FOG PROTOCOL AND FOGKIT: A JSON-BASED PROTOCOL AND FRAMEWORK FOR COMMUNICATION BETWEEN BLUETOOTH-ENABLED WEARABLE INTERNET OF THINGS DEVICES

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

Spencer Lewson

June 2015

COMMITTEE MEMBERSHIP

TITLE:	Fog Protocol and FogKit: A JSON-Based Protocol and Framework for Communication Between Bluetooth-Enabled Wearable Internet of Things Devices
AUTHOR:	Spencer Lewson
DATE SUBMITTED:	June 2015
COMMITTEE CHAIR:	Professor John Bellardo, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor David Janzen, Ph.D. Department of Computer Science
COMMITTEE MEMBER:	Professor John Clements, Ph.D. Department of Computer Science

Abstract

Fog Protocol and FogKit: A JSON-Based Protocol and Framework for Communication Between Bluetooth-Enabled Wearable Internet of Things Devices

Spencer Lewson

Advancements in technology have brought about a wide variety of devices, such as embedded devices with sensors and actuators, personal computers, smart devices, and health devices. Many of these devices are categorized as "wearables," meaning that they are intended to be carried and used on one's body. As this category increases in popularity and functionality, developers will need a convenient way for these devices to communicate with each other and store information in a standardized and efficient manner.

The Fog protocol and FogKit framework developed and demonstrated for this thesis address these issues by providing a set of powerful features, including data posting, data querying, event notifications, and network status requests. These features are defined as convenient JSON formatted messages which can be communicated between Bluetooth peripherals using an iOS device running FogKit as router and server.

ACKNOWLEDGMENTS

Thanks to:

- Dr. John Bellardo, my thesis advisor who provided guidance and support throughout the project
- Dr. David Janzen, a member of my committee
- Dr. John Clements, a member of my committee
- Sam, Stephanie, and Michael Lewson, my loving family who offered unending support
- Neil Daniels, a friend who provided assistance with the implementation of iOS technologies
- Michael DeWitt, a friend who provided feedback on the design and implementation of the FogKit and Bluetooth technologies

Contents

List of Figures

Chapter 1

Introduction

The world is currently experiencing an "explosion" of connected devices. Advancements in technology and significant decreases in hardware costs have led to the mass adoption of electronic devices, collectively known as the "Internet of Things" (IoT). These IoT devices, which are constantly sensing and generating data, are all interconnected, allowing them to communicate with each other and make decisions to help improve our lives.

There are many visions of the IoT that range from the massive deployment of thousands of sensors and actuators to a few smart objects worn on one's body that communicate with each other. However, in all of these visions, unique sets of challenges arise that must be overcome. Although the IoT encompasses a wide variety of devices, from laptops and smart phones to embedded chips with sensors and actuators, the majority of them fall in the latter category. This category is characterized by devices that are typically constrained with limited energy and computational power.

One class of device in the IoT is commonly referred to as wearable technology. "Wearables" include devices that are carried on one's body, such as smart phones, smart watches, pedometers, and heart rate monitors. These devices often connect to a smartphone via Bluetooth and gather health information and display data or notifications to the wearer[17]. As this category becomes more popular and the function-

ality of these devices increases in complexity, it is clear that many of these devices will need an improved protocol to communicate with each other and/or store data externally.

Current protocols used in the IoT are designed to face a few key challenges in IoT networks, including efficiency and interoperability. The Internet Engineering Task Force defined the layer 3 and 4 protocols[28], RPL and CoAP, to address these challenges. However, they were designed for massive networks with thousands or millions of nodes[34][42]. These layers and protocol message formats are difficult to work with and have an unnceccessary overhead for small Bluetooth networks used by wearable technology.

The Fog protocol and FogKit framework developed for this thesis attempt to address these shortcomings by defining a convenient method of communication that is designed specifically for wearable devices operating on Bluetooth networks. This was accomplished by reviewing other protocols and technology, learning from their design decisions, and integrating these lessons into the Fog system. The Fog protocol defines a simple set of messages that are sent from device to device. These messages can be of many types, including event messages notifying a device that an event occurred, post messages that include data for the central device to store, query messages that request data, and more. For developer convenience, messages are transferred from the peripheral to the central device in an easy key-value pairing known as the Javascript Object Notation (JSON) form, and then depending on the destination, routed to the designated device or consumed by the central device itself.

The Fog protocol was designed alongside its prototype implementation, FogKit. The purpose of FogKit was to act as a proof-of-concept that guided the direction and features of the Fog protocol. FogKit creates this functionality between an iOS device and multiple Bluetooth devices. In the FogKit model, the iOS device acts as a server that routes communication between the other Bluetooth devices and as an external database to store data from these devices.

The Fog protocol and FogKit framework are named after the fog computing paradigm, in which the cloud computing and service model is brought to the edge of a network[13]. In other words, Fog computing is similar to cloud computing, but focuses on small local networks. One way to imagine this is with a "cloud" that is grabbed and pulled down to surround a person. This "cloud" would then be near the ground, making it "fog."

The rest of this thesis is organized as follows: Chapter 2 gives an overview of the Internet of Things, Bluetooth, and other information that will be helpful in understanding the Fog protocol and FogKit. Chapter 3 explains related technologies in the IoT, including an overview of a general IoT stack and commonly used IoT protocols. Chapter 4 details the Fog protocol itself and explains its design. Chapter 5 gives an overview of the implementation of FogKit and the technologies used within it. Chapter 6 focuses on validation of the protocol through theoretical examples. Finally, chapters 7 and 8 notes what future work should be completed before release and ends with a conclusion.

Chapter 2

Background

2.1 Overview of the Internet of Things

2.1.1 Introduction to the Internet of Things

The Internet of Things (IoT) is an emerging concept that is changing the way we interact between and the real world and virtual world. The IEEE defines the Internet of Things as "a system consisting of networks of sensors, actuators, and smart objects whose purpose is to interconnect 'all' things, including everyday and industrial objects, in such a way as to make them intelligent, programmable, and more capable of interacting with humans and each other[25]." Essentially, the idea of the Internet of Things is to connect all "things" to the internet, allowing them to communicate and make decisions[39] in order to collaborate to perform actions and achieve goals[27].

This concept of interconnected devices can be traced back to work done by Kevin Ashton in 1999[5]. Ashton reportedly coined the term when presenting his linking Radio-Frequency Identification (RFID) technology to track items in a supply chain[5]. The term quickly increased in popularity and is well known as the communication system where the internet is connected to the physical world through ubiquitous deployment of devices[36].

Interest in the Internet of Things is currently exploding as it becomes a topic of research in both academia and industry[36]. Every year, the Gartner company provides a report on the "hype cycle" for emerging technologies. This report acts as a broad aggregate of technologies that are the focus of attention and are believed to have the potential for significant impact. Industry often uses this information to identify technologies that are emerging and are candidates for future development. In 2014, Gartner estimated the Internet of Things to be at the peak of its hype[35]. This suggests an upcoming boom in the IoT and its widespread adoption and implementation.

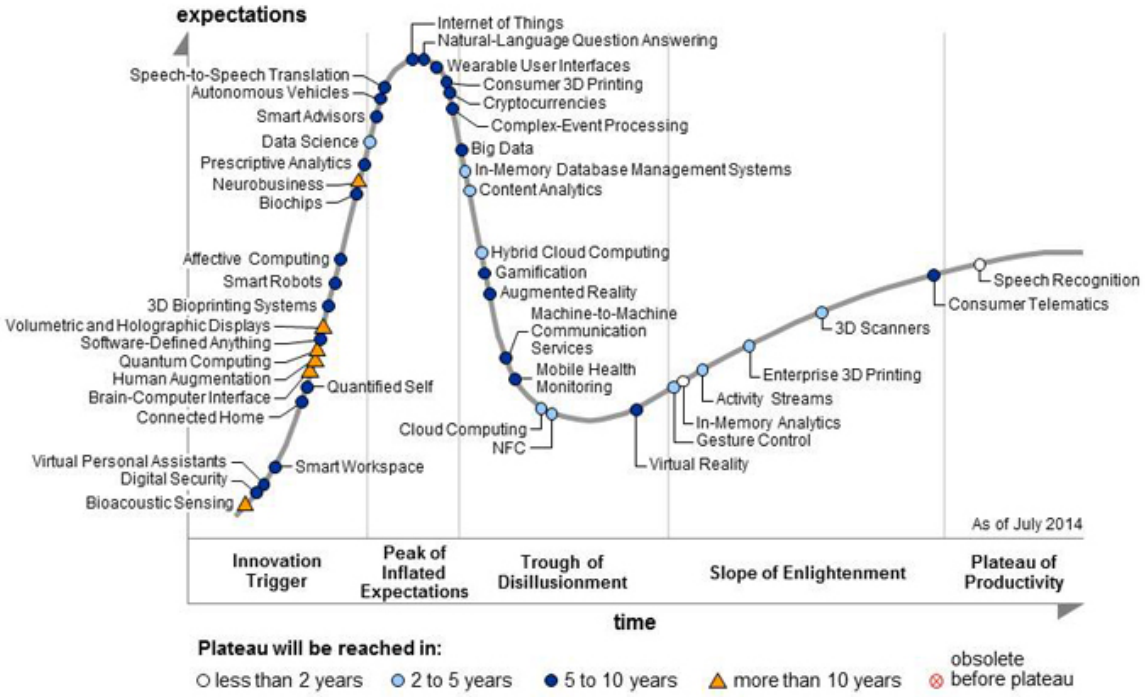

Figure 2.1: The Gartner Hype Cycle chart[35].

This report on hype is supported by predictions from industry. Cisco forecasts that the economic value created by the Internet of Things will be 19 trillion dollars and IDC forecasts that the worldwide market for the IoT will be 7.1 trillion dollars by 2020[33]. These impressive numbers are not surprising, considering Cisco predicts that by that same year, there will be 50 billion connected devices in the world[18].

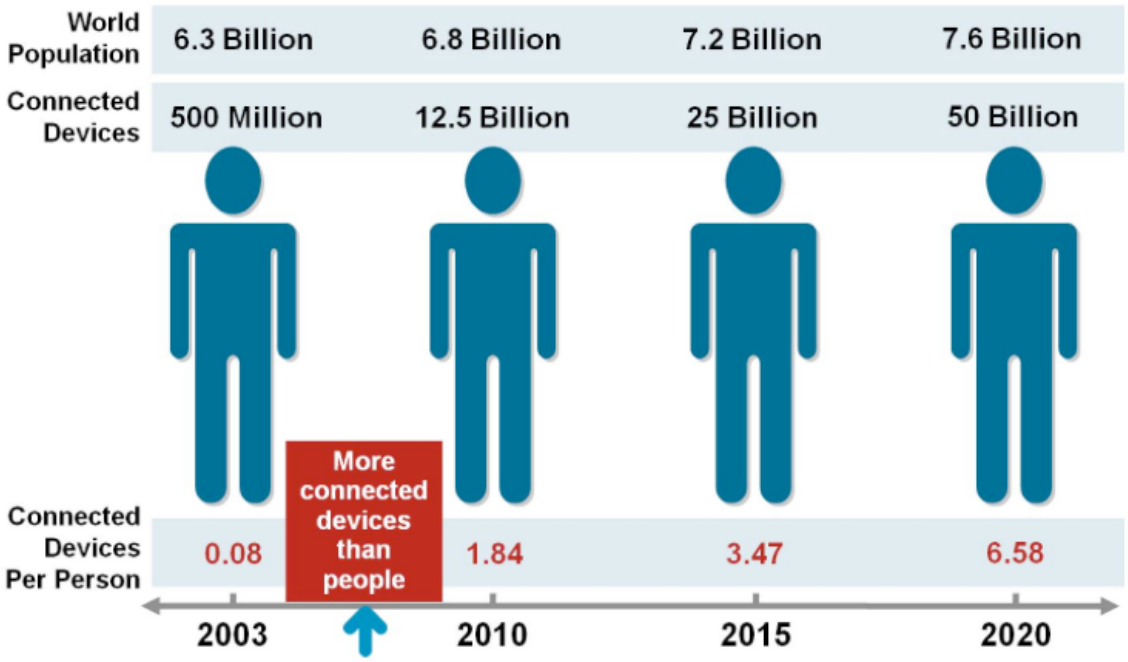

Figure 2.2: Cisco's prediction for number of connected devices by 2020[18].

2.1.2 Visions of the Internet of Things

While it is widely agreed that the Internet of Things consists of a network of interconnected devices, there are many different visions for exactly what these devices are and how they will behave. This ambiguity is likely due to the fact that the phrase "Internet of Things" is syntactically composed of two vague terms, "internet" and "things.[27]" The term "internet" suggests a vision for a large number of interconnected devices while the term "things" focuses on the idea of integrating technology into generic objects[27]. For this reason, differences, sometimes substantial, arise when industry and academia begin researching and implementing systems from an "internet oriented" or "things oriented" perspective for specific interests and backgrounds[27]. These different opinions of the IoT are typically placed into three categories, a "things" oriented vision, an "internet" oriented vision, and a "semantic" oriented vision[6].

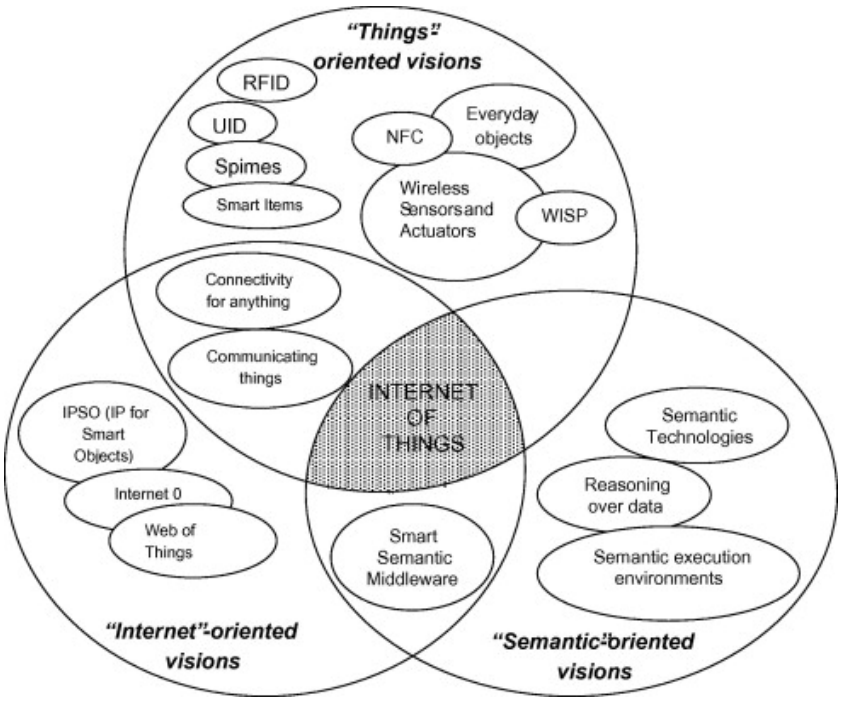

Figure 2.3: The three visions of the Internet of Things[6].

"Things Oriented" Vision

The "things oriented" vision of the IoT was originally devised to improve object traceability by tracking an object's status and location over time, such as the RFID tracking system proposed by Ashton in 1999[27][5]. However, many others have visions that go beyond simple RFID tracking and focus on augmenting the intelligence of things[27]. These "smart items" may not only be equipped with sensors, wireless communication, memory, etcetra, but eventually may have the potential to move into context awareness and collaborative communication[27].

"Internet Oriented" Vision

The "internet oriented" vision of the IoT focuses on the design of protocols for the addressing and routing of internet enabled objects and their communication[27]. The IP for Smart Objects Alliance (IPSO) has a protocol that is aligned with this vision. IPSO is focusing on a protocol that reduces the complexity of the IP architecture for IoT devices.

