2016

Fog Computing with Go: A Comparative Study

Ellis H. Butterfield
Claremont McKenna College

Claremont McKenna College

Fog Computing with Go: A Comparative Study

submitted to
Professor Arthur H. Lee

by
Ellis Hiroki Butterfield

for
Senior Thesis
Spring 2016
April 25, 2016

Abstract

The Internet of Things is a recent computing paradigm, defined by networks of highly connected things – sensors, actuators and smart objects – communicating across networks of homes, buildings, vehicles, and even people. The Internet of Things brings with it a host of new problems, from managing security on constrained devices to processing never before seen amounts of data. While cloud computing might be able to keep up with current data processing and computational demands, it is unclear whether it can be extended to the requirements brought forth by Internet of Things.

Fog computing provides an architectural solution to address some of these problems by providing a layer of intermediary nodes within what is called an edge network, separating the local object networks and the Cloud. These edge nodes provide interoperability, real-time interaction, routing, and, if necessary, computational delegation to the Cloud.

This paper attempts to evaluate Go, a distributed systems language developed by Google, in the context of requirements set forth by Fog computing. Similar methodologies of previous literature are simulated and benchmarked against in order to assess the viability of Go in the edge nodes of Fog computing architecture.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The number of objects connected to the Internet is growing at an exponential rate. By 2017, it is predicted that there will be 7 trillion devices serving 7 billion people [2]. These devices will be part of the "Internet of Things" (IoT), the highly connected network of heterogeneous devices, sensors, actuators, and smart objects, communicating between themselves and with humans. These devices introduce a new set of problems. They rely on Internet technologies but are constrained by very different factors than typical computing stations that access worldwide and local networks. Internet of Things is a wide definition that encompasses several layers of hardware and networks, each with unique and interesting problems to be explored.

1.1 Background Information

The definition of IoT is broad, encompassing a variety of contexts, each with unique architectures, devices, and challenges. The idea of IoT however is easily understood within the context of a domain already quite penetrated by the IoT paradigm — the smart home. An example modern-day smart home consists of a large number of smart devices such as smart televisions, smart thermostats, smart lights, and smart security: cameras, smoke detectors, and door locks. These devices will vary in computational complexity and capability. Generally, the simplest types of IoT devices are wireless sensors such as limited connectivity smoke and light detectors. More advanced devices might include smart refrigerators with greater processing power and local network and internet connectivity. Smarter still are devices such as Amazon's Echo [3], which can control lights and switches, order food online, and play music, all via voice commands.

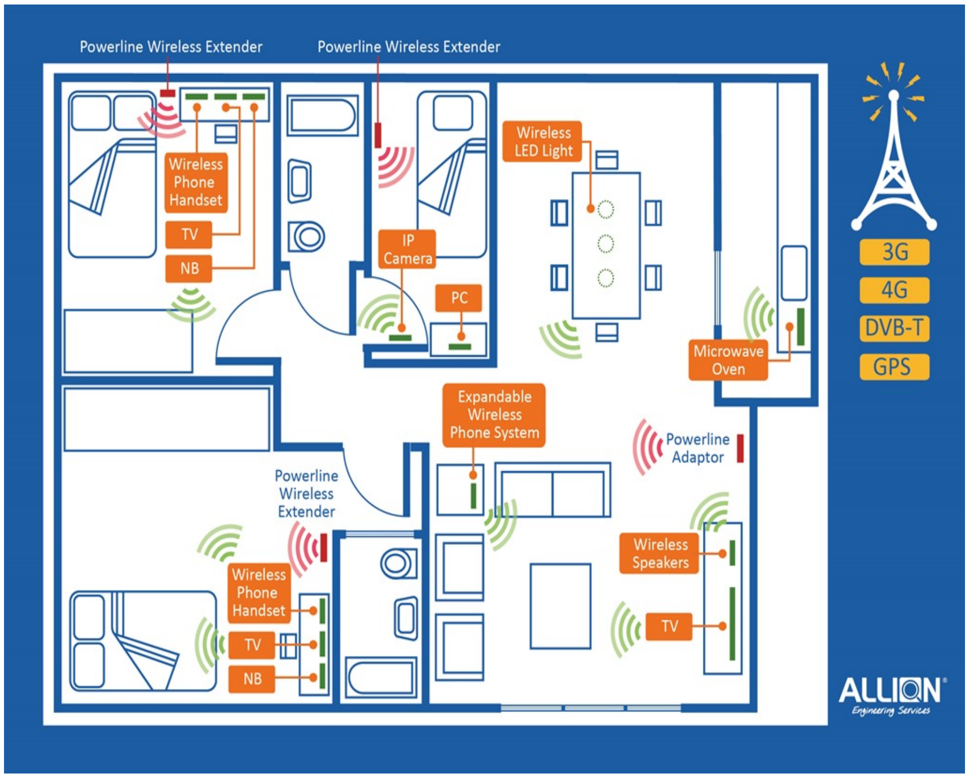

Figure 1.1: The figure above is an example of a smart test home run by Allion Test Labs. Though the example illustrates a smart home primarily connected through WiFi, the type of wireless connections in IoT can also include simple radio, Bluetooth based, cellular connections, and more [4].

