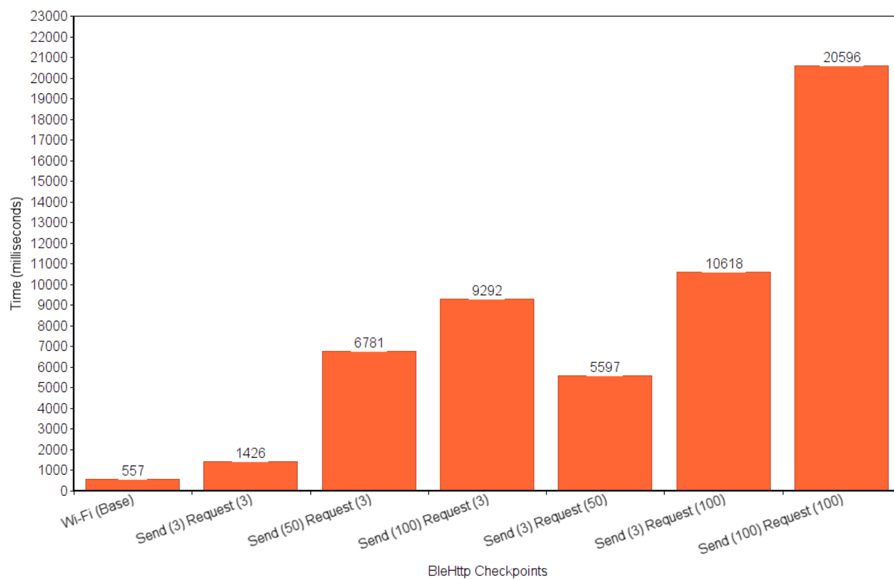


Figure 5.2: Average round trip time's for requests with different numbers of send and request packets.

Network On Server - Signal Strength - Type	Average RTT (Milliseconds)	Success Rate
Wi-Fi - Stationary	5242	100%
Mobile - Spotty - Stationary	19676	100%
Mobile - Good - Stationary	10349	100%
Mobile - Spotty - Driving	17730	80%
Mobile - Good - Driving	11456	100%

Figure 5.3: Average RTT and success rate for different types of network conditions on the server.

Role	Bluetooth	CPU
Server	0.499 mAh	0.689 mAh
Client	0.962 mAh	0.228 mAh

Figure 5.4: Battery life of server and client over a course of five minutes.

Chapter 6

CONCLUSION

My primary goal of this thesis was to implement a solution to obtain network connectivity from nearby peers when a user loses their mobile signal. The technology that I chose to implement BleHttp used Bluetooth Low Energy for the seamless user experience. Bluetooth Low Energy allowed the possibility of discovering, connecting, and sending/receiving data without any user interaction. My results showed that BleHttp is capable of supporting such a system.

BleHttp is exceptional for small requests such as posting GPS coordinates periodically to a REST service. In Figure 5.1, we saw that sending and receiving small amounts of data consecutively can be accomplished in as little as half a second.

BleHttp falters when performing large requests. Delays from five seconds up to ten seconds were seen when attempting to send or receive data of 50-100 packets. When attempting to send and receive 100 packets, we saw delays of up to 20 seconds.

6.1 Future Work

With regards to the BleHttp implementation, several aspects can be improved. To name a few, compressing packets to reduce byte size, connecting and sending requests to multiple devices, security, privacy and user incentives are functions that would benefit BleHttp and improve its robustness.

If BleHttp employed compression and decompression for both requests and responses, we could see a considerable performance gain. Chapter five showed high delays that lead to unreasonable use cases when performing large data requests. Gzip compression format is commonly used in HTTP compression to speed up the sending

of HTML and other content. Additionally, it is considered one of the three standard formats for HTTP compression specified in RFC 2616 [5].

BIBLIOGRAPHY

[1] AppBrain. Most popular google play categories. http://www.appbrain.com/stats/android-market-app-categories, November 2015. [Online; accessed 5-May-2016].

[2] I. Bluetooth SIG. What is bluetooth. http://www.bluetooth.com/what-is-bluetooth-technology/bluetooth, 2015. [Online; accessed 5-May-2016].

[3] D. Bosomworth. Mobile marketing statistics 2015. http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/, July 2015. [Online; accessed 5-May-2016].

[4] M. Constantinescu, E. Onur, Y. Durmus, S. Nikou, M. de Reuver, H. Bouwman, M. Djurica, and P. M. Glatz. Mobile tethering: overview, perspectives and challenges. *info*, 16(3):40–53, 2014.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616, hypertext transfer protocol – http/1.1, 1999.

[6] Google. Android 5.0 apis. http://developer.android.com/about/versions/android-5.0.html. [Online; accessed 5-May-2016].

[7] Google. Bluetooth low energy. https://developer.android.com/guide/topics/connectivity/bluetooth-le.html. [Online; accessed 5-May-2016].

[8] Google. Project fi - frequently asked questions. https://fi.google.com/about/faq/#network-and-coverage-1. [Online; accessed 5-May-2016].

[9] Google. Android history. https://www.android.com/history/#/marshmallow, 2015. [Online; accessed 5-May-2016].

[10] Google. Event bus explained. https://github.com/google/guava/wiki/EventBusExplained, 2015. [Online; accessed 5-May-2016].

[11] S. B. R. Institute. Android phone statistics. http://www.statisticbrain.com/android-phone-statistics/, March 2015. [Online; accessed 5-May-2016].

[12] R. Metrics. First half 2015 mobile network performance in the us. http://www.rootmetrics.com/us/blog/special-reports/2015-1h-national-us, August 2015. [Online; accessed 8-May-2016].

[13] T. S. Portal. Number of apps available in leading app stores as of july 2015. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, July 2015. [Online; accessed 5-May-2016].

[14] I. Research. Smarthone os market share, 2015 q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2015. [Online; accessed 5-May-2016].

[15] A. Smith. U.s. smartphone use in 2015. http://www.pewinternet.org/2015/04/01/us-smartphone-use-in-2015/, April 2015. [Online; accessed 5-May-2016].

[16] J. Thomas and J. Robble. Off grid communications with android. *The MITRE Corporation*, 2012.

[17] K.-K. Yap, T.-Y. Huang, M. Kobayashi, Y. Yiakoumis, N. McKeown, S. Katti, and G. Parulkar. Making use of all the networks around us: A case study in android. *SIGCOMM Comput. Commun. Rev.*, 42(4):455–460, September 2012.

[18] B. R. Zeller. Bandwidth aggregation across multiple smartphone devices. Mas-

ter's thesis, California Polytechnic State University, San Luis Obispo, January 2014.