"Semantic Oriented" Vision

The "semantic oriented" vision of the IoT is focused primarily on the issues related to data[27]. As shown by Cisco's predictions, the number of devices involved in the future internet is predicted to be extremely high[18][27]. This will create issues regarding how to represent, store, interconnect, search, and organize the information generated by the IoT devices[27]. In this situation, semantic technologies could be used to provide appropriate modeling solutions and descriptions for the large amount of data[27].

These three visions, which focus on adding technology to things, facilitating communication between things, and processing the data generated by things, are clearly different. However, each of them are important to the overall vision of the IoT. Researchers believe that the future of the IoT paradigm lies at the center of the diagram, where all these visions converge[27].

2.1.3 Applications of the IoT

The potential applications for the Internet of Things are seemingly endless. While there are currently only a relatively few deployments of IoT technologies, the IoT is expected to be very important in the transportation, medicine, manufacturing, agriculture, and environmental monitoring fields as well as other applications[6]. Listed below are only a few of the possibilites for the Internet of Things:

Transportation

The IoT could be used to improve the safety and reliability of parts in aircraft or automobiles[6]. RFID tags embedded in parts and products could help prove the authenticity of parts and protect against counterfeits[6]. Additionally, sensors and actuators could be used to monitor and report various parameters such as tire pressure or object proximity[6]. Other predictions focus on vehicle-to-vehicle communication, which could be used to increase vehicle safety and improve traffic management[6].

Health Care

In the medical and healthcare industries, the IoT could be used as a platform for monitoring patient health[6]. Implantable devices could be used to store information about a patient's health record that could be used to save his/her life in an emergency situation, or used to track patient's vitals and alert users about irregularities using pattern detection and machine learning[6].

Environmental Monitoring

The IoT is being used to implement "smart cities" where public resources are used more efficiently and the quality of service offered to citizens is improved[43]. Distributed devices are being used to monitor waste management to create more efficient collection routes, air quality to ensure public health, traffic congestion to better deploy officers, lighting to improve efficiency, and more[43].

Home Automation

Wireless home automation is essentially the use of embedded sensors and actuators which are monitored by control applications to increase the homeowner's comfort and maximize the efficiency of energy, water, and other resource usage. Other possible uses in home automation include light controls which turn on automatically when people are present or HVAC systems which can control window blinds and make optimizations for what rooms to heat at what times. There could also be applications in security with glass-break sensors and more[21].

2.2 Wearable Technology

Wearable technology is a category of devices in the Internet of Things that are carried on one's body. These devices are predicted to produce a large amount of innovation and integration in medical lifestyles, fitness, gaming, entertainment, and other consumer applications[41]. Essentially, these devices consist of embedded computing

systems that provide ubiquitous personalized services to the wearer[41]. Wearables are now regarded as one of top categories in the IoT[41]. Since the late 2000's, the wearable category has experienced surprising growth in market adoption[41]. This is likely due to the decrease in cost of embedded electronics and the advent of the smartphone, which provides the necessary data services and user-interaction support[41].

Although the market is relatively new and there is truly a wide spectrum of devices and uses, we believe that the devices emerging can be separated into two main categories based on their functionality as either a simple data-focused device or fully featured system:

- **single purpose** sensors or actuators that track information about the wearer or perform a simple action
- **multifunctional** devices capable of interfacing directly with the wearer and providing sophisticated actions

2.2.1 Single Purpose Devices

The single purpose category consists of sensors or actuators that track information about the wearer or perform simple actions. Although there is a seemingly inexhaustable list of items that fall in this category, a few examples include devices such as heart rate monitors, pedometers, and event triggering devices that track aspects of the wearer or offer him/her a convenience.

Fitness and other health related trackers make up a large portion of this category. These devices include products such as the FitBit; which is a miniature fitness tracking device worn on the body[19]. These products, although variable in the exact features included, essentially track a user's steps, sleep times, heart rate, and more[19].

Another example of a device that falls in this category is a product in development by Cal Poly, San Luis Obispo students: the DingBot[16]. The DingBot is a Bluetooth enabled keychain with a button for triggering actions on a user's smart phone. The

device can be configured to do almost anything on your smartphone, such as launch apps, control alarms and timers, or through the use of additional services, unlock doors, turn on lights, and more[16].

2.2.2 Multifunctional Devices

Unlike single purpose devices, multifunctional devices are capable of gathering and processing a variety of types of data, presenting it to a user and/or transferring it to other devices, running local applications, and more. Examples of these devices include smart watches, smart glasses, and more.

Smart watches are becoming increasingly popular as almost every major technology company, including Apple, Google, Samsung, Sony, Motorola, and more, have launched products into the market[38]. These wearable devices offer a wide variety of features to users from providing notifications to running full applications. The Apple Watch, for example, offers functionality for viewing notifications, taking phone calls, tracking health and fitness, making digital payments, and much more through the development of 3rd party applications[3].

Smart glasses, such as the Google Glass, are another form of multifunctional wearable technology. Google markets the Glass as a device that is "There when you need it. Out of the way when you don't[23]." This pair of "smart glasses" has a small transparent screen on which an image is projected[23]. Essentially, the device is similar in functionality to the smart watches, but takes a different form factor. It also allows users to view notifications, search the internet, send messages, and much more through the development of 3rd party applications.[23].

2.2.3 Body Area Networks

These single purpose and multifunctional wearable devices explained above are connected using a Body Area Network (BAN), or a low power wireless communication standard[32]. There are many existing technologies, such as Bluetooth, ZigBee, ANT,

Sensium, and Zarlink for use for BANs. Each of these technologies have their advantages and disadvantages which are outlined in the figures and sections below.

Technology	Suitability for BAN	
	Merits	Demerits
Bluetooth classic	Established standard, widespread adoption in cell phones and laptops, health device profile defined, sufficient data rate, low cost	Higher power, limited scalability, limited QoS, coexistence with ISM band technologies, limited security, on-body only
Bluetooth Low Energy	Interoperable with Bluetooth, lower power than Bluetooth, leverage Bluetooth brand	Compatibility requirements limit design freedom, limited scalability, limited QoS, coexistence with ISM band technologies, on-body only
ZigBee	Emerging standard, healthcare profile defined, lower power than Bluetooth, scalable, smaller memory footprint	Low data rate, limited QoS, coexistence with ISM band technologies, on-body only
ANT	Simple protocol, low power, healthcare device profiles defined, smaller footprint	Proprietary, limited throughput, limited QoS, coexistence with ISM band technologies, general-purpose design, on-body only
Sensium	Ultra-low-power, custom designed for BANs	Proprietary, low data rate, limited QoS, coexistence with ISM band technologies
Zarlink ZL70101	Ultra-low power, MedRadio compliant, custom designed for implants	Proprietary, implants only

Figure 2.4: The merits and demerits of different BAN technologies.[32]

Technology	Spectrum	Modulation	Channels	Data rate	Operating space	Peak power	nJ/b	Topology	Join time
Bluetooth classic	2.4 GHz	GFSK	79	1–3 Mb/s	1–10 m on-body only	~45mA @3.3V	50	Scatternet	~3 s
Bluetooth Low Energy	2.4 GHz	GFSK	3	1 Mb/s	1–10 m on-body only	~28mA @3.3V	92	Piconet Star	<100 ms
ZigBee	2.4 GHz	O-QPSK	16	250 kb/s	10–100 m on-body only	~16.5m A@1.8V	119	Star, Mesh	30 ms
ANT	2.4 GHz	GFSK	125	1 Mb/s	10–30 m on-body only	~22mA @3.3V	73	Star, tree, or Mesh	
Sensium	868 MHz 915 MHz	BFSK	16	50 kb/s	1–5 m on-body only	~3mA @1.2V	72	Star	< 3 s
Zarlink ZL70101	402–405 MHz 433–434 MHz	2FSK/4FSK	10 MedRadio, 2 ISM	200–800 kb/s	2 m in-body only	~5mA @3.3V	21	P2P	< 2 s

Figure 2.5: The characteristics of different BAN technologies.[32]

Bluetooth Classic and Bluetooth 4.0 LE

The Bluetooth interface defines a link and application layer to create a short range wireless communication standard that supports data transfer and voice applications[32].

There are two significant versions of Bluetooth: Classic and Low Energy (LE). Bluetooth Classic has been succeeded by Bluetooth 4.0 LE, which is power optimized and offers new features[32]. Further details on Bluetooth can be found in section 2.3.

ZigBee

ZigBee defines a network, security, and application layer protocol to create a flexible network[32]. ZigBee is optimized for low-duty-cycle devices, or devices that have the radio powered off most of the time[32]. This differs from Bluetooth, which requires devices to remain synchronized with each other, resulting in increased radio usage and higher power consumption[32].

ANT

ANT is a wireless communication technology designed for general-purpose sensor network applications[32]. ANT features a simple design, low latency, and ability to trade off data rate for power efficiency[32].

Sensium and Zarlink

Sensium and Zarlink are proprietary ultra-low-power transceiver platforms for the healthcare and medical industries[32]. Sensium focuses on creating a simple low-power network while Zarlink focuses on error detection and reliability[32].

2.3 Overview of Bluetooth

2.3.1 About Bluetooth

Bluetooth is a wireless personal area network technology designed by the Bluetooth Special Interest Group. The technology is geared towards voice and data applications and is commonly found in devices such as smart phones, computers, and more. The

release of Bluetooth 4.0 Low Energy, marketed as "Bluetooth Smart" by the Bluetooth SIG, focuses on reducing power consumption and range and increasing data rates[9].

Bluetooth LE focuses on enhancing Internet of Things devices, such as watches, toys, pedometers, and glucose monitors. Simple devices such as pedometers take advantage of the power efficiency, while more complex devices, such as smart watches utilize the improved speed[9].

Bluetooth LE is a complicated technology, so there are many frameworks that abstract away the details and provide a convenient API for developer use. One such framework is Apple's Core Bluetooth framework, which allows developers to interface with the Bluetooth technology available on iOS devices. This Core Bluetooth framework is based on the Bluetooth 4.0 specification, which defines the set of protocols for communication between Bluetooth low energy (Bluetooth LE) devices. The framework abstracts many of the low-level implementation details, allowing engineers to more easily develop apps that interact with other devices through Bluetooth[1].

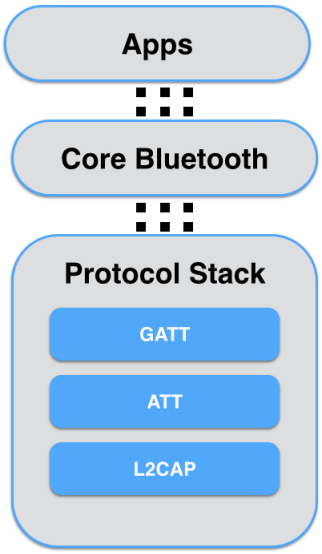

Figure 2.6: The Core Bluetooth Stack[1] .

2.3.2 Bluetooth Roles

In Bluetooth LE connections, there are two key roles: the central and the peripheral. These roles may be likened to that of client and server, where the central acts as the client and the peripheral acts as the server. The central device typically uses information gathered from peripheral devices to accomplish a task[1]. For example, an iPhone central may gather heart rate information from a Bluetooth Pulse Monitor peripheral. Each of these roles have an assigned set of tasks, which are outlined below:

Centrals: Centrals are responsible for scanning for advertising peripherals and connecting to devices in which they are interested. When a central discovers a device, it can request a connection and begin to explore all of its data and interact with all of its features. A central is capable of both reading from and writing to a peripheral device.[1]

Peripherals: Peripherals advertise their presence by broadcasting information in the form of *advertising packets*. These packets contain brief descriptive information on what the device has to offer, such as the peripheral's name and primary function. Once a central connects to a peripheral, the peripheral may announce secondary data and functionality. While the devices are connected, the peripheral is responsible for responding to the central device's read and write requests[1].

2.3.3 Bluetooth Data Representation

When centrals interface with peripherals, they are using the peripheral's Bluetooth 4.0 LE Generic Attribute Profile (GATT). Bluetooth 4.0 Low Energy uses a GATT profile to define how a device works for a specific application[22]. This includes the establishment of common operations and a framework for data transfer and storage as defined by the Attribute Protocol (ATT)[37]. In the traditional client/server model, the GATT server defines the format of the data transported over ATT and accepts requests, commands, and confirmations from a GATT client[37]. It follows then that the GATT server sends responses to requests and notifications of specified events to a GATT client[37].

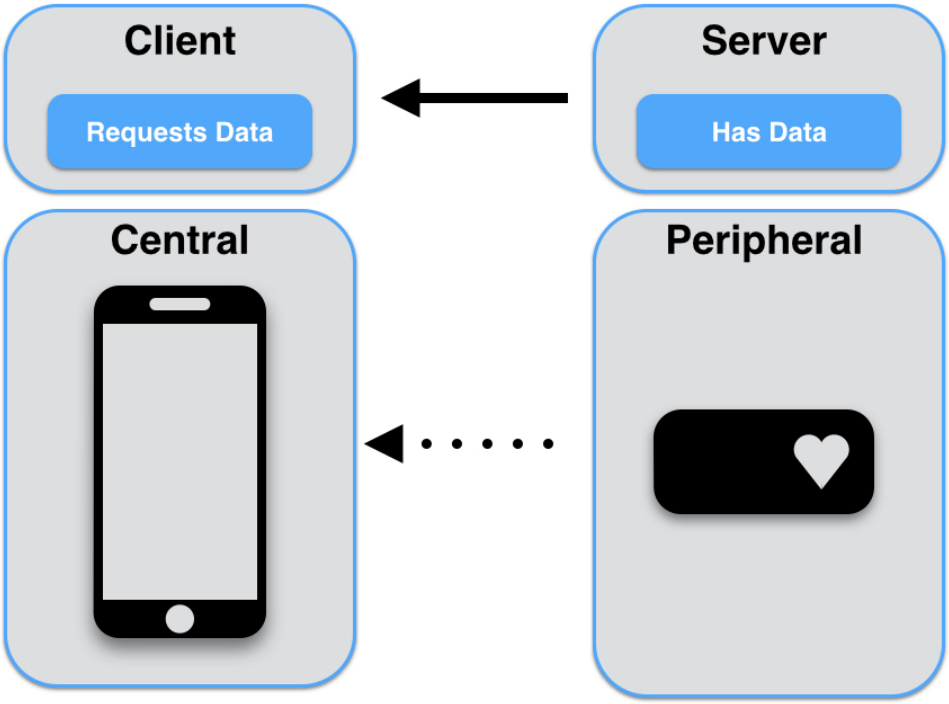

Figure 2.7: The Bluetooth Central and Peripheral[1].

The Attribute Protocol (ATT) defines the capabilities of a Bluetooth 4.0 LE peripheral, including data values and descriptions[37]. These *attributes* are defined as services, characteristics, and descriptors[37].

Service: Services represent a peripheral's capabilities–that is to say, the associated behaviors and features that the peripheral offers to centrals. These services can be denoted as either *pimary*, which indicates that the service serves as the primary function of the device, or *secondary*, which denotes that the service is auxilary and is referenced from at least one primary service.[37].

Characteristic: Characteristics represent detailed information about a peripheral's services. Specifically, a characteristic contains a declaration, property, and single

value that can be read from or written to by a central[37]. Additionally, a characteristic may contain a description known as a *descriptor*.[1].

Descriptor: Descriptors provide detailed information about a characteristic's value, such as the format of the data, whether the value is configured by the central or peripheral, or whether or not the value has changed. These descriptions are often presented in human-readable form.[1]

The following figure is an example of a very simple GATT profile for a Bluetooth Heart Monitor. The device contains a service that indicates it's capable of sensing a heartrate and defines a characteristic from which a central may read that value. The characteristic's descriptor describes the unit of that value to be in "Beats/Min" or "beats per minute."