1.2 Challenges

The following is a non-exhaustive list of challenges currently facing Internet of Things implementations.

Energy Efficiency: Many IoT devices are not connected to persistent sources of power. Improvements in battery technology are much slower than increases in computational requirements. Smart ways of reducing unnecessary communication and processing hold the key for longer lasting battery life.
Security: This is a broad category that can refer to the security measures of devices themselves, such as tamper resistance, as well as network security, such as securing communication and device identity authentication. Constrained devices lacking in computational or energy resources may not be capable of implementing complex security schemes used in conventional computers and thus new protocols or architectures must be researched [5].
Naming and Identity Management: Standard notions of IP addressing cannot work in the realm of IoT given the huge number of possible devices in a single network [5]. New protocols and methods of identity management must be implemented to handle communication between Wireless Sensor Networks (WSNs) and the Cloud. How can devices be uniquely identified across networks and matched to real or virtual entities?
Interoperability: The large number of heterogeneous devices within a network will necessarily make machine-to-machine (M2M) communication difficult. Protocol standardization becomes especially important in solving the problem of interoperability and will be discussed later in this paper.
Architecture: A standard or reference model architecture for IoT has yet to be developed. The European Lighthouse Integrated Project has attempted to fix this by "providing an architectural reference model for the interoperability for IoT systems, outlining principles and guidelines for the technical design of its protocols, interfaces and algorithms" [6]. These guidelines, though robust, do not appear to be adopted much in mainstream IoT services so far, at least within the United States. Architectural models proposed in literature consist of many multi-layer structures, most often a 5-layer structure, consisting of an objects layer, object abstraction layer, service management layer, application layer, and business layer [7].

1.3 Importance

The Internet of Things is rapidly penetrating everyday life. Large companies such as Microsoft, Apple, Google, and Amazon are quickly entering the IoT market [8] [9] [10] [11]. Many startups are competing in this space, such as Electric Imp [12] and RetailNext [13]. Offerings range from consumer applications such as Google's Nest [14] to professional commercial solutions [15]. Furthermore, the US National Intelligence Council lists Internet of Things in the *Six Technologies with Potential Impacts on US Interests out to 2025* alongside Biogerontechnology, Energy Storage Materials, Biofuels and Bio-Based Chemicals, Clean Coal Technologies, and Service Robotics [16]. Lastly, in their Internet of Things forecast, International Data Corporation predicts the IoT market will be worth \$7.1 trillion dollars by 2020 [17].

This paper evaluates the Go language in the context of a specific architectural implementation of IoT known as Fog computing. I follow an implementation by Cirani et al. [18] based on Cisco research [19] and attempt to show that Go not only performs better, but is also positioned to solve the problem of Fog computing better, due to unique language constructs and the trajectory of its development. The implementation by Cirani et al. uses a Java based implementation of CoAP (Constrained Application Protocol) on a Raspberry Pi Model B single board computer. This paper replicates the methodology with slight modifications and with an implementation simply referred to as "FogPi". Chapter 2 reviews the application and limitations of Fog computing architecture. Chapter 3 provides an overview of the Go language. This section covers design choices, features unique to the language, and why it stands as a desirable choice for Fog computing. Chapter 4 discusses the FogPi architecture and features, the testing implementation, and evaluation. Lastly, Chapter 5 discusses the results and implications for future work.

Chapter 2

Fog Computing

Fog computing is an architecture that extends the paradigm of cloud computing to Internet of Things by placing higher-powered nodes between smart object (SO) networks and the cloud computing and data storage backbones. It is a concept developed by researchers at Cisco [19] defined as “a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centers, typically but not exclusively located at the edge of network.” Fog computing can be thought of as an adaption of edge computing to the IoT context. These edge nodes form an intelligent layer, managing communication, computation, and access between outer smart object nodes, capable of determining whether extra cloud computation is necessary. Fog computing aims to provide a powerful layer for real-time low latency services.

2.1 Characteristics

Cisco defines certain characteristics for Fog computing which distinguish its role as a unique and necessary extension to the cloud computing model [19]. The following are some of the the key features to Fog computing:

- Wide Scale Distribution/Geographic Reach — Fog nodes reside closer to smart objects, playing the roles of routing, computational delegation, communication, and resource access.

- Mobility Support — Nodes within the network are not expected to be geographically static. Device identity is decoupled from location and IP.

- Wireless Access — Nodes are not expected to be hardwired to the Internet and thus rely on wireless networks for communication between smart objects and Fog nodes, as well as between Fog nodes to the Cloud.
- Heterogeneity — Fog nodes are expected to perform in a multitude of differing environments and contexts and will exist in a variety of form factors.
- Interoperability — Fog nodes must be capable of seamlessly communicating and cooperating with a variety of nodes and services.

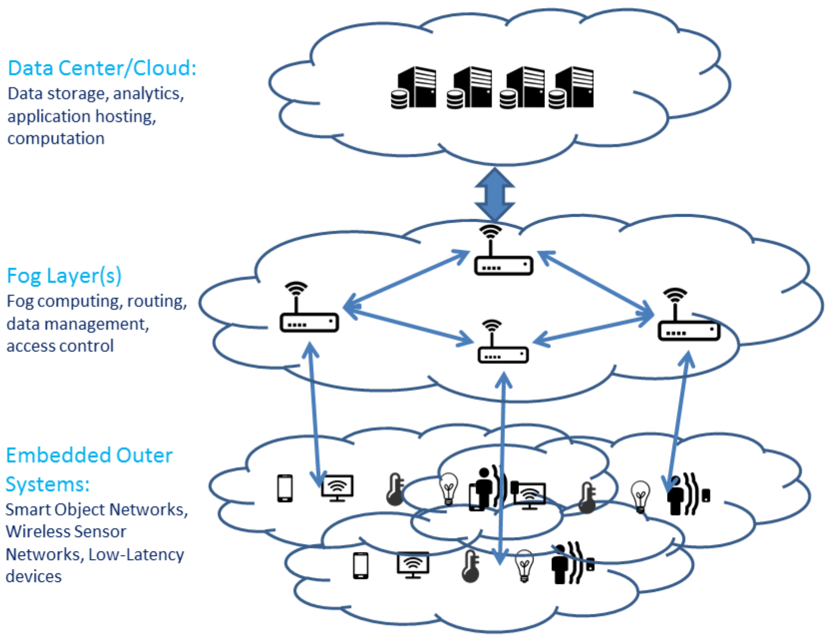
Data Center/Cloud:
Data storage, analytics, application hosting, computation
Fog Layer(s)
Fog computing, routing, data management, access control
Embedded Outer Systems:
Smart Object Networks, Wireless Sensor Networks, Low-Latency devices

Figure 2.1: Fog Computing Architecture

Bonomi et al. [19] list several examples for Fog computing such as a Connected Vehicle, Smart Grid, and WSNs. Environments that have yet to implement Fog computing but might benefit from the paradigm include Thin Server Architecture [20] and Advanced Manufacturing Systems [21] [22].

2.2 Protocol

Fog computing is a description of an architecture that resides at the application and service management layer. The Cisco paper [19] neither implements nor suggests any specific protocol for Fog computing. There are numerous IoT protocols currently in development and use, some more applicable than others to Fog computing. This paper's implementation uses the Constrained Application Protocol for reasons discussed later. However some other potential options and features are listed here.

2.2.1 MQTT - Message Queue Telemetry Transport

MQTT uses a publish/subscribe pattern with a central broker mediating communication and information storage between the two layers. The protocol was initially developed by Andy Stanford Clark and Arlen Nipper in 1999. The message queueing is suitable for unreliable or low bandwidth use. It was originally developed using TCP but has since been adapted to use UDP in MQTT-SN, a sensor network variant [23]. There are several variants of MQTT — RabbitMQ [24], Mosquitto [25], IBM MessageSight [26] — which vary in features and may implement additional features upon the standard.

2.2.2 AMQP - Advanced Message Queuing Protocol

AMQP is an open standard application layer protocol which relies on TCP. Its primary use is providing interoperability between servers. It is message-centric middleware which has been heavily used in the banking industry. It is a robust system which tracks messages and guarantees they get from endpoint to endpoint regardless of node failure [27]. It is heavyweight and more established than other protocols but has potential uses in IoT contexts where message reliability is the primary focus.

2.2.3 DDS - Data Distribution Service

DDS is a protocol developed specifically for M2M communication. It is developed by Object Management Group (OMG) to address the needs of big data applications using a publish/subscribe model. Similar to AMQP, DDS is used in mission critical systems such as banking where reliability and fault tolerance are important. However, DDS is a data-centric model which focuses on making sure the data of any one node is known by all other nodes accurately and quickly. It also supports UDP multicast alongside TCP [28].

2.3 Applicability

The applicability of Fog computing has been investigated by Yanuzzie et al. [29] and Preden et al. [30] These papers analyze Fog computing at a high level, only considering use cases and potential challenges without delving into implementation. More in-depth research has been done by Sakar and Misra [31], in which they develop mathematical models that demonstrate that where large numbers of low-latency devices exist, Fog computing reduces latency as well as energy dissipation due to reduced data transmission. Research into security for this paradigm has explored various theoretical vulnerabilities [32] [33]. However, there has been little research into actual implementation of Fog computing. Cirani et al. [18] explored one such implementation of Fog computing, creating a Fog node dubbed "IoT Hub". Their results demonstrated low memory and processing requirements. The research section of this paper hopes to show an improved approach using the Go language.