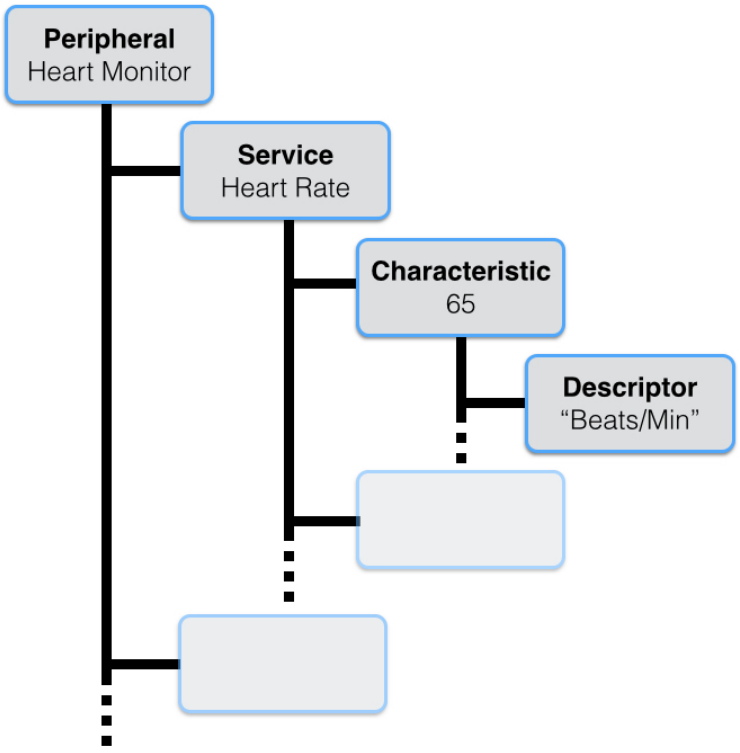

Figure 2.8: The Bluetooth GATT Profile.

2.3.4 Bluetooth Data Transfer

Although Bluetooth 4.0 LE is a newer standard, data transfer is still relatively slow when compared to other technologies. Bluetooth 4.0 specifies a packet size of only 27 octets which are transferred at a rate of 1 Mbps[10].

Data is transferred through a device's characteristics. When a device attempts to transfer data, the device will loop through the data, one packet at a time, and set the characteristic's value. The receiving device will then read and buffer the data every time the value changes.

2.4 OSI 7 Layer Network Model

The Open Systems Interconnect (OSI) model defines a network stack composed of 7 layers which build upon each other, abstracting away control and implementation details[7].

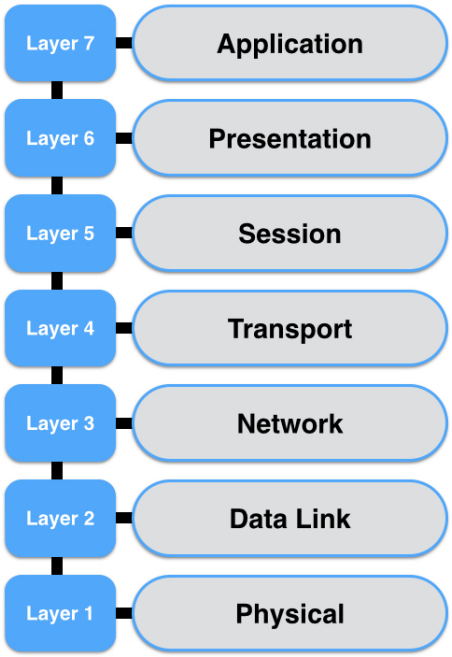

Figure 2.9: The OSI 7 Layer Model[7].

1. Physical Layer

The physical layer is the lowest layer of the OSI model. It is responsible for the transmission and reception of a raw bit stream over a physical medium, such as a cable or wireless transceiver[28]. Data is transmitted as electrical or optical signals, as determined by the physical medium used by the device[28].

2. Data Link Layer

The data link layer is responsible for error-free transmission from one network node to another network node over the network layer[28]. This allows the higher levels to assume the transmission is error free[28]. This is accomplished by managing the link establishment and termination, controlling the frame rate, and reporting network confirmations and errors[28].

3. Network Layer

The network layer decides which physical paths across a network transmitted data should take based on the network topology and conditions[28]. Essentially, it's responsible for routing data through the network and managing data transmission speeds and reliability[28].

4. Transport Layer

The transport layer ensures that messages, or pieces of data, are delivered in the correct order and without errors across different networks or end systems[7]. The transport layer accomplishes this by splitting and reassembling long messages, reporting acknowledgements of receipt of data and errors, and controlling the speed at which data is transmitted[28].

5. Session Layer

The session layer manages the establishment of sessions (similar to a "conversation") between processes running on different devices[7]. It is responsible for the establishment, maintenance, and termination of sessions, and performs the functions necessary to allow processes to communicate over a network[28].

6. Presentation Layer

The presentation layer translates the data from a network format to the application layer format, or vice-versa, depending on if the device is sending or receiving data[28]. The layer is also responsible for data compression and encryption[28].

7. Application Layer

The application layer is the layer at which the user or process operates[28]. Examples of the various functions that operate at this layer include e-mail, web browsing, and remote file access.

Chapter 3

Related Technologies

In 2008, the U.S. National Intelligence Council predicted that by 2025, internet nodes may reside in everyday objects, such as food packages, furniture, documents, and more.[31]. What "things" are produced in the future cannot truly be predicted, but we do know that with the price of wireless sensors and other technologies diminishing rapidly, there will be a boom in large autonomous networks of technology[11]. However, there are many challenges faced with these large networks, including the diversity of data sources and communication protocols[11]. Experts predict that coordinating the communication and maintaining the interoperability of all these devices is a challenging prospect. This is still an area of significant research today[11].

The following sections discuss some of the technologies and features used in the Internet of Things that were influential in the design and implementation of this thesis project. An understanding of these technologies is necessary in order to fully understand the design choices made in the Fog protocol and FogKit framework.

3.1 IoT Stack

In order to accommodate the massive numbers of heterogeneous devices, the IoT stack is organized into a set of layers. These layers are described as follows:

1. **Edge technology layer** consisting of all of the deployed sensors, actuators, and other embedded devices[6]

2. **Access gateway layer** responsible for abstracting communication and data handling of the devices[6]

3. **Internet layer** which facilitates communication between the gateway and other layers[6]

4. **Middleware layer**, which acts as an interface for device and data management[6]

5. **Application layer**, which includes any and all applications that use or manipulate the data[6]

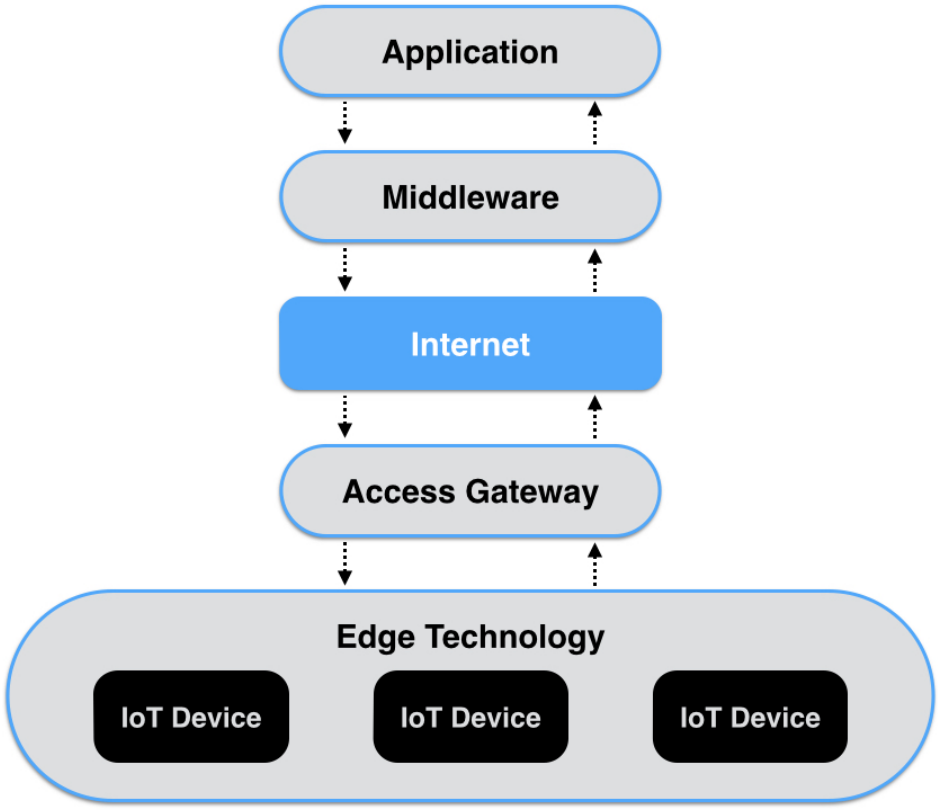

Figure 3.1: An Internet of Things Stack[6].

3.2 Edge Technology Layer

The Edge Technology Layer consists of the distributed sensor networks and the embedded systems, sensors, actuators, and more that make up Internet of Things networks[6]. As part of the nature of being an embedded system, these devices generally share common features, such as constrained energy resources, limited processing capabilities, unreliable connections, and limited direct human contact[6]. For these reasons, they typically are just capable of reading sensors and reporting values to the middleware layer via a network connection. However, although these devices share common features, there is a huge diversity in the implementation of these features, such as the hardware capabilities, networking protocols, and data representation.

3.3 Access Gateway Layer

In the IoT stack, the access gateway layer is responsible for coordinating access from the middleware layer to the embedded devices[6]. Essentially, this layer abstracts the different communication protocols and allows for a uniform interface to read the values from the sensors. Additionally, the access gateway layer can cache the data from the sensors, improving the efficiency of the overall system.

3.4 Middleware Layer

The middleware layer is one of the most critical layers in the Internet of Things stack as it acts as the interface between the hardware layers and the application layers[6]. Essentially, the middleware is responsible for device and data management through the use of the access gateway layer. It abstracts away the distributed aspects of the IoT, such as data filtering, data aggregation, and device acccess control, and presents a convenient interface for the application layer[6]. Like the access gateway layer, there are many technical difficulties presented at the middleware layer. Even when the challenges of dealing with the diversity in devices and protocols is dealt with, there is still the issue of a consistent sensor and data representation[11].

XGSN is one such solution that addresses these problems by abstracting away the hardware layers and providing consistent annotations for the descriptions of the devices themselves and the data they produce[11]. This consistency is accomplished through the use of a standardized vocabulary that ensures interoperabilty between IoT systems[11]. In XGSN, sensor data and observational data are annotated using the World Wide Web Consortium (W3C) Semantic Sensor Network (SSN) ontology[11]. This ontology defines standardized representations of data that makes it easier for other services and applications to share, discover, and integrate sensor information[24]. The two main types of semantic annotation in XGSN are metadata annotations, related to sensors, and observational annotations, which includes a description of the observed values.

3.4.1 Sensor Metadata Annotations

Sensor metadata annotations are focused on sensors, sensing devices, and the device capabilities[11]. This metadata is formated as an XML document that includes information of interest such as sensor information, location, pull rates, security parameters, and more[11].

Presented below is an excerpt of an XML virtual sensor configuration in XGSN from [11] that includes metadata information for a sensor, such as the capabilities it provides (included in the field name elements) and how to access those properties (included in the query elements).

```
<virtual-sensor name="sens1" priority="10">
  <processing-class>
    <class-name>org.openiot.gsn.vsensor.LSMExporter</class-name>
    ...
    <output-structure>
      <field name="temperature" type="double" />
      <field name="humidity" type="double" />
    </output-structure>
  </processing-class>
```

```
  <streams>
    <stream name="input1">
      <source alias="source1" sampling-rate="1" storage-size="1">
        <address wrapper="csv">
        ...
        </address>
        <query>select * from wrapper</query>
      </source>
      <query>select temp as temperature, humid as humidity, timed
        from source1</query>
    </stream>
  </streams>
</virtual-sensor>
```

3.4.2 Observational Value Annotations

Observational value annotations are focused on the observations and measurements produced by the Internet of Things sensors[11]. These annotations could include data on when the observation happened and the context of the observation, in addition to the values and units of the measurements[11].

XGSN is capable of handling the data acquisition process for a wide number of devices and protocols[11]. In XGSN, the server stores information about the connected sensors, including the properties they have and the URL's from which they can be read. Essentially, the server stores information about a resource and requests data from the sensor. The sensor then responds with raw, unformatted data and the XGSN system uses the annotations to store the received data correctly.

3.5 Application Layer

The application layer gathers information from service layers and uses the data to generate information. It may be as complex as a system that manages and analyzes huge amounts of data or it may be simple as a website that displays graphs or other

information. One such application layer is OpenIoT, an open source infrastructure for implementing and integrating Internet of Things devices[30]. Essentially, the platform allows users to dynamically aggregate and compose IoT services following a cloud or utility based model, and then use the raw data to generate high level human-readable information[30].

OpenIoT is capable of collecting and processing data from any type of sensor that conforms to the XGSN and W3C Semantic Sensor Networks (SSN) specifications[30]. It can dynamically discover and query service layers to obtain data, and then use the data to visualize information in the form of charts, graphs, maps, and more[30]. Shown below is an example of this data visualization in OpenIoT.

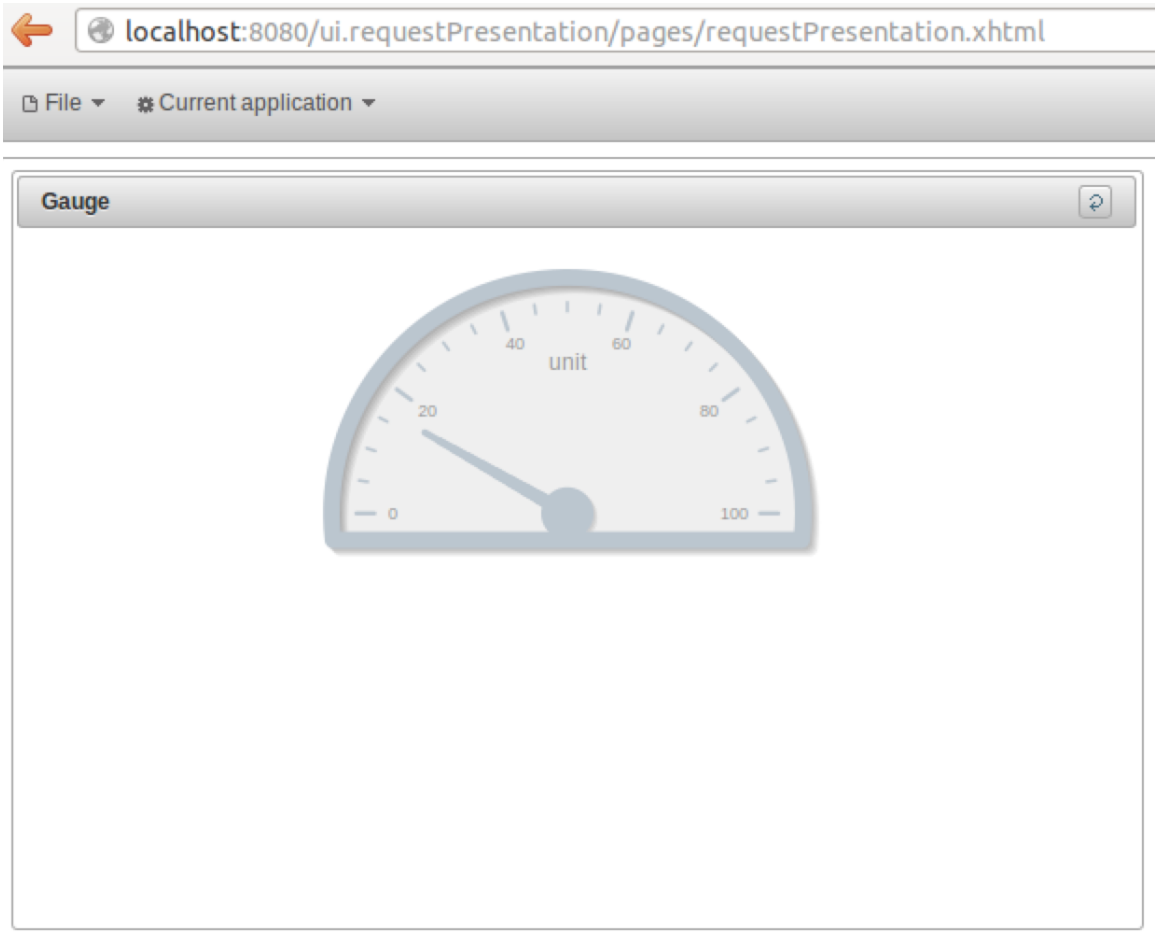

Figure 3.2: A Screenshot of an OpenIoT Visualization.