Chapter 3

Go

This chapter goes over the history and development behind the Go language. Constructs particular to Go make it uniquely suitable for use in Fog nodes. This chapter covers these various features and why they are particularly suitable for this problem.

3.1 History of Go

The Go programming language was conceived by engineers Robert Griesemer, Rob Pike, and Ken Thompson in 2007. The language is statically typed, compiled, garbage-collected, and provides primitives for concurrency and synchronization. The language was initially developed to deal with problems arising from growing numbers of computational clusters, highly networked systems, and multicore processors. Simply put, Go was developed to make software development for huge systems more productive and scalable [34]. The language was developed with the idea that “less is more” [35]. Go is an attempt at making systems programming more expressive and as a result is more opinionated and restrictive. Despite being developed as a counterpart to the highly verbose and complex C++, Pike states that most users of the language are Python and other dynamically typed, interpreted language users. He posits that those who are accustomed to using C++ don’t care to give up their level of control over features such as memory allocation and pointer arithmetic, whereas Python and Ruby users gain a much faster, concurrent language, without giving up their expectations of expressiveness [35].

3.2 Goroutines

A *goroutine* is a built-in primitive created to aid in concurrency. Goroutines provide the ability to make any function in Go concurrent by simulating thread-like functionality. However, goroutines are far lighter weight than threads, a feat achieved by multiplexing multiple goroutines into one OS level thread[1]. The creation of a goroutine only requires 2 kilobytes of stack space and grows by allocating and freeing heap space when required [36]. An OS level thread for Linux/x86-32 by default starts with a stack size of 2 megabytes [37]. Furthermore, the blocking of a single goroutine does not cause other goroutines within the same thread to block. Goroutines are intended to make threading and concurrent models easier to access and manage than traditional thread creation and management. One caveat of using goroutines is that, unlike traditional threading models, they do not signal when they are complete due to the multiplexing nature of the routines. Go channels are used instead to communicate values within a Go program and between goroutines [38]. The lightweight nature of goroutines is ideal for programs, no matter how small, that have to deal with large amounts of concurrent data processing quickly and efficiently. Fog nodes that have to deal with client-subscriber models and M2M communication will be expected to operate asynchronously in order to be performant. The ability for a programmer to use concurrent constructs such as goroutines not only leads to greater productivity, but also takes much of the memory management for threading outside the direct control of the programmer. Automatic memory management of this type is ideal for dealing with large numbers of unstable connections and delegating routines for connecting, parsing data, and transmitting it.

Goroutines are called by prepending function calls with the `go` keyword. Functions called with the `go` keyword are automatically made into goroutines which shall run concurrently within the process. An example of a goroutine can be seen below [39]:

```
func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}
```

[1]Not to be confused with green threading which involves a virtual machine emulating system level threads

```
func main() {
    go say("world")
    say("hello")
}
```

3.3 Go Channels

Go channels are also primitives built into the language to provide support for concurrency and synchronization. Channels are created by allocating memory for them with the `make` command, similar to C, and defining the datatype the channel passes. Channels are synchronous by default, causing execution to block until both the sender and receiver of the channel are ready. They can be made asynchronous however, by providing a buffer size argument. Channels can be extended to be channels of channels, expanding functionality. A common use is to implement safe parallel demultiplexing [38].

Channels are illustrated in the following example [40]

```
func sum(s []int, c chan int) {
  sum := 0
  for _, v := range s {
  sum += v
  }
  c <- sum // send sum to c
}

func main() {
  s := []int{7, 2, 8, -9, 4, 0}

  c := make(chan int)
  go sum(s[:len(s)/2], c)
  go sum(s[len(s)/2:], c)
  x, y := <-c, <-c // receive from c

  fmt.Println(x, y, x+y)
}
```

Channels are used in a variety of contexts within programs written in Go outside of passing values between goroutines. They can be used as semaphores, as a signaling mechanism to start or stop goroutines, or even as a pool and delegation tool for Job-Worker patterns. Below is a stripped down example from this paper's work for running code at an interval.

```
func refreshResources() {
    ticker := time.NewTicker(time.Second *60)
    for {
        select {
        case <- ticker.C:
            // refresh Resource Directory on channel read
        default:
            // do nothing
        }
    }
}
```

Despite the versatility of Go channels, Go still promotes and uses standard methods of synchronization for shared resources such as mutexes and waitgroups located within their `sync` package [41]. Despite the general applicability of Go channels, much like any other language construct, it is best used in appropriate contexts. Channels are better for passing ownership of data, distributing units of work, and communicating asynchronous results, whereas mutexes are better for protected data situations such as states or caches [42].

3.4 Go Compilation

The Go language is a statically typed, compiled language, despite having expressiveness similar to dynamically typed, interpreted languages such as Python and Ruby. Go binaries are provided for all major operating systems: OSX, Linux, and Windows [43]. Most importantly, Go can be cross-compiled for a multitude of instruction sets and operating systems [44]. The lists provided below are non-exhaustive:

Instruction Sets:

- amd64 (x86-64)

- 386 (x86 or x86-32)
- arm (ARM)
- arm64 (AArch64)
- ppc64, ppc64le (64-bit PowerPC big- and little-endian)

Operating systems:

- FreeBSD
- Windows
- Linux
- OSX (Darwin)
- Solaris

The large support for cross-compilation and development environments for Go is paramount to creating fast, compatible programs for the large number of heterogeneous systems within Internet of Things. Outside of wireless sensors and highly constrained smart objects, Internet of Things objects will likely support modified operating systems. For the purposes of this project, support for the `arm` instruction set is of particular importance for Raspberry Pi OS, a trimmed down Debian system. Windows 10 [45] and Linux variants on "things" are supported as well. Lastly, one of the central drivers behind Go's development was the concept of fast compile times [34]. Go was constructed such that dependency analysis for programs is easy, avoiding the use of header files, a time-consuming procedure for C and C++ [46]. This feature is paramount for distribution of code on large numbers of devices. To be able to quickly compile code natively for a large number of instruction sets makes Go incredibly versatile, eliminating the need for a virtual machine or interpreter.

3.5 Go Packaging

Go's development as a highly networked systems language means that it provides many well developed packages related to networking and I/O. The `net` package provides an interface for network I/O with TCP and UDP, domain name resolution, and Unix sockets [47]. Within this package exists a variety of incredibly useful subpackages including, but not limited to, `http`[2],

[2]HTTP client and server implementations

`cgi`[3], `pprof`[4], `rpc`[5], and `url`[6]. Go was developed as an open-source language as well with 118,742 packages accessible remotely online via github.com [7]. During compilation, any remote packages not contained on the device can be downloaded automatically and built via the go `get` command. Once built, these packages can be referred to by their remote url.

```
// Gets and builds remote package
go get github.com/golang/example/stringutil

// Example remote package import statement
import "github.com/golang/example/stringutil"
```

3.6 Other Go Features

3.6.1 First Class Functions

Functions are first class objects [8]. Those coming from a JavaScript background will be accustomed to callback style programming available in Go. Surveys conducted by StackOverflow, one of the most popular programming Q&A websites, presented JavaScript as the most popular language in the world [48]. Furthermore, Node.js[9], a JavaScript runtime used for web server technology, is a very popular event-driven, non-blocking IO program built on Google's Chrome V8 engine. The ability to attract developers with similar expressiveness and functionality [49] is important to cementing Go as a language in the IoT domain.

3.6.2 Automatic Memory Management

When dealing with concurrency models and large numbers of references to possible objects, memory management becomes difficult to handle and debug. Go garbage collection (GC) in early stages of development was not very good, often exhibiting large pauses and spikes during cleanup. Garbage collection within Go is particularly challenging for the language's developers

[3]Implements Common Gateway Interface

[4]HTTP server runtime profiling data

[5]Remote Procedure Calls

[6]URL parsing and query escaping

[7]This is the number of repositories as of April 14th, 2016 via the search "language: Go"

[8]Functions can be passed as arguments to other functions

[9]Already used in the IoT context by high-powered startup Electric Imp

because, unlike languages like Java which might use 10 threads and synchronize with locks, Go GC has to somehow work in the presence of thousands of goroutines and channels [50]. Since then, Go has massively improved its GC [51]. It now employs a *concurrent, tri-color, mark-sweep collector*, a GC system quite different than conventional enterprise algorithms, but better suited to the unique nature of Go [52].

3.6.3 Interfaces

Go utilizes interfaces rather than class type systems common to traditional Object Oriented (OO) languages. Go supports structs and some features of Object-Oriented design, but lacks the notion of classes and type hierarchy. Interfaces are lightweight to implement and reduces the tracking of relationships between types common in OO languages, reducing compilation and runtime complexity. A Go interface is a set of methods as well as a type. Thus, types can satisfy many interfaces at once, akin to multiple inheritance [38]. Lacking traditional notions of classes and subtyping is actually quite valuable in the context of IoT. Given the potentially large heterogeneity of objects, representing objects in code using traditional classes would prove cumbersome and interfaces would be largely used anyway. Interfaces can model devices simply as they create abstractions which consider the functionality common between datatypes rather than fields common between datatypes.

Chapter 4

Architecture, Implementation, and Evaluation

4.1 Architecture

The design choices for the Raspeberry Pi Fog node for the purposes of this paper attempted to recreate functionality analogous to the study in comparison by Cirani et al. [18]. This section will discuss the design of the Fog node and how Go fits within these paradigms.