3.6 Protocols

Due to the restrictions imposed by the constrained and unreliable nature of the networks used by the Internet of Things, the existing standard client/server protocols are not always suitable[31]. They are not optimized for low-power communication due to verbose meta-data and headers and optimization for packet acknowledgement at higher layers[31]. Additionally, in IoT networks, both devices and the middleware can act as a client and/or server, so the ability to accomodate both roles is important[31]. There are many exisitng protocols that devices can use to publish data to the web, however this section will only review the established standards from the Internet Engineering Task Force.

3.6.1 Routing Protocol for Low-Power and Lossy Networks

The networks on which the edge layer technology operates are often unstable and unreliable[31]. In response to this, IETF Routing over Lossy and Low-Power Networks (RoLL) working group was established in 2008 to determine the application scenarios and routing requirements for IoT devices, evaluate the shortcomings of the current protocols, and create a new set of IoT optimized protocols[36]. The RoLL working group eventually published RFC 6550 which defined the IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL)[34]. This level 3 network layer protocol is able to accommodate the Low-Power and Lossy Networks (LLN) and the constrained network devices and routers that operate on them[34]. Additionally, in compliance with the OSI model, RPL is not dependent on any layer 2 link-layer technology, allowing it to operate over any link layers, including constrained or potentially lossy networks[34].

RPL was designed to support three kinds of traffic flow:

1. **Point-to-point**, which is traffic between devices within the LLN[36]
2. **Point-to-multipoint**, which is traffic from a central point to a subset of devices within the LLN [36]
3. **Multipoint-to-point**, which is traffic from devices within the LLN to a central

control point[36]

RPL routes information over a dynamically formed network topology[34]. RPL uses a distance-vector routing protocol, meaning that the nodes determine how to route their traffic by generating an acyclic graph[36]. Essentially, each node knows the distance or cost to every other node to which it is directly connected. This information is shared with the other nodes and is used to determine the shortest paths through the network[40]. The route is completed by exchanging distance vectors with a controller node, which determines the optimum path to route network traffic[40].

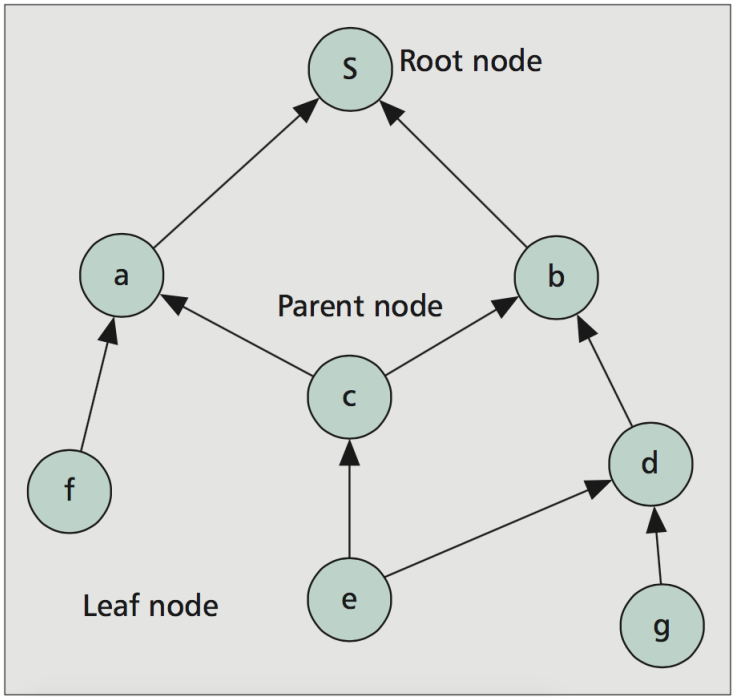

Figure 3.3: An example RPL routing tree[36].

Routing is an extremely important aspect of creating an efficient end-to-end networking service, especially in IoT networks[36]. RPL aims to support ubiquitous sensing applications in large scale and low-power, however, this goal has not yet been met[36]. According to the research paper [36], there are a few challenges with RPL that need to be worked out[36].

1. **End-to-end throughput:** RPL faces throughput challenges when multiple applications coexist on the same physical network. A queue-aware back-pressure

routing algorithm is suggested to send packets through all possible end-to-end paths[36].

2. **Packet reordering:** RPL supports multipath routing solutions, however, the multi-path routing structure used can result in packets being received in the incorrect order. This is a problem that should be addressed[36].

3. **Dynamic duty cycling:** The current RPL design does not accomodate dynamic duty cycling, which could improve the performance of the end-to-end throughput, latency, and more[36].

4. **Multi-topology routing vs. traffic diversity:** RPL utilizes a multi-topology routing approach to create new routes to accomodate each type of traffic currently on the network. However, the cost of constructing and maintaining these routes as the number of applications increases introduces priority and fairness issues that negatively impact performance[36].

3.6.2 Constrained Application Protocol

The Constrained Application Protocol (CoAP), according to RFC 7252, is "a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks[42]." The Internet Engineering Task Force (IETF) RESTful Environments workgroup designed the protocol to accodmodate 8-bit microcontrollers with small amounts of ROM and RAM and to meet the specialized requirements of multicast support, low overhead, and simplicity for constrained environments[42]. The main features of CoAP include:

1. **Constrained web protocol** which is optimized for efficient data transfer and power usage for machine-to-machine communication[31]

2. **Stateless HTTP mapping** through the use of proxies and interfaces, which allows the protocol to easily be used with HTTP applications[31]

3. **UDP transport** that reliably sends messages to one or more devices through unicasts and multicasts[31]

4. **Asynchronous message exchanges,** which allow for the request/response paradigm[31]
5. **Low header overhead and parsing complexity,** which allow constrained devices to handle the protocol[31]
6. **URI and content-type support,** which allow applications to easily address resources[31]
7. **Simple proxy and caching capabilities,** which reduce response time and network usage[31]
8. **Optional resource discovery,** which provides URIs for offered resources[31]
9. **Dual-role endpoints** which are capable of acting as both server and client[31]

Interaction Model

The interaction model of CoAP is comparable to the client/server model in HTTP, as CoAP can easily be translated into the HTTP format for integration with the existing web[42]. However, CoAP endpoints do not assume such a clear and straightforward roles as they do in HTTP[31]. The machine-to-machine interactions typically result in CoAP nodes taking on both server and client roles[42]. In this situation, a CoAP request is similar to that of an HTTP request: a client requests an action using a method code on a resource identifier by a URI on the server[42]. The server then responds with the correct data and response code[42].

However, CoAP differs from HTTP by dealing with these interchanges asynchronously over a datagram-oriented transport, such as UDP[42]. This is accomplished by using a messaging layer that supports optional reliability[42]. CoAP defines four main types of messages, which includes but are not limited to:

1. **Confirmable,** indicating that a response or ACK is expected[31]
2. **Non-confirmable,** indicating that a response or ACK is not expected[31]
3. **Acknowledgement,** indicates that a message has been received[31]

4. **Reset,** indicating that an expected message has not been received[31]

Whether or not the message is a request or response is determined by the method codes and response codes included in the message[42]. The RFC notes that the basic exchanges of these message are "somewhat orthogonal" to the request/response interaction model. Messages can be sent multicast and can be carried in confirmable, non-confirmable messages while responses can be carried in confirmable, non-confirmable, and piggybacked acknowledgement messages[42].

Messaging and Request/Response Models

Logically, CoAP uses a two-layer model to accomodate UDP and the asynchronous nature of interactions[42]. These two layers include a messaging layer and a request/response layer[42].

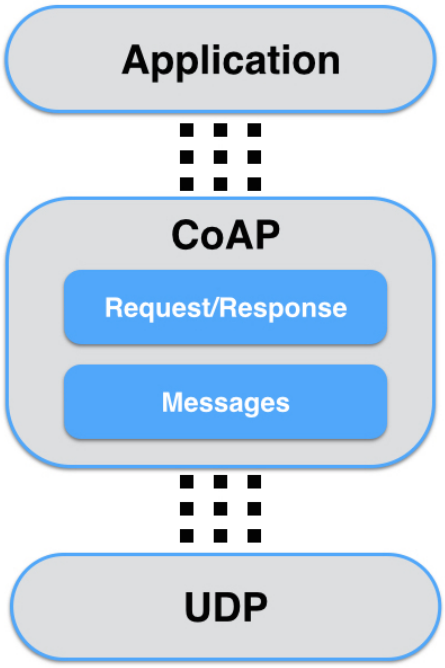

Figure 3.4: The Layering of CoAP[42].

The CoAP messaging model focuses on exchanging messages between endpoints[42]. These messages include a short four byte header which may be followed by options

and a payload. Included in this header is a message ID which is used to detect duplicates and to increase reliability for confirmable messages[42].

When a message is marked as confirmable it indicates that it should be sent using reliable transmission[42]. Reliable messages are sent repeatedly (with a back-off) until an acknowledgement message is received for the correct message id[42]. In the event of an error, a reset message is sent instead of an acknowledgement[42]. On the other hand, messages that do not require reliable transmission are marked as non-confirmable messages[42]. Although these messages are not acknowledged, a message id is still included to detect duplicates and to send reset messages for any errors that may have occurred[42].

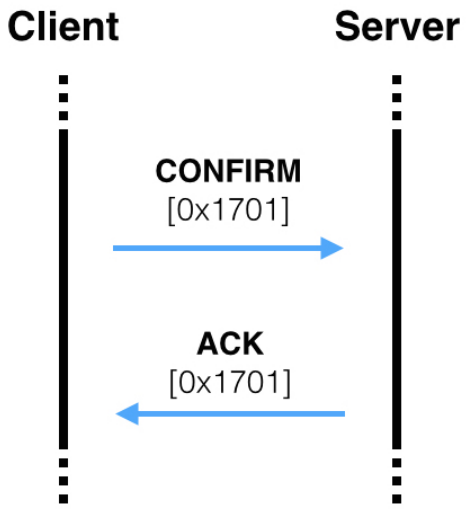

Figure 3.5: Example of reliable transmission[42].

On the other hand, the request/response model builds on the messaging model to offer the ability to send requests and responses for data using method codes and response codes.[42]. Requests, which can be sent by either confirmable or non-confirmable messages, can include a Universal Resource Identifier (URI) to specify what information is needed[42]. Responses or errors are then sent with the relevant data, and if necessary, an ACK[42]. This combined response and acknowledgement is known as a piggybacked response[42].

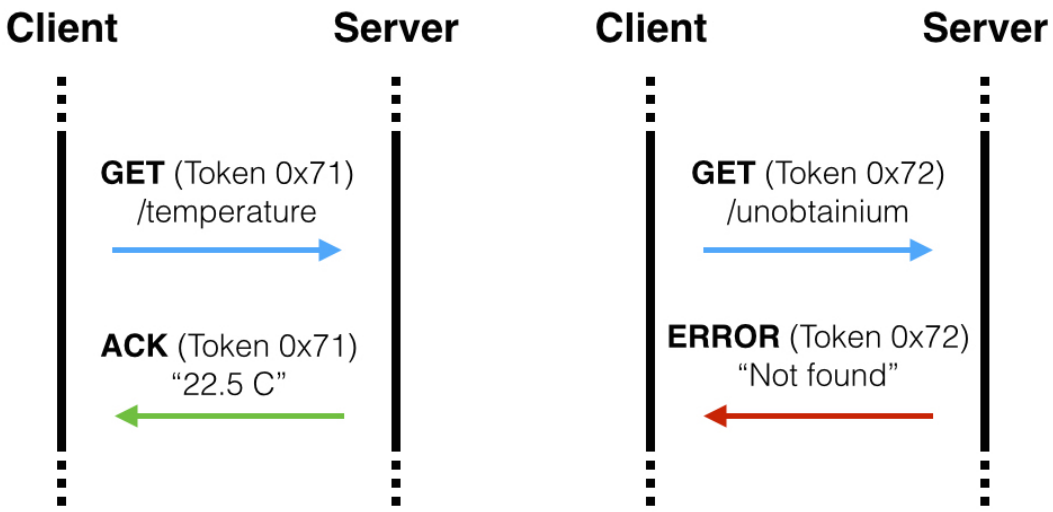

Figure 3.6: Example of the requests/response model in CoAP[42].

Similar to HTTP, CoAP uses GET, PUT, POST, and DELETE methods to indicate the type of request. These methods, which are responsible for requesting data, updating data, posting data, and deleting data, respectively, are not all the methods CoAP is capable of specifying[42]. Further information on these other methods can be found in RFC 7252[42].

Message Format

CoAP messages are encoded in simple binary and follow a specific format, including a 4 byte header followed by a variable length token value, CoAP options, and the payload[42]. The fields in the header are further defined below:

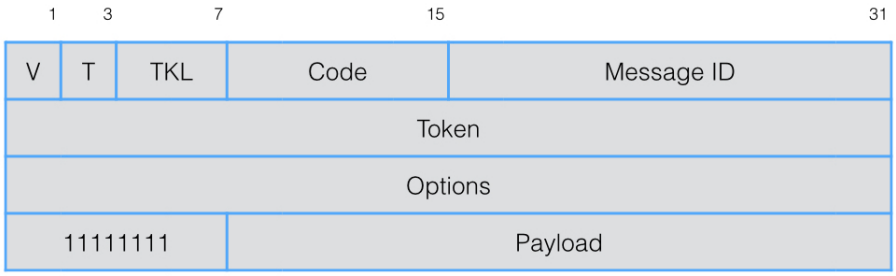

Figure 3.7: The CoAP Message Format.

1. **Version (V):** The version field is a 2-bit unsigned integer that indicates the CoAP version number. This is a required field for every message[42].

2. **Type (T):** The type field is a 2-bit integer that indicates whether the message is confirmable, non-confirmable, acknowledgement, or reset[42]. Further information on these messages can be found in section 3.6.2.

3. **Token Length (TKL):** The token length field is a 4-bit unsigned integer that indicates the length (0-8 bytes) of the token field[42]. Longer lengths (9-15 bytes) are currently reserved and must not be used[42].

4. **Code:** The code field is an 8-bit unsigned integer that indicates whether the message is a success response, client error response, or server error response[42]. Further details on these codes can be found in RFC 7252[42].

5. **Message ID:** The message ID field is a 16-bit unsigned integer that is used to detect message duplication and is used in acknowledgement/reset messages. Further information can be found in section 3.6.2.

6. **Token:** The token field which is 0-8 bytes, depending on the token length field, correlates request and response message[42]. Essentially, this field allows requests and responses to be "matched up."

7. **Options:** The options field is a variable length field used to indicate the selected options for a message[42]. Further information on options can be found in RFC 7252[42].

8. **Payload:** The payload field includes the variable length transmitted data for the request or response message[42].