This specific implementation of Fog computing utilizes the CoAP protocol, "a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks", designed by the IETF (Internet Engineering Task Force) [1]. The primary design is for M2M applications, many of which fall directly into the Fog computing model. Important features of CoAP include [53]:

- Fulfillment of M2M requirements for networks with constrained nodes via CoAP-to-CoAP (C2C).
- User Datagram Protocol (UDP) — UDP is smaller than Transmission Control Protocol (TCP) and easier to send and manage for smaller nodes. The total size of each message is intended to fit within one IP packet, though some support for multi-packet messages exists [54].
- Asynchronous message passing.
- Small message overhead — Headers are only 4 bytes.

- Support for URI paths and querying.
- Basic caching and proxy support — Cross-protocol proxy support provides basic mapping from HTTP requests to CoAP requests (H2C) and vice-versa through an intermediary. Caching allows an intermediary node to store values and limit unnecessary communication with devices so as not to tax resource constrained devices and limit network traffic.
- Resource and Service Discovery — CoAP dictates methods for service discovery, requiring a default `coap` URI port to be open on CoAP-compatible servers, as well as multicast for endpoint discovery where endpoints are SOs and CoAP nodes.
- DTLS Security — A security protocol between clients and servers that allows UDP messages to be sent over a secured connection.

Fog Computing	**CoAP**
Low latency	UDP and simple messaging model [55]
Access point and proxying	C2C and H2C proxying [56]
Management of large-scale sensor networks	Resource Directory [57]
Real-time interactions	Request/Response Semantics [58]
Heterogeneity	CoRE Link Format [59]
Interoperability	Uniform `coap` URI [60]
Subscriber Models	Resource Observation [61]

Table 4.1: Fog Characteristics and CoAP

The interface for CoAP is modeled after Internet HTTP requests. Furthermore, it follows the REST (Representational State Transfer) paradigm common to web development, but adapted to fit the constraints of weaker smart objects. Like REST, CoAP implements GET, PUT, POST, and DELETE methods which function similarly to their HTTP counterparts. However, CoAP messages differ drastically from HTTP messages as seen in Figure 4.1.