Chapter 4

Fog Protocol

4.1 Discussion

The Fog protocol defines a set of messages developed for this thesis that can be used to facilitate communication between a Bluetooth central device and connected Bluetooth peripheral devices. While other protocols designed for the IoT focus on efficiency of communication in a massively distributed network with thousands or even millions of devices, the Fog protocol was designed to focus on ease of use and implementation in a very small network with a handful (i.e. 1-8) of devices.

The Fog protocol accomplishes this by condensing the functionality of many protocols into one and formatting the messages in a convenient manner. The OSI network model specifies 7 layers of technologies that build upon each other to offer complete network functionality[28]. In the Fog protocol, the network layer (layer 3), transport layer (layer 4), and application layers (layers 5-7) are combined into a single protocol to reduce the implementation overhead for developers[28].

To further reduce the implementation overhead, the Fog protocol takes a different approach to message and payload formatting. Other networking protocols, such as IP, RPL, HTTP, and CoAP format messages at the byte level. In other words, these

protocols define headers and payloads where attributes are formatted into a specific number of bits or bytes in a certain order. Although this method is very efficient and reduces the amount of data being transmitted, these formats are often difficult to implement and work with. Instead, the Fog protocol uses the Javascript Object Notation, commonly known as JSON, which is a format that is easily read/written by humans and parsed/generated by computers[15]. JSON objects are collections of key-value pairs that are represented as strings[15].

Although the goals and requirements of the Fog protocol are relatively unique, the design of the protocol itself still takes many cues from these other protocols and inspiration from the features of the systems that use them. For example, the routing and messaging aspects of the protocol, including where to send and receive requests and responses are modeled after the RPL and CoAP protocols. On the other hand, the data and feature representations of each device are inspired by the XGSN system and the SSN ontology explained in section 3.4.

The following sections will review the functionality of the Fog protocol and explain the rationale of the design, including information on what protocol or system, if any, the design was inspired by or evolved from. Due to the length of the message protocol keys and specifications, a full example and description of each message type's structure will not be included in this chapter. Readers are encouraged to review further Fog Protocol documentation included in the appendix.

4.2 Identification Information

The identification information is a unique message in FogKit that defines the name, identifier, and capabilities of a Fog peripheral device. This identification format is inpsired by XGSN's sensor metadata annotations outlined in section 3.4.1. These annotations include information about a sensor's location, type, properties, and more[11]. However, in order to remain simple and convenient, the Fog protocol only includes the device's id, name, type, and capabilities as annotations. An example of the configuration JSON message is shown below:

```
{
   "name":"Fog Pedometer",
   "type":"fitness_tracker",
   "id":<id>,
   "capabilities": [
      "step_count",
      "walk_distance"
   ]
}
```

4.2.1 ID Attribute

The Id attribute is the unique identifier for the device. This attribute must be unique to the network as it is used for network routing. An example of an ID that could be used could be a MAC address which is unique identifier assigned by manufacturers to every electronic device. Devices set their own device id so that devices can still communicate even after disconnecting and reconnecting to the network.

4.2.2 Name Attribute

The name string attribute specifies the name of a Fog peripheral. This information should not be used for routing information across the network as it is simply provided as a human-readable way to describe a device. An example of the name attribute might be "Spencer's Fog Peripheral."

4.2.3 Type Attribute

The type attribute is a string that specifies the type of the Fog peripheral device. This would be used to identify devices on the network that offer some desired capabilities and functionality. Developers may use types that are included in the Fog protocol specifications to ensure interoperability of devices, or they may define their custom types for their own usage. Examples of types might include "heart rate monitor" or "game controller."

4.2.4 Capabilities Attribute

The capabilities attribute specifies what functionality the device offers. This capabilities attribute is analogous to the sensor metadata annotations used in XGSN, explained in section 3.4.1[11]. Developers may use capabilities from a list provided by the Fog protocol to ensure interoperability or define their own for their devices. The capabilities are represented as an array of strings, and may include descriptions such as “blood pressure,” “heart rate,” and “temperature.”

4.3 Notification Information

When a Fog peripheral wishes to send data across the network, it updates the Notify characteristic to indicate to the central that it has a new message. This notification is represented as a boolean with key “newdata.” When this boolean changes from false to true, the central will initiate a read request to retrieve the new data. When the read is complete, the boolean will be set to false. An example notification is shown below:

```
{
   "newdata":true
}
```

4.4 Message Header Structure

All Fog messages share a common message header structure as part of their JSON object. The Fog protocol header includes aspects from both layer 3 and layer 4 technologies in the OSI model. This context information may include the source of the message, destination of the message, confirmable flag, identifier, response identifer, version number, and type. Shown below is an example of a Fog message header without the body of the message:

```
{
   "source":"JF7ES3",
```

```
   "destination":"KF82Y",
   "confirmation":true,
   "id":123,
   "response_id":456,
   "version":1,
   "type":"type",
    .  .  .  .
    .  BODY  .
    .  .  .  .
}
```

4.4.1 Source and Destination Attributes

The source ("source") and destination ("destination") attributes of the header are used to route the messages. A Fog device will set the source attribute to its own identifier and then the destination attribute to the destination's identifier or list of destination identifiers. Additionally, the Fog device can specify "BROADCAST" as the destination, which will result in the message being received by all connected devices.

When a Fog device sends a message to the Fog central, the central will read the message's header. If the message is destined for the central device, the central will consume, and if necessary, respond to the message. However, if the message is destined for another Fog device, the central will then relay the message to the intended destination. If the intended destination does not exist or is not connected, the central will respond with an error message.

4.4.2 Confirmation Attribute

The confirmation attribute ("confirmation") indicates whether or not the message is to be sent using reliable transmission. The idea of confirmable messages is borrowed from CoAP's interaction model explained in section 3.6.2. CoAP specifies four types of messages: confirmable, non-confirmable, acknowledgement, and reset[42]. This allows the system to verify that a message was received correctly or an error occurred.

Similarly, a Fog message can be marked as confirmable or non-confirmable. If a Fog message is marked confirmable, the recipient will send either an acknowledgement or error back to the sender upon receiving the message.

4.4.3 Message ID and Response Message ID Attriutes

The message identifier ("id") and response message identifier are used to coordinate message sending and responding between Fog devices. These attributes are optional, as these identifiers are added for the benefit of the transmitting device so that it may identify a response to a message. When a response message is sent, the response message identifier will contain the original message's identifier, allowing the Fog device to identify the what request the response belongs to.

The message and response ids correspond to the message id and token fields in the CoAP message format, outlined in section 3.6.2 and is used in a similar manner. An example of its usage may be a peripheral device that sends a request with a message id of 807. The recipient of this message may do some processing and then respond with new data. The response message will have a response message identifier of 807 so that the sender will know that the new incoming data belonged to the request with id 807. Alternatively, the recipient may respond with an error message, indicating that there was a problem with the received data.

4.4.4 Version Attribute

Similar to the version attribute ("version") in an CoAP header, the version attribute of the Fog message header indicates the version of the protocol being used[4]. Over time, the structure of the Fog protocol may evolve to accomodate new features and requirements. It's possible that these changes may result in a loss of compatibility. The version attribute can be used to identify the protocol version of the peripheral device so that a central can communicate with it.

4.4.5 Type Attribute

The type attribute ("type") in the message header identifies what structure and keys the rest of the message will have. The types of messages include event messages, post messages, query messages, error messages, and network status messages. This attribute is unique to the Fog protocol, so a more in-depth review of the structure and function of these messages is included in the following sections of this chapter.

4.5 Event Messages

Event messages are a paradigm that is unique, or at least unusual, in IoT protocols. They are used to notify other Fog devices that an event or action occured, similar to that of events in event-driven programming. An example of an event might include that a button was pressed or that an operation occured. These event messages have attributes to identify the event type ("event-type") and an optional dictionary attribute ("context") to provide context for the event. An example of an event for a new message with the context information of the message text and sender name is shown below:

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"event",
   "event" {
      "event-type":"new-message",
      "context": {
         "text":"I'm on my way!",
         "sender":"Bruce Wayne"
      }
   }
}
```

4.5.1 Event Type Attribute

The event type attribute is a required key that identifies the type of event that occurred. This key is a user-defined string set by a Fog peripheral device. The intended recipient of the message will need to be able to interpret the string and act accordingly. Examples of an event type might include "button pressed," "operation occured," or "settings changed."

4.5.2 Context Attributes

The content attribute ("context") is an optional dictionary of context information for the event that occured. This dictionary may be filled with user-defined content from the sending device. However, as with the event type attribute, the recipient must know how to interpret this data. Examples of how this might be used could be with a "button pressed" event. The content of the event might include the identifier of the button and how many times it was pressed. Alternatively, the event type might be "operation occured," with the content of the event including the name of the operation, the runtime, and some minimal results data.

4.6 Post Messages

Post messages are used to send large amounts of data to another device. They're composed of a collection of objects. The objects do not necessarily have to be of the same type, though the format and schema of each type must be consistent across the entire system. An example of a post message with medical data consisting of heart rate and blood pressure information is shown below:

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"post",
   "post": [
      {
```

```
         "type":"medical_reading",
         "data": {
            "reading_id":182
            "heart_rate":68,
            "blood_pressure": "120/80"
         }
      },
      {
         "type":"medical_reading",
         "data": {
            "reading_id":183
            "heart_rate":63,
            "blood_pressure": "115/75"
         }
      }
   ]
}
```

4.6.1 Post Attribute

This post attribute is an array of the JSON objects themselves. The objects have the following structure:

Type Sub-Attribute

The type attribute specifies the type and name of the object in the post. The attribute must be a string and can be considered like a the class of the object used throughout the entire network. In the case of the central, this attribute specifies the table from which to query objects.

Data Sub-Attribute

The data attribute specifies the schema and values of the object represented as JSON object. The keys in the objects represent the schema of the object. Developers are free to include as many attributes in the object as necessary, however, the objects are limited to a depth of one key-value pairing. That is to say, an attribute of a JSON object must be of type String, Int, Boolean, or Double. Arrays and dictionaries are

not supported at this time, however this functionality may be added in the future. These types must remain consistent across the entire network. The values, of course, can vary for the specific instances of the objects.

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"post",
   "post": [
      {
         "type":"location",
         "data": {
            "lat":35.305005,
            "long":-120.662494
         }
      }
   ]
}
```

4.7 Query Messages

Query messages are used to send requests for information stored on another device, such as the central. The query message's structure is reminiscent to that of a basic SQL statement, including a SELECT and WHERE attribute. A recipient device will execute the query and respond with the resulting data.

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"query",
   "query": {
      "select":"steps",
      "where":"(type == 'running') OR (type == 'stairs')"
   }
}
```

4.7.1 Select Attribute

The select attribute ("select") corresponds to the SELECT attribute of SQL query. It identifies the type of object that the query is for. These types correspond to the types of objects sent in Post messages. At this time, only one object may be specified with the select statement.

4.7.2 Where Attribute

The where attribute corresponds to the WHERE attribute of an SQL query. It includes the conditions used to evaluate the query. This property is optional, and if left out of the message, the recipient should respond with every object of the given type.

Overview of Syntax

As the WHERE clause of query is more complex than a simple key-value pairing, this section will discuss the syntax of the where clause. The syntax evolved significantly over the course of implementation. Originally, the conditions of the clause were represented in JSON form. Logical operations were represented as arrays of statements or other logical operations. Statements were dictionaries that included a key, value, and operation to be performed for that statement.

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "query": {
     "select":"workout",
     "where": {
        "and": [
           "or": [
               {
                  "key":"exercise",
                  "value":"running",
                  "operation":"=="
               },
```

```
                {
                    "key":"exercise",
                    "value":"jogging",
                    "op":"=="
                }
            ],
            {
                "key":"distance",
                "value":1,
                "operation":">"
            }
        ]
    }
}
}
```

However, this quickly proved to be impractical to both build up the JSON object on the sender and to parse out the JSON on the recipient due to excessive nesting of objects and tedious specification of keys, values, and operations. The syntax then evolved into a simple string statement that developers could easily enter into a JSON object.

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"query",
   "query": {
      "select":"workout",
      "where":"(exercise == 'running' OR exercise == 'jogging')
        AND (distance > 1)"
   }
}
```

Since this framework is written with an iOS central and data storage is built on top of Apple's CoreData framework, the syntax for the query is identical to the syntax for CoreData, so that it may directly map to the syntax of a CoreData NSFetchRequest, the object used for querying[2]. This allows developers to take advantage of a diverse and efficient set of query operations without the implementation overhead.

However, there drawbacks to this decision that need to be noted. First, CoreData's predicate syntax, although similar to other syntaxes, is unique and would require developers to spend time learning it. Additionally, the syntax is owned and controlled by Apple, meaning any changes in syntax could introduce errors in Fog networks.

4.7.3 Query Responses

Query response messages are the responses to a query. These response objects contain an array of objects that result from the query. They have the same dictionary representation as when they are posted to another device and include the type of object and the data for that object.

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"query_response",
   "query_response": [
      {
         "type":"workout",
         "data": {
            "calories":243,
            "exercise":"running"
         }
      },
      {
         "type":"workout",
         "data": {
            "calories":132,
            "exercise":"jogging"
         }
      }
   ]
}
```

4.7.4 Deleting

Deletion of objects shares a similar synatx and structure to the querying of objects. As opposed to the "select" keyword which specifies which objects the query should return, the "delete" keyword is used to indicate that the results should be deleted. Furthermore, developers can include a WHERE clause that specifies which objects to delete. The example below demonstrates the deletion of all "workout" objects where the "exercise" attribute equals "walking:"

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"query",
   "query": {
      "delete":"workout",
      "where":"exercise == 'walking'"
   }
}
```

4.7.5 Updating

The format for updating objects also shares a similar syntax and structure to the querying of objects. Again, rather than using the "select" keyword, the "update" keyword will be used. All the results of the query will be updated to have the contents of the dictionary specified by the "data" key. This method of specifying objects to update can easily update a large number of objects. If the user desires to update a single object, a primary key such as the "reading-id" identifier can be used to identify a unique instance of an object. An example of a message that updates the values of the medical reading object with a reading id of 182 is shown below:

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"query",
   "query": {
      "update":"medical-reading",
```

```
      "where":"reading-id == 182"
       "data": {
          "reading-id":182
          "heart_rate":68,
          "blood_pressure": "120/80"
        }
   }
}
```

4.8 ACK and Error Messages

As mentioned in section 4.4.2, the Fog protocol transmits data reliably when the confirmation flag is set on a message. The behavior of ACK and Error messages takes a design cue from the CoAP messaging model in section 3.6.2. The Fog protocol ACK message is sent when a recipient device receives data that was transmitted reliabily. Similarly, the Fog protocol error messages are sent when a recipient device receives data that has an error.

4.8.1 ACK

ACK, or acknowledgement messages, are very simple messages that are sent to confirm that the recipient has received a message. Since they contain a response id, context information is not necessary. Shown below is an example of an ACK message:

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "response_id":5385,
   "type":"ACK"
}
```

4.8.2 Error

Error messages are messages that are sent back to the sender when an error has occurred. These errors can fall into two categories: errors in transmission and errors in the data. Errors in transmission can be sent by the central when a peripheral is unreachable or does not exist. For example, if a peripheral sends a message to the device with id x, but there are no devices with id x connected to the central, the central will send the error message. Errors in the data are much more diverse, but may include cases of invalid structures, missing data, etcetera. For example, a device may send a query to the central for an object that is not stored on the central. The central will then send an error message with an appropriate explanation. Shown below is an example of an error message that indicates a problem with a query:

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "response_id":5385,
   "type":"error",
   "error": {
      "id": 1942,
      "message": "Database does not have object 'heart_rate'"
   }
}
```

4.9 Network Status Requests

Network status requests are for obtaining information about the devices currently connected to the network. Fog peripherals may need to identify other devices on the network which have desired characteristics.

4.9.1 Network Status Queries

Fog peripheral devices may query the central device for information on devices currently connected to the network. This information includes a device's id, name, type,

and capabilities. The querying syntax for devices is similar to that of the querying for objects, but has a few differences. For example, developers may search for devices by their id, name, and type much like a standard query. Shown below is an example of a network status query for any device with type "heart rate monitor:"

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"status",
   "status": {
      "query":"type == 'heart rate monitor'"
   }
}
```

However, if a user wants to query by devices that have a specific capability, the syntax is slightly different. This is because the capabilities of a device are stored as an array, not a simple value. The syntax for searching for a capability on the backend looks like "ANY capabilities.type == ," which is specific to the implementation of the persistent store. To abstract this away, developer can use the keyword "HAS CAPABILITY" to query for devices with a specific capabiilty. An example of the usage is shown below:

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"status",
   "status": {
      "query":"type == 'trigger' OR HAS CAPABILITY 'steps'"
   }
}
```

4.9.2 Status Responses

The response of these network status requests includes the information from the connected devices that satisfy the query. The response includes an array of dictionaries

of device metadata which includes the ID, name, type, and capabilities. The example below shows a response that returned the Fog Tracker device, which is has the capability "steps" and the Fog Button device, which has the type "trigger."

```
{
    .  .  .  .
    . HEADER .
    .  .  .  .
   "type":"devices"
   "devices": [
      {
         "id":"NCC1701",
         "name":"Fog Tracker",
         "type":"fitness-tracker",
         "capabilities": [
            "steps",
            "heartrate",
            "set-transfer-interval",
            "set-post-interval"
         ]
      },
      {
         "id":"R2D2C3PO",
         "name":"Fog Button",
         "type":"trigger",
         "capabilities": [
            "event-trigger",
            "set-destination"
         ]
      }
   ]
}
```

4.10 Comparison to Other Protocols

Although the Fog protocol takes many design cues from RPL and CoAP, there are several important differences in usage that should be noted, including how messages are routed across the network, differences in the complexity of these messages, and differences in the length of messages.

4.10.1 Message Routing

In the Fog protocol, all messages are routed to their destination through a central control point. This control point (the Bluetooth central) then analyzes the message and relays it directly to its destination. This is significantly different from RPL's routing methods, which are much more dynamic and direct. In an RPL network, the nodes construct a destination-oriented acylic graph using an algorithm known as the distance-vector protocol[36]. In this algorithm, information about each node and its connected neigbors is used to determine paths throughout the network[36]. Messages are then sent directly to their destination through the optimum path (as opposed to through a central point).

The Fog protocol was designed to use a central control point because it is compatible with the central-peripheral paradigm of Bluetooth. Unlike nodes in an RPL network, the only Bluetooth devices ("nodes") that can initiate read or write connections are centrals. This means that peripherals are not capable of directly connecting to each other and transferring data, so the data must be transferred through a central which relays messages.

4.10.2 Message Content Complexity

In the Fog protocol, there are many types of messages, including post, query, event, et cetera. These message types have different formats to accomodate the different types of data that are sent across the network. However, this variety of messages results in a higher degree of complexity to comparable protocols. For example, CoAP specifies only four methods for messages, including GET, PUT, POST, and DELETE which can be used to send generic payload data[42].

The additional complexity in the Fog protocol messages results from the inclusion of all the networking and application layers specified in the OSI model. The structure of the Fog messages helps ensure consistency and interoperabilty of application data between devices on the Fog network, something that is not ensured in other protocols.

4.10.3 Message Size

In the Fog protocol, messages are formatted as JSON objects, which contain string representations of data. Formatting data as strings is a very inefficient way to represent data that results in relatively long message sizes. On the other hand, protocols such as RPL and CoAP format their messages using binary representations of data[34][42]. These binary representations are often much more efficient representations and result in relatively smaller message sizes.

Although Fog messages have relatively inefficient representations of data, their use of JSON formatting is significantly more convenient for developers to work with. The notation is human-readable which allows for easy debugging and supports a key-value data model which allows for easy data retrieval.

Chapter 5

FogKit

5.1 Overview

As explained in previous sections, there are now many Bluetooth peripherals that connect to smart phones and other devices. Some of these Bluetooth peripherals may need to communicate with other devices or store significant amounts of data. For example, a Bluetooth smart watch could display information from a user's Bluetooth enabled heart rate monitor, pedometer, or other devices as wearer's average heart rate or distance walked over a time period.

Such a watch would need to pair and maintain a connection with each device. It may also require a significant amount of memory to store the data from the other peripheral devices. While this is certainly possible to implement, managing a web of connections between devices is inefficient and adding significant storage capactities to embedded devices is costly.

This is where FogKit can help: it provides a framework for Bluetooth communication and external data storage for connected peripheral devices. FogKit uses the Fog protocol for communication and external data storage. With this framework, a set of Bluetooth peripheral devices can connect to an iOS central device. The iOS cen-

tral is capable of routing communication between the devices and acting as external storage for the devices. This reduces the networking complexity for the peripherals and places the burden on a far more capable iOS device. Additionally, it eliminates the need for the peripheral devices to have large amounts of storage.

The FogKit and Fog protocol network is organized into a star topology. In this arrangement, many Bluetooth peripherals are connected to a single Bluetooth central, which acts as a server and router. This design was chosen for two reason:

1. it is consistent with the master-slave paradigm of Bluetooth
2. it allows for a single point of access to the rest of the network, simplifying the implementation for the peripherals

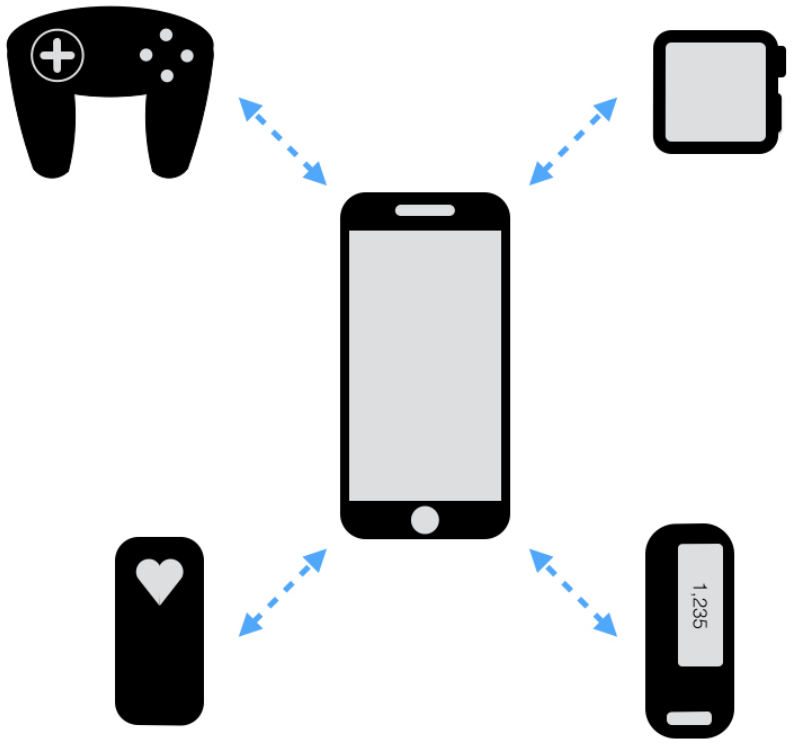

Figure 5.1: The FogKit star topology.

5.2 Framework Design

The FogKit framework provides functionality for the central and peripheral roles. Although there is some overlap and shared code for these functions, they can be considered to be separate. For this reason, the design details for each will be considered as different sections.

5.2.1 Central Framework

The central framework handles the responsiblities of the Fog Bluetooth central, including the routing of communication between peripherals, handling of data posts and queries, and the interaction between the currently running iOS device.

Although the FogKit framework does not have the same physical separations as the general IoT stack outlined in section 3.1, it does share the same logical layers: edge technology, access gateway, middleware, and application.

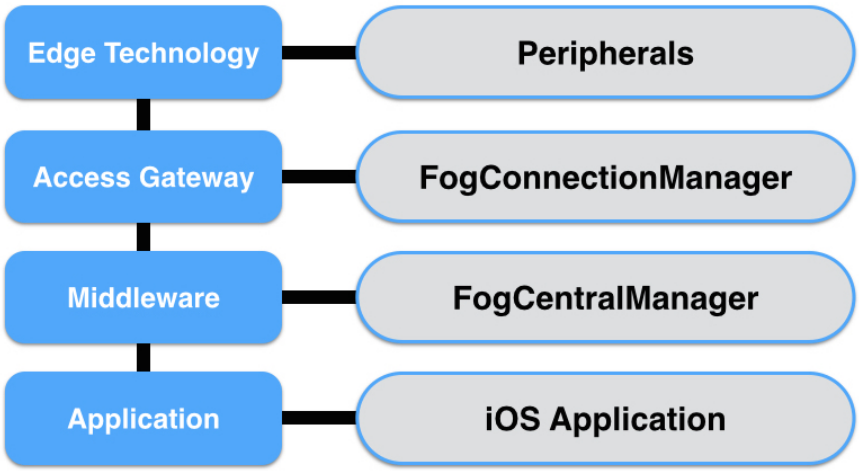

Figure 5.2: The FogKit layers.

The majority of the central framework is organized into several manager classes, including the FogConnectionManager, FogCentralManager, and FogQueryManager classes. Each of these classes plays an important role in managing Bluetooth connections, handling the central's data and operations, and executing queries and posts,

respectively.

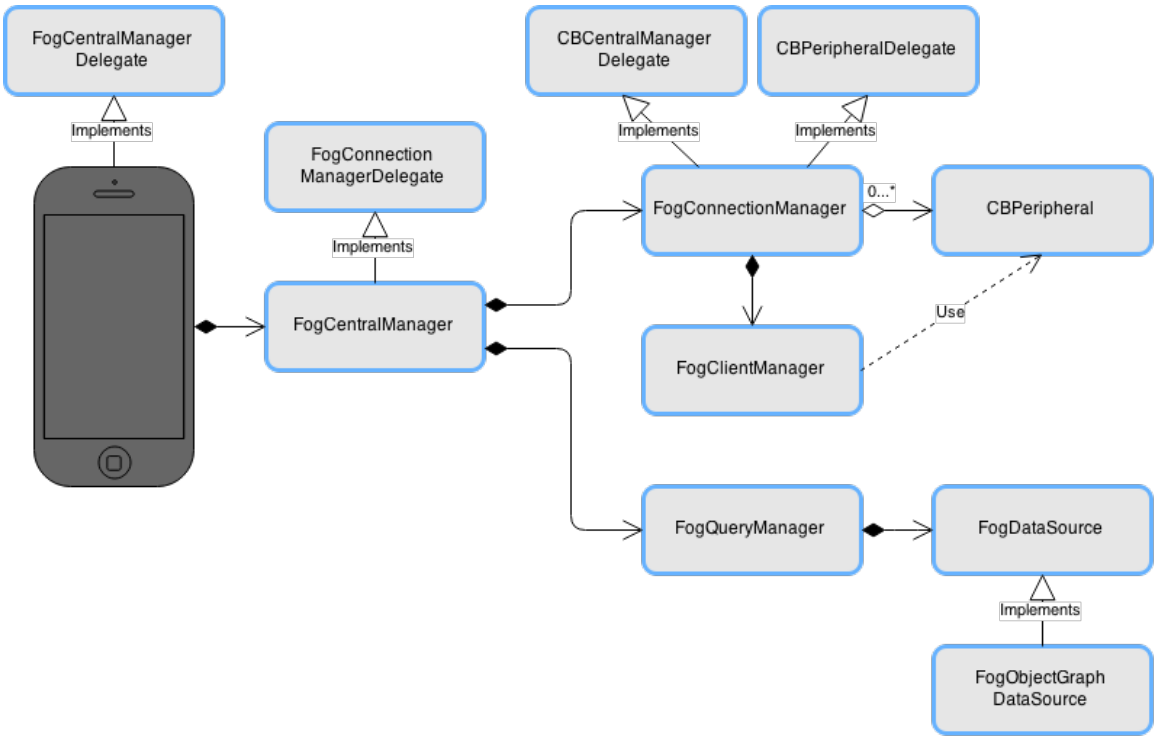

Figure 5.3: UML overview of FogKit Prototype Framework.

FogConnectionManager

The FogConnectionManager class is responsible for managing the Bluetooth 4.0 LE communication between the connected peripherals and the iOS central itself. This class will scan for Bluetooth peripheral devices that are currently advertising the Fog service. Once a device is discovered, the class will integrate it into the Fog network and allow other devices to communicate with it.

This class is also responsible for routing data between the Fog peripheral devices. When a peripheral sends data, this class will examine the destination of the message. If the message is intended for the iOS central, such as an event, post, or query, the FogConnectionManager will delegate the message to the FogCentralManager. However, if the message is intended for another Fog peripheral, the FogConnectionManager will

forward the message to the correct device.

FogClientManager

The FogClientManager class is responsible for managing the Fog network state information, including the peripherals currently connected and descriptive information, such as their names, identifiers, and capabilities. This repository of information allows peripheral devices to query the central for a list of other connected devices to which they can send or request data.

The class is capable of returning information regarding devices that meet specific requirements by executing queries. For example, a device may request information regarding any devices that are capable of sensing temperature. The class will execute the query and return the information from the devices that meet this requirement.

FogCentralManager

The FogCentralManager class is responsible for managing the responsibilities of the Fog central role, handling posts and queries to the central's peripheral store, and interfacing with the Fog iOS app.

The central is responsible for managing a persistent store of data for the central and peripheral devices. The central manager will handle queries and posts by managing their logistics, including the pre-processing of queries or posts into the correct data format, executing the request through the FogQueryManager, processing the returned request data, and sending the response back to the correct peripheral devices.

Finally, the central class is also responsible for interacting with the hosting central iOS app. Developers may develop their own central iOS application to be included in the Fog network. This class will delegate to the application events or data that is intended to be consumed by the central.

FogQueryManager

The FogQueryManager is responsible for abstracting away the different persistent stores available for use in the FogNetwork. It provides an asynchronous wrapper around any class that implements the FogDataSource protocol, allowing developers to easily "swap" the underlying persistent store implementation.

The FogDataSource protocol defines a small set of methods for inserting, deleting, updating, and querying data from a data source. One implementation of this FogDataSource protocol is the FogObjectGraphDataSource class. This class is a wrapper around Apple's persistent storage framework, Core Data[2], which allows developers to store data persistently on the iOS device. Additionally, the database is dynamic, with the ability to add new types of data during runtime.

Although Core Data is technically an object-graph, for the purposes of this discussion Core Data will be compared to a standard SQL relational database[2]. Core Data storage is defined by a collection of NSEntityDescriptions (similar to a SQL table) consisting of a set of NSAttributeDescriptions (similar to a column of a table) which describe the name and type of data[2]. When data is inserted or queried from Core Data, it is represented as an instance of an NSManagedObject (similar to a row in a table)[2].

When the FogObjectGraphDataSource class is given a post with a new type of data as defined in the Fog protocol section 4.6, it will dynamically create a new NSEntityDescription and add the NSAttributeDescriptions with the provided names and inferred types. After this action is completed, it will create an instance of an NSManagedObject for every object in the post and save them persistently.

5.2.2 Peripheral Framework

The peripheral framework handles the responsiblities of the Fog Bluetooth peripheral, including the setup, management of the communication, and interaction with the peripheral application. The peripheral framework is primarily implemented in

the FogPeripheralManager class. However, it does make use of some of the shared code, explained in the Shared Code section.

The setup of the Fog peripheral includes the advertisement of the Fog services and characteristics necessary for communication in the Fog network. Using these services and characteristics, the peripheral can communicate its name, identifier, capabilities, and more to the central.

The management of communication includes the code that transfers data to and from the Fog central device. The code is responsible for handling the incoming and outgoing Bluetooth packets, reassembling them into a single piece of data, and determining whether to handle it as an event, query, or another type of Fog message.

Finally, the class is responsible for interacting with the client application, including processing incoming messages into a convenient format, delegating these messages to a class that implements the FogDataManagerDelegate protcol, and providing methods that abstract away the details of sending messages to another device.

5.2.3 Shared Code

Unfortunately, the difference in the Bluetooth master-slave roles between the central and peripheral devices resulted in the majority of networking code being unshareable. Thus, the code that was shared between the two frameworks consisted only of classes that abstract away Fog data structures and defined Fog constants.

The Fog data structures defined in the chapter 4, the Fog Protocol, were abstracted into a set of Fog message objects, including the FogQuery, FogDataObject, and FogEvent. These classes made working with the Fog protocol easier because they abstracted away the details of parsing JSON and provide convenient getter and setter methods for the properties of the objects.

There were also many constants defined between the two frameworks, including Blue-

tooth constants such as a characteristic's unique identifier, Fog message constants such as the keys for the JSON objects, and finally the Fog query constants which included the keys and constants used in querying.

5.3 Bluetooth GATT Profile

FogKit implements this functionality by creating a set of requirements, known as the GATT profile, that specify a service and characteristics that are used for communication through the Fog protocol. The Bluetooth requirements include the use of the Fog Bluetooth service with four characteristics:

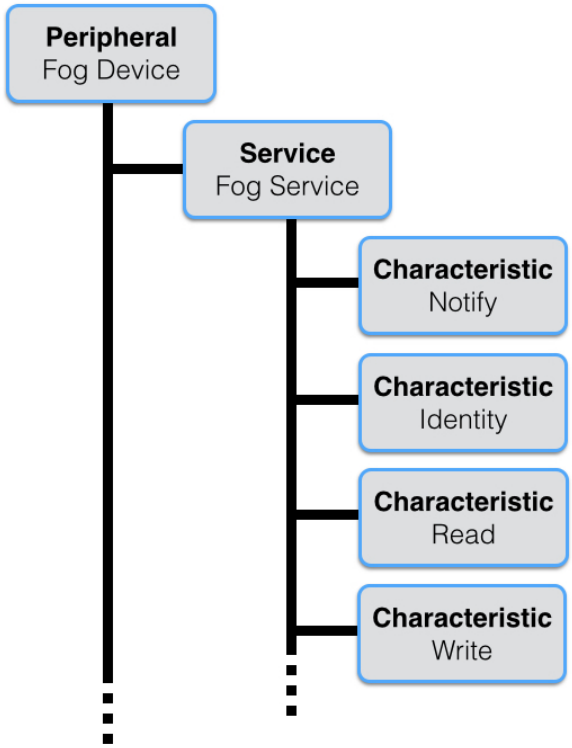

Figure 5.4: The GATT profile for a Fog Bluetooth peripheral.

- **Read:** A characteristic that acts as a read endpoint for the peripheral. All output data is transferred through this characteristic.

- **Write:** A characteristic that acts as a write endpoint for the peripheral. All input data is transferred through this characteristic.
- **Notify:** A characteristic that a central may subscribe to and monitor for updates. As these characteristics have limited data capacity, this characteristic simply notifies the central that the peripheral has data to be processed. The central will then initiate a data transfer using the read characteristic.
- **Identity:** A characteristic that contains identity information for the peripheral, including a unique identifier, name, and list of capabilities.

Chapter 6

Expected Use Cases

Validation of the Fog protocol will be offered through a set of thought exercises and example applications that demonstrate the expected use cases rather than through tests of network performance, data usage, etcetera. Although these metrics are very important in evaluating the efficiency and usefulness of the protocol, we have decided that they are not relevant to this thesis because the intent of the thesis was to focus on the design rather than the implementation of the protocol. The implementation of the Fog protocol, FogKit, is a prototype which although demonstrates and proves the concept, without further work and optimizations would not have metrics representative of the final product.

Throughout the research of this project, it became clear that there are three broad categories of wearables devices:

1. devices that primarily sense data and post that information to a central
2. devices that primarily send events or notifications to a central
3. devices that are fully integrated systems capable of sending and receiving data and performing a variety of tasks

The following sections will review the categories of devices and demonstrate how the Fog protocol can be or is used to implement their primary functionality. As the

functionality of these devices will have a significant amount of overlap, the amount of explanation for these repeated functionalities will be reduced by having each section focus on a different area of the Fog protocol.

6.1 Thought Exercises

6.1.1 Wearable Sensing Devices

Many of the wearables available today focus on activity and health readings, such as steps taken or heart rates, such as the FitBit[19]. The core functionality of these wearables is gathering data and posting it to a central for storage and processing. This section will propose the Fog protocol implementation of an activity device that tracks a wearer's steps and heartrate. The rate at which the proposed devices take readings and transfer the data will be configurable by another device. Additionally, the device will be capable of handling basic queries for the current heartrate or number of steps.

Identity Metadata

When a Fog device connects to the network, it must provide metadata information about itself, including its name, type, id, and list of capabilities. A proposed wearable activity tracker that implements the Fog protocol could have a name such as "Fog Tracker," a type of "fitness-tracker," a unique id such as "NCC1701," and a list of capabilities, including the following:

1. "steps," indicating that the device is capable of tracking the number of steps taken
2. "heartrate," indicating that the device is capable of taking heart rate measurements
3. "set-sense-interval," indicating that the device has a configurable sensing interval

4. "set-post-interval," indicating that the device has a configurable data transfer interval

An example message for this device's metadata is shown below:

```
{
   "name":"Fog Tracker",
   "type":"fitness-tracker",
   "id":"NCC1701",
   "capabilities": [
      "steps",
      "heartrate",
      "set-transfer-interval",
      "set-post-interval"
   ]
}
```

Configuration

This wearable device will be configurable, allowing users to set the rate at which the device takes readings and posts the data for storage. This configuration can be implemented by sending the device an event message that has configuration information. This event might have a type of "configuration" with two context parameters, "transfer-interval" and "read-interval", which represent a time interval in milliseconds. An example where the device is being configured to take a reading once a minute (60,000 milliseconds) and a post interval of five minutes (300,000 milliseconds) is shown below:

```
{
   "source":"CENTRAL",
   "destination":"NCC1701",
   "confirmation":true,
   "id":123,
   "version":1,
   "type":"event",
   "event" {
      "event_type":"configuration",
      "context": {
         "post-interval":300000,
```

```
         "read-interval":60000
      }
   }
}
```

Since the sender set the confirmation flag to true in this message, the device will respond with either an ACK or an error message. If the configuration settings are valid, the device will respond with an ACK, shown below:

```
{
   "source":"NCC1701",
   "destination":"CENTRAL",
   "response_id":123,
   "version":1,
   "type":"ACK"
}
```

On the other hand, if the configuration settings are incorrect, the peripheral could respond with an error message that indicates there was a problem. An error message that indicates that a problem with the configuration settings is shown below:

```
{
   "source":"NCC1701",
   "destination":"CENTRAL",
   "response_id":123,
   "version":1,
   "type":"error",
   "error": {
      "id": 1942,
      "message": "Invalid configuration settings"
   }
}
```

Posting Data

As a fitness tracker, the majority of communication done by the device will be posting the wearer's readings for heart rate and number of steps to the destination device. This data will be represented in two objects, a "heartrate-reading" object which contains a timestamp and heartrate and a "steps-reading" object which contains a time and step count.

```
{
   "source":"NCC1701",
   "destination":"CENTRAL",
   "confirmation":true,
   "id":435,
   "version":1,
   "type":"post",
   "post": [
      {
         "type":"heartrate-reading",
         "data": {
            "timestamp":1825432
            "heartrate":68,
         }
      },
      {
         "type":"steps-reading",
         "data": {
            "timestamp":1825432
            "steps":2421,
         }
      }
   ]
}
```

This proposed device will verify that the data was posted successfully, so it will set the confirmation flag to true. When it receives an acknowledgement, the device may do some cleanup, such as deleting old information.

```
{
   "source":"CENTRAL",
   "destination":"NCC1701",
   "response_id":435,
   "version":1,
   "type":"ACK"
}
```

However, if the device does not receive an ACK or receives an error, it will attempt to resend the information until it is successfully received or it runs out of memory and is forced to delete the data.

Handling Queries

This proposed device will also be capable of handling simple queries from other devices for the current step count or heart rate. This functionality could be used by devices that need to display or use the current information, but do not need to change the read or posting rates. In this situation, the peripheral may receive a query message like:

```
{
   "source":"RJ2DE",
   "destination":"NCC1701",
   "id":2341,
   "version":1,
   "type":"query",
   "query": {
      "select":"steps-reading"
   }
}
```

Since the device is constrained and only capable of handling queries for the current step count or heart rate, specifying a "where" clause is unneccesary. This peripheral would then respond with a query response, which may be formatted as:

```
{
   "source":"NCC1701",
   "destination":"RJ2DE",
   "response_id":2341,
   "version":1,
   "type":"query_response",
   "query_response": [
      {
         "type":"steps-reading",
         "data": {
            "time":1825432
            "steps":2421,
         }
      }
   ]
}
```

On the other hand, if there was an error or an unsupported operation in the query message, the peripheral could respond with an error:

```
{
   "source":"NCC1701",
   "destination":"RJ2DE",
   "response_id":2341,
   "version":1,
   "type":"error",
   "error": {
      "id": 246,
      "message": "Unsupported query operation"
   }
}
```

6.1.2 Wearable Event Device

Another type of wearable devices are those that primarily send events or notifications. An example of these devices might be a game controller, which sends events for button presses, or the aforementioned wearable keychain, DingBot[16]. The DingBot has a button that when pressed triggers an event which can be executed by a connected smartphone, such as turning on a light or sending a tweet. A device similar to the DingBot could potentially implement the Fog protocol. Furthermore, this device, since it is only a button, will be very constrained, and so this section will demonstrate how a constrained device may behave.

Identity Metadata

When a Fog device connects to the network, it must provide metadata information about itself, including its name, type, id, and list of capabilities. A proposed wearable activity tracker that implements the Fog protocol could have a name such as "FogButton", a type of "trigger", a unique id of "R2D2C3PO" and one capability:

1. "event-trigger", indicating that the device can trigger events

An example of this device's identification message is shown below:

```
{
   "name":"FogButton",
```

```
   "type":"trigger",
   "id":"R2D2C3PO",
   "capabilities": [
      "event-trigger"
   ]
}
```

Sending Events

When the button on this device is pressed, the peripheral will send an event to the specified destination device that includes the relevant context information, such as the event type and event ID. This message may look like:

```
{
   "source":"R2D2C3PO",
   "destination":"CENTRAL",
   "confirmation":false,
   "id":254,
   "version":1,
   "type":"event",
   "event" {
      "event_type":"trigger",
      "context": {
         "event-id":4921
      }
   }
}
```

This proposed device is very constrained and has limited computing power and battery life. In order to accomodate these limitations and reduce complexity, the device will not wait for ACK or error messages. However, this simplicity comes with a cost: without this handshaking, messages may not reach their destination reliably.

6.1.3 Wearable System

The final class of wearable devices are those that are fully integrated systems, such as a smart watch or smart glasses. As smart watches are becoming increasingly popular, this section will propose a smart watch that uses the Fog protocol. This device will

be fully featured, including a display with touch capability, health sensors such as a heart rate monitor and pedometer, and the ability to run 3rd party applications. It will fully implement the Fog protocol and be capable of bothing sending and receiving posts, queries, events, and more.

Identity Metadata

The proposed smart watch is fully featured and capable of executing 3rd party apps, so it's possible it would have an large list of capabilities. Although the proposed smart watch is capable of a wide variety of features, especially through the implementation of 3rd party applications, for the sake of this discussion, we will focus on the core functionality of the device with the following capabilities:

1. "steps", indicating that the device track the wearer's steps
2. "heartrate", indicating that the device can track the wearer's heartrate
3. "display-notifications", indicating the device can display notifications

```
{
   "name":"FogWatch",
   "type":"system",
   "id":"J83ZED",
   "capabilities": [
      "steps",
      "heartrate",
      "display-notifications"
   ]
}
```

Displaying Notifications

A key aspect of smart watches is the ability to display notifications from the wearer's smartphone. This functionality could easily be implemented by sending events to the device that contain the necessary context information, such as the application and message to display. For example, a post for a new e-mail notification may be modeled as follows:

```
{
   "source":"CENTRAL",
   "destination":"J83ZED",
   "confirmation":true,
   "id":763,
   "version":1,
   "type":"event",
   "event" {
      "event_type":"notification",
      "context": {
         "notification-type":"email",
         "email-id":654
         "display-message":"New e-mail from Cal Poly"
      }
   }
}
```

The "display-message" attribute would contain the information shown to the wearer as the notification. This watch could offer the ability for the wearer to click the notification for further information. The "notification-type" attribute tells the smart watch what application should handle the notification while the rest of the context may be used for application specific information, such as the ID of the e-mail. This may be used by the device to fetch the necessary detail information.

Discovering Other Devices

One key aspect of the Fog protocol is the ability for devices to search for other devices with a desired capability and communicate with them. This proposed smart watch may implement that functionality to display information from another sensor, such as a blood pressure monitor. This process begins with the device sending a network status request to the central for a device with the capability "blood-pressure:"

```
{
   "source":"J83ZED",
   "destination":"CENTRAL",
   "confirmation":true,
   "id":835,
   "version":1,
   "type":"status",
```

```
   "status": {
      "query":"HAS CAPABILITY 'blood-pressure'"
   }
}
```

The central device, which stores information about the connected devices, will then respond with the network status information. An example response that includes a medical tracking device with the capability "blood-pressure" is shown below:

```
{
   "source":"CENTRAL",
   "destination":"J83ZED",
   "confirmation":true,
   "response_id":835,
   "version":1,
   "type":"devices",
   "devices": [
      {
         "id":"S92JDL2",
         "name":"Fog Pressure",
         "type":"medical-tracker",
         "capabilities": [
            "blood-pressure",
            "blood-oxygen"
         ]
      },
   ]
}
```

Using this information, the smart watch can then send a query for blood pressure data to the medical tracking device. This process would be similar to the query process outlined in the following section, 6.1.3.

Sending Queries for Information

This proposed device will have the ability to send queries to other devices for information to display. For example, this device could have a calendar application that displays upcoming calendar events. These events are likely stored on the wearer's smartphone, so it would be necessary to send a query to the smartphone for this data that looks like:

```
{
   "source":"J83ZED",
   "destination":"CENTRAL",
   "id":4162,
   "version":1,
   "type":"query",
   "query": {
      "select":"events"
      "where":"start_date == 'June 13, 2015'"
   }
}
```

The smartphone may then respond with the relevant event information. In this case, there might be two events on June 13, 2015 including the Cal Poly Commencement Ceremony and a Graduation Party. The smart watch would then be able to display this information to the user. An example response could be formatted as:

```
{
   "source":"CENTRAL",
   "destination":"J83ZED",
   "response_id":4162,
   "version":1,
   "type":"query_response",
   "query_response": [
      {
         "type":"event",
         "data": {
            "name":"Cal Poly Commencement Ceremony",
            "time":"9:00am - 11:00am",
            "date":"June 13, 2015",
            "location":"1 Grand Avenue, San Luis Obispo, Ca"
            "event-id":4211
         },
         "type":"event",
         "data": {
            "name":"Graduation Party",
            "time":"7:00pm - 12:00am",
            "date":"June 13, 2015",
            "location":"555 Ramona Dr, San Luis Obispo, Ca"
            "event-id":7293
         }
      }
   ]
```

```
}
```

6.2 Demonstrations

In addition to the thought exercises, two sets of examples were developed with FogKit to demonstrate the querying, post, and event functionality of the framework. These applications were written in Swift and were executed on an iPod touch, iPhone, and iPad.

6.2.1 Post Application Example

The "number" example demonstrates the functionality of the post messages in the Fog protocol. This application consists of two components: one component that generates sets of ten random numbers from 1 to 10,000 and then sends them to the central to store them and a second component that queries the central and displays the numbers in a list.

Identity Metadata

Each component only needs one capability in order to be functional. The number generator has the capability "numbers", indicating that the device can generate numbers. The identity metadata for this component is as follows:

```
{
   "name":"Number Generator",
   "type":"generator",
   "id":"nbrgen",
   "capabilities": [
      "numbers"
   ]
}
```

On the other hand, the number display component has a capability of "display-numbers". This component's metdata is shown below:

```
{
   "name":"Number Display",
   "type":"display",
   "id":"nbrdsp",
   "capabilities": [
      "display-numbers"
   ]
}
```

Posting Numbers to the Central

When a user presses a button on the number generator application, a set of ten random numbers will be generated and posted to the central device for storage in a message as follows:

```
{
   "source":"NBRGEN",
   "destination":"CENTRAL",
   "confirmation":false,
   "id":435,
   "version":1,
   "type":"post",
   "post": [
      {
         "type":"number",
         "data": {
            "value":12352
         }
      },
      {
         "type":"number",
         "data": {
            "value":32561
         }
      },
       .  .  .  .  .  .
       .  8 MORE #'S  .
       .  .  .  .  .  .
   ]
}
```

Querying the Numbers from the Central

On the number display component, the user can decide whether to query for numbers that are greater than 5,000 or less than 5,000. An example of one of these queries is shown below:

```
{
   "source":"NBRDSP",
   "destination":"CENTRAL",
   "id":3213,
   "version":1,
   "type":"query",
   "query": {
      "select":"value < 5000"
   }
}
```

The central will then return a query response that includes the data that satisfies the conditions, as shown below:

```
{
   "source":"CENTRAL",
   "destination":"NBRDSP",
   "response_id":3213,
   "version":1,
   "type":"query_response",
   "query_response": [
      {
         "type":"number",
         "data": {
            "value":"3251"
         }
      },
      {
         "type":"number",
         "data": {
            "value":"2198"
         }
      },
      .  .  .  .  .  .
      . MORE NUMBERS .
      .  .  .  .  .  .
```

```
   ]
}
```

6.2.2 Event Applications Example

The "color" application consists of two components that demonstrate the functionality of the event messages in the Fog protocol. One component acts as a display, setting the color of the screen to a specified color and responding to queries. A second component acts as a remote, sending event messages to update the color on the color display.

Identity Metadata

As a simple color remote and display, each component only needs one capability in order to be functional. The color remote has the capability "control-color", indicating that the device can set colors. The identity metadata for this component is as follows:

```
{
   "name":"Color Remote",
   "type":"remote",
   "id":"CLRRMT",
   "capabilities": [
      "control-color"
   ]
}
```

On the other hand, the color display component has a capability of "display-color". This component's metadata is shown below:

```
{
   "name":"Color Display",
   "type":"display",
   "id":"CLRDSP",
   "capabilities": [
      "display-color"
   ]
}
```

Connecting the Remote to the Display

In order to remotely control the display, the remote component has to send a network status request to retrieve the id of the color display. This network status request is shown below:

```
{
   "source":"CLRRMT",
   "destination":"CENTRAL",
   "confirmation":true,
   "id":835,
   "version":1,
   "type":"status",
   "status": {
      "query":"HAS CAPABILITY 'display-color'"
   }
}
```

The central will process this network status request and return the correct identity metadata to the remote, which is shown below:

```
{
   "source":"CENTRAL",
   "destination":"CLRRMT",
   "confirmation":true,
   "response_id":835,
   "version":1,
   "type":"devices",
   "devices": [
      {
         "id":"CLRDSP",
         "name":"Color Display",
         "type":"display",
         "capabilities": [
            "display-color"
         ]
      }
   ]
}
```

Controlling the Display

The remote control allows users to control the display by typing in a color and sending the color to the display as an event message. The event message is shown below:

```
{
   "source":"CLRRMT",
   "destination":"CLRDSP",
   "confirmation":true,
   "id":254,
   "version":1,
   "type":"event",
   "event" {
      "event_type":"set-color",
      "context": {
         "color":"green"
      }
   }
}
```

If the display component supports the color, it will update the screen and return an ACK to the remote. However, if the display does not support the color, it will respond with an error message that indicates the problem:

```
{
   "source":"CLRDSP",
   "destination":"CLRRMT",
   "response_id":2341,
   "version":1,
   "type":"error",
   "error": {
      "id": 1,
      "message": "Unsupported display color"
   }
}
```

Querying the Display

The remote could also be used to query the display for the color that is currently shown. When the remote makes this request, it will send the display the following message:

```
{
   "source":"CLRRMT",
   "destination":"CLRDSP",
   "id":2341,
   "version":1,
   "type":"query",
   "query": {
      "select":"color"
   }
}
```

The color display will then respond with the correct color information for the query:

```
{
   "source":"CLRDSP",
   "destination":"CLRRMT",
   "response_id":2341,
   "version":1,
   "type":"query_response",
   "query_response": [
      {
         "type":"color",
         "data": {
            "value":"green"
         }
      }
   ]
}
```

Chapter 7

Future Enhancements

Although the Fog protocol has been demonstrated to work using the prototype implementation FogKit, there is still further research and work that needs to be done before it could be a viable product. As a prototype that was intended to guide the design of the protocol, certain aspects such as security, privacy, efficiency, and testing were set aside for future work. This section will outline the future work that needs to be done before this protocol can be fully launched.

7.1 Fog Protocol

7.1.1 Value Standardization

The Fog protocol was designed to be flexible and to accomodate any type of data structures. However, this flexibility introduces interoperability problems if similar data from different devices is formatted differently. This issue is caused by the Fog protocol's extensive use of keywords used in both the keys and values of the JSON messages. This issue could be dealt with by creating a standardized and documented vocabulary of all the keys and common data structures. Developers could then use this to ensure consistent use of the protocol.

7.1.2 Protocol Optimizations

There are many protocol optimizations that could be made to reduce the quantity and size of messages sent. Some messages, such as the event and query messages, only send one instance of a query or event per message. The format of these messages could be restructured to send an array of queries or events, reducing the overall number of transmissions. Furthermore, the size of the messages themselves can be reduced by significantly reducing the character count of messages through abbreviated keys. Alternately, the Fog protocol could define and support a set of verbose keys for development and debugging and a set of abbreviated keys for production environments.

7.1.3 Binary Data Transfer

One necessary but unimplemented feature of the Fog protocol is the ability to send binary data between devices. This would allow peripherals to send data that cannot be formatted as a string in a JSON message, such as a picture from a camera, audio from a microphone, or other types of raw sensor data. One possible solution would be to use a variant of JSON called Binary JSON (BSON) which has extensions to the notation that allow developers to use binary data types[29].

7.2 FogKit

7.2.1 Security

One significant omission in the FogKit framework is security. As a prototype, the FogKit framework does not take any security precautions, such as establishing secure connections by pairing devices. However, the release of Bluetooth 4.2 introduces many new features that extend security and privacy for Bluetooth devices. Bluetooth 4.2 introduces a new security model, the LE Secure Connection, which specifies a new key generation and pairing procedure[20]. This new model protects again passive eavesdropping and man-in-the-middle attacks[20].

7.2.2 Background Processing

Currently, the FogKit framework executes as part of an iOS application, which means that if the application closes the framework process is killed. Further implementation can be added to make the Fog framework execute in the background of the device even when the iOS application is not open. Ideally, the FogKit framework could be integrated as a service on all phones by default, but this would require support at the operating system level.

7.2.3 Implementation Testing

This thesis provides basic validation of the Fog protocol thorugh some simple real world examples. In order to thoroughly test all of the FogKit's potential, further testing will be required. The testing should consist of measuring performance characteristics such as response times, data transmission rates, message reliability, realtime detection of devices, etc. Although this testing would be useful to verify the FogKit functionality, it should be noted that although FogKit currently supports these tests, it is presently a proof-of-concept prototype and actual speeds and other numerical characterisitcs may not represent its future or potential characteristics.

Chapter 8

Conclusion

The Internet of Things is driving innovation in almost every area of our lives. Advancements in technology have led to the deployment of an innumerable number of diverse devices, ranging from simple sensors to fully integrated systems. One of the most popular classes of these devices is "wearables" that are worn on the body, such as a pedometer or smart watch.

The Fog protocol and FogKit framework were designed to facilitate communication between these wearable devices while minimizing implementation time for developers. This was accomplished through the definition of a protocol that used messages in the convenient JSON format. Additionally, the protocol was designed for use in a star network topology, which simplifies the implementation of the Fog protocol on Bluetooth peripheral devices by providing a single access point to the network.

Through the use of thought experiments with potential implementations and actual mobile devices with functional FogKit framework implementations, the usefulness of the protocol was demonstrated. The thought experiments focused on a potential network of devices consisting of a smart watch, fitness tracker, and a remote device. The actual mobile applications, executing on an assortment of iPhones, iPods, and iPads demonstrated simple applications involving devices detecting each other, querying capabilities, and sending data to each other.

Although the Fog protocol and FogKit framework are working as functional prototypes, additional capabilities need to be added before it can be released as a final product. These additions include new general features, security capabilities, and general optimizations. With these improvements, the Fog Protocol will hopefully be welcomed as a new and easy to use protocol for the Internet of Things.

Bibliography

[1] Apple. About core bluetooth, 9 2013. [Online; accessed 16-May-2015].

[2] Apple. Core data programming guide, 2014.

[3] Apple. Welcome to apple watch, 2015.

[4] H. Arora. Ip protocol header fundamentals explained with diagrams, 3 2012.

[5] K. Ashton. That 'internet of things' thing, Jun 2009.

[6] D. Bandyopadhyay and J. Sen. Internet of things: Applications and challenges in technology and standardization. *Wireless Pers Commun Wireless Personal Communications*, 58(1):4969, 2011.

[7] V. Beal. The 7 layers of the osi model, Apr 2015.

[8] Bluetooth. Architecture - overview of operations, 2015.

[9] Bluetooth. Bluetooth smart (low energy) technology, 2015.

[10] Bluetooth. The low energy technology behind bluetooth smart, 2015.

[11] J.-P. Calbimonte, S. Sarni, J. Eberle, and K. Aberer. Xgsn: An open-source semantic sensing middleware for the web of things.

[12] S. Cirani, M. Picone, and L. Veltri. mjcoap: An open-source lightweight java coap library for internet of things applications. *Interoperability and Open-Source Solutions for the Internet of Things Lecture Notes in Computer Science*, page 118133, 2015.

[13] Cisco. Fog computing, ecosystem, architecture and applications.

[14] Cisco. How many internet connections are in the world? right. now., 2013.

[15] D. Crockford. JSON. RFC 4627, IETF, July 2006.

[16] DingBot. Meet dingbot, 2014.

[17] M. Electronics. Wearable devices and the internet of things, 2014.

[18] D. Evans. How the next evolution of the internet is changing everything. Technical report, 04 2011.

[19] FitBit. Fitbit store, 2015.

[20] V. Gao. Everything you want to know about bluetooth-4.2 security — bluetooth technology website, Jan 2015.

[21] C. Gomez and J. Paradells. Wireless home automation networks: A survey of architectures and technologies. *IEEE Communications Magazine IEEE Commun. Mag.*, 48(6):92101, 2010.

[22] Google. Bluetooth low energy. [Online; accessed 16-May-2015].

[23] Google. Google glass.

[24] W. S. S. N. I. Group. Semantic sensor network ontology, 11ADAD.

[25] IEEE. Internet of things (iot) ecosystem study, January 2015.

[26] M. A. Khan, A. Khan, M. N. Khan, and S. Anwar. A novel learning method to classify data streams in the internet of things. *2014 National Software Engineering Conference*, 2014.

[27] G. M. Luigi Atzori, Antonia Iera. The internet of things: A survey, 2010.

[28] Microsoft. The osi model's seven layers defined and functions explained, Jun 2013.

[29] MongoDB. Bson.

[30] OpenIoT. Openiot - the open source internet of things, 11 2014. [Online; accessed 16-May-2015].

[31] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler. Standardized protocol stack for the internet of (important) things. *IEEE Commun. Surv. Tutorials IEEE Communications Surveys and Tutorials*, 15(3):13891406, 2013.

[32] M. Patel and J. Wang. Applications, challenges, and prospective in emerging body area networking technologies. *IEEE Wireless Commun. IEEE Wireless Communications*, 17(1):8088, 2010.

[33] G. Press. Internet of things by the numbers: Market estimates and forecasts, Aug 2014.

[34] Y. Rekhter and T. Li. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550, March 2012.

[35] T. Rueters. Gartner hype 2014 cycle for emerging technology, 2014.

[36] Z. Sheng, S. Yang, Y. Yu, A. Vasilakos, J. Mccann, and K. Leung. A survey on the ietf protocol suite for the internet of things: standards, challenges, and opportunities. *IEEE Wireless Commun. IEEE Wireless Communications*, 20(6):9198, 2013.

[37] B. SIG. Generic attribute profile, 2015.

[38] M. Swider and L. Prasuethsut. Best smartwatch 2015: what's the best wearable tech for you?, May 2015.

[39] C.-W. Tsai, C.-F. Lai, M.-C. Chiang, and L. T. Yang. Data mining for internet of things: A survey. *IEEE Commun. Surv. Tutorials IEEE Communications Surveys and Tutorials*, 16(1):7797, 2014.

[40] B. University. Distance-vector routing.

[41] J. Williamson, Q. Liu, F. Lu, W. Mohrman, K. Li, R. Dick, and L. Shang. Data sensing and analysis: Challenges for wearables. *The 20th Asia and South Pacific Design Automation Conference*, 2015.

[42] C. B. Z. Shelby, K. Hartke. The Constrained Application Protocol (CoAP). RFC 7275, IETF, June 2014.

[43] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *Internet of Things Journal, IEEE*, 1(1):22–32, Feb 2014.

Appendix A

Fog Protocol Keys

1 Identity Messages

Key	Required	Type	Description
name	yes	string	Developer-defined name
type	yes	string	Developer-defined type
id	yes	string	Developer-defined unique identifier
capabilities	yes	array of string	Array of developer-defined capabilities

2 New Data Notification Message

Key	Required	Type	Description
newdata	yes	boolean	Indicator that the device has new data that needs to be read

3 Network Status Messages

Key	Required	Type	Description
query	yes	string	Predicate that specifies which peripheral's metadata is requested

4 Network Status Response Messages

Key	Required	Type	Description
devices	yes	array	Array of dictionary information including peripheral metadata
name	yes	string	Developer-defined name
type	yes	string	Developer-defined type
id	yes	string	Developer-defined unique identifier
capabilities	yes	array of string	Array of capabilities

5 Message Header

Key	Required	Type	Description
source	yes	string	Fog ID of message sender
destination	yes	string or array of string	Fog ID of message destination
id	no	int	Developer-defined identifier for the message
response_id	no	int	Identifier of the message being responded to
confirmation	no	boolean	Indicator that the message should be sent reliably
version	yes	double	Indicator of the version of the protocol
type	yes	string	Indicator of the type of message

6 Event Messages

Key	Required	Type	Description
event-type	yes	string	Indicator of the developer-defined type of the event
context	yes	dictionary	Developer-defined data

7 Post Messages

Key	Required	Type	Description
type	yes	string	Indicator of the developer-defined type of the object
data	yes	string	Dictionary of developer-defined data

8 Query Messages

Key	Required	Type	Description
select	no	string	Indicator of the type of object to select
update	no	string	Indicator the type of object to update
delete	no	string	Indicator the type of object to delete
where	no	string	Predicate that specifies objects
data	no	Dictionary	Dictionary of developer-defined data to used to update objects

9 Query Response Messages

Key	Required	Type	Description
type	yes	string	Indicator of the type of the object
data	yes	string	Dictionary of developer-defined data