Despite using UDP, CoAP has similar functionality to TCP to deal with packet loss and general reliability. CoAP messages can be considered confirmable or non-confirmable, dictating whether or not they need acknowledgments. As an example, the communication between a Fog node and a switch node will need confirmation to ensure proper functionality whereas receiving messages from location sensors might be non-confirmable, reducing communication for the weaker sensor. The request-response model also supports message piggybacking, which means responses can included on top of acknowledgment messages [58].

```
 0                   1                   2                   3
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |Ver| T |  TKL  |      Code     |          Message ID           |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |   Token (if any, TKL bytes) ...
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |   Options (if any) ...
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |1 1 1 1 1 1 1 1|    Payload (if any) ...
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.1: CoAP Message Format [1]

The Token field is used to match responses with requests. The Options portion of the payload is 0 or more option codes contained in both response and request messages containing information such as Max-Age, Uri-Path, and Content-Format.

4.2 Implementation

The research for this paper used code adapted from a bare-bones library by Dustin Sallings [62]. The library itself provided message parsing as well as basic server and client capabilities. Proxying, caching, and a variety of other features were implemented on top of this library for the purposes of this paper.

FogPi was installed and run on a Raspberry Pi Model B v1.1 revision a21041. Due to cost constraints, an actual test-bed of live devices was not used, and was instead simulated. The following describes the setup for this experiment:

1. A Windows 10 laptop running with an Intel Core i5-4200 CPU @ 1.60 GHz was connected to FogPi over a 1Gbps Ethernet connection. Wireless connection was unavailable during testing but unnecessary to test processing and memory usage.

2. A mock smart object server was deployed on the laptop to simulate the 100 smart objects within the Cirani et al. live test bed [18]. The differences in capabilities were simulated via the type of messages and frequency with which they were sent.

3. A client program was run on the laptop from which HTTP requests could ping FogPi for information about the SOs within FogPi's "network".

4. The FogPi code was copied to the Raspberry Pi unit and started manually for the purposes of testing. On run, it would immediately begin accepting messages from the client and smart object server as well sending responses.

5. Data was collected over a one hour interval. CPU profiling and memory information was collected, utilizing built-in tools to the language, namely pprof [63].

The Fog node created for the purposes of this study implemented border router, resource directory, caching, and C2C/H2C proxy functions, analogous to the Cirani et al. study [18]. It did not include origin server or resource discovery functionality.

4.3 Evaluation

The code for FogPi was written on a Windows machine and then cross compiled to the `arm` architecture for Linux systems using the command `env GOOS=linux GOARCH=arm GOARM=7 go build server.go`. The program was copied over to the Raspberry Pi unit via ssh. The size of the binary on compilation was approximately 6.3MB. For comparison, the Californium Java binary alone, a library used by Cirani et al., is 4.93MB. For devices with constrained memory, this size may prove unfeasible.

The refresh interval noted in Cirani et al. was simulated with 100 requests to the smart object server every 60 seconds for each object in the resource directory. Furthermore, every second, a request from the client server pinged FogPi. During testing, FogPi memory usage fluctuated between 0.33MB and 0.95MB. CPU usage peaked at approximately 50%, though averaged around 38%. This peak usage came from the program's management of goroutines on initialization and when the stack was increased, a fairly expensive but infrequent operation. Cirani et al. experienced approximately 26MB of heap memory usage with an average of 5% CPU usage with peaks around 35% [18].

FogPi clearly outperformed this implementation in terms of heap memory usage but used quite a bit more computational resources. This can be attributed to several factors. Firstly, our study had clients ping for an entire Resource Directory response every second. It is unclear what the request-response rates were in the Cirani et al. study but it was likely considerably less than a per second basis. Secondly, I suspect that as a result of being a distributed systems language, Go attempts to maximize the amount of CPU being used with a mindset similar to Erlang as echoed by this quote from *Programming Erlang* [64].

> Use Lots of Processes. This is important – we have to keep the CPUs busy. All the CPUs must be busy all the time. The easiest way to achieve this is to have lots of processes. When I say lots of processes, I mean lots in relation to the number of CPUs. If we have lots of processes, then we won't need to worry about keeping the CPUs busy.

The use of many goroutines in FogPi was unoptimized and perhaps taxed the system more than it might have needed to. These findings are echoed by open source benchmarks providers where Go CPU loads tend to be slightly higher but memory usage lower compared to similar Java programs [65].

Though Go was never installed on the Raspberry Pi device, the binary size for Go 1.6.1, the most recent release [43], is only 67 MB. The smallest version of the embedded Java SE platform as of writing is 102.21MB, almost double the size [66]. However, it is likely that one would only install the Java virtual machine on embedded devices; in which case the binary size is about 3.5MB [67], slightly smaller than the total compiled FogPi binary.

Chapter 5

Discussion

In this paper, I have presented a Fog node implementation in Go, run on a similar platform as presented by Cirani et al. [18]. The results of my experiment have shown that Go is suitable for this domain if not more competitive than Java. The Fog node that was developed for the purposes of this paper has a lot of room for improvement. Origin server and resource discovery, both pivotal parts of CoAP, were left out of FogPi in this version. Future work will expand on creating a more robust node for greater analysis of Go in the constrained IoT context.

For purposes of analyzing the benefits of Go, certain benchmark tests could have been expanded to include the broader requirements of Fog computing outside the comparative benchmarks done within this paper. Notably, in a Fog computing solution for interfacing mobile device clouds by Shi et al. [53], the Erlang language was used for their CoAP server due to their dissatisfaction with performance of C-, Python-, and Java-based CoAP solutions for Raspberry Pi. Within the study, throughput, average round trip time, and timeout probability for clients was measured on their CoAP server. Erlang is language with similar characteristics to Go: concurrent, garbage collected, compiled, and developed with distributed systems in mind [68]. For a truly comparative study, I believe that comparing similar performance metrics for FogPi against Shi et al. would place Go's computational performance and concurrency handling in a better context. One critique of Shi et al. is the lack of any metrics for memory usage. Fog computing edge nodes are expected to be more performant than the smart object network that they manage. However, these edge nodes are still expected to be relatively constrained. Even a Raspberry Pi, considered a moderately powerful IoT device, has far less RAM and total storage than conventional computers [69].

Neither Cirani et al. [18], Shi et al. [53], nor this study, were adequate in testing major components of Fog computing, specifically mobility support and its role as an intelligent layer. For mobility support, protocols such as LISP [70] or HIP [71] should have been considered and implemented to decouple smart object identity from location. Future works should expect to validate a Fog node's ability to handle large amounts of requests in conjunction to providing a layer of computation. Fog computing is intended to reduce computational load as well as latency. In the context of these studies, adequate testing might be to run vector or matrix computation, mimicking the data structures of location sensor data. Processing coordinate data will provide a reasonable use case and evaluate language level computational abilities. In this role, there is evidence that Go outperforms Java due to its goroutine functionality [72]. Computation of this sort brings into question Fog computing priorities. On one hand, event-driven architecture provided by concurrent languages makes sense given the nature of sensor readings and the request-response model. However, in many cases, computation required for sensor data falls into regular parallelism[1] [73]. In this case, the proper language might fall into the class of parallel languages such as *Cilk* [74] or *Chapel* [75].

Go is an incredibly powerful language. Its concurrent primitives make adding concurrency to programs seamless. Above all, the language is more expressive than other high performance compiled languages. Another thing to consider is that Go is a relatively new language, having had its first stable release in 2012 [76]. For a language less than 5 years old, its performance is near top tier languages with years of technical investment.

In the broader context of IoT, Go is definitively a solid choice for middleware IoT devices. Go is a fast-moving language with major updates released on a 6 month average cycle that come with huge performance gains. I expect that Go will become a major language in IoT and Fog computing in the near future for its performance, memory usage, and its ease of development.

[1]A problem where it is easy to find independent sub-tasks

Bibliography

[1] Z. Shelby, K. Hartke, C. Bormann, and B. Frank, "Rfc 7252," 2014.

[2] S. Babar, P. Mahalle, A. Stango, N. Prasad, and R. Prasad, "Proposed security model and threat taxonomy for the internet of things (iot)," in *Recent Trends in Network Security and Applications*, pp. 420–429, Springer, 2010.

[3] Wikipedia, "Amazon echo." `https://en.wikipedia.org/wiki/amazon_echo`, 2016.

[4] Technical Direct, "Allion smart home test environment," 2015.

[5] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[6] "Mission iot-a: Internet of things architecture." `http://www.iot-a.eu/public/introduction/missioncollage`.

[7] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: the internet of things architecture, possible applications and key challenges," in *Frontiers of Information Technology (FIT), 2012 10th International Conference on*, pp. 257–260, IEEE, 2012.

[8] "Tap into the internet of your things with azure iot suite." `http://www.microsoft.com/en-us/server-cloud/internet-of-things/azure-iot-suite.aspx`.

[9] "AWS IoT - Amazon Web Services." `https://aws.amazon.com/iot/`.

[10] "Internet of things." `https://cloud.google.com/solutions/iot/`.

[11] "Homekit." `http://www.apple.com/ios/homekit/`.

[12] Electric Imp, “Connectivity that transforms.” `http://www.electricimp.com/`.

[13] RetailNext, “Analytics for any retail professional.” `http://retailnext.net/`.

[14] Nest, “Home.” `https://nest.com/`.

[15] PTC, “ IoT Solutions .” `http://www.ptc.com/internet-of-things/solutions`.

[16] N. NIC, “Disruptive civil technologies: Six technologies with potential impacts on us interests out to 2025,” 2008.

[17] D. Lund, C. MacGillivray, V. Turner, and M. Morales, “Worldwide and regional internet of things (iot) 2014–2020 forecast: A virtuous circle of proven value and demand,” *International Data Corporation (IDC), Tech. Rep*, 2014.

[18] S. Cirani, G. Ferrari, N. Iotti, and M. Picone, “The iot hub: a fog node for seamless management of heterogeneous connected smart objects,” in *Sensing, Communication, and Networking-Workshops (SECON Workshops), 2015 12th Annual IEEE International Conference on*, pp. 1–6, IEEE, 2015.

[19] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, ACM, 2012.

[20] M. Kovatsch, S. Mayer, and B. Ostermaier, “Moving application logic from the firmware to the cloud: Towards the thin server architecture for the internet of things,” in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, pp. 751–756, IEEE, 2012.

[21] F. Tao, Y. Cheng, L. Da Xu, L. Zhang, and B. H. Li, “Cciot-cmfg: cloud computing and internet of things-based cloud manufacturing service system,” *Industrial Informatics, IEEE Transactions on*, vol. 10, no. 2, pp. 1435–1442, 2014.

[22] Z. Bi, L. Da Xu, and C. Wang, “Internet of things for enterprise systems of modern manufacturing,” *Industrial Informatics, IEEE Transactions on*, vol. 10, no. 2, pp. 1537–1546, 2014.

[23] "Mqtt." http://mqtt.org/.

[24] "Rabbitmq - messaging that just works." https://www.rabbitmq.com/.

[25] "An open source mqtt v3.1/v3.1.1 broker." http://mosquitto.org/.

[26] "Secure gateway to the internet of things and high-performance mobile messaging." http://www-03.ibm.com/software/products/en/messagesight.

[27] "Advanced message queuing protocol." https://www.amqp.org/.

[28] "The proven data connectivity standard for the iot." http://portals.omg.org/dds/.

[29] M. Yannuzzi, R. Milito, R. Serral-Gracià, D. Montero, and M. Nemirovsky, "Key ingredients in an iot recipe: Fog computing, cloud computing, and more fog computing," in *Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2014 IEEE 19th International Workshop on*, pp. 325–329, IEEE, 2014.

[30] J. Preden, J. Kaugerand, E. Suurjaak, S. Astapov, L. Motus, and R. Pahtma, "Data to decision: pushing situational information needs to the edge of the network," in *Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2015 IEEE International Inter-Disciplinary Conference on*, pp. 158–164, IEEE, 2015.

[31] S. Sarkar and S. Misra, "Theoretical modelling of fog computing: a green computing paradigm to support iot applications," *IET Networks*, vol. 5, no. 2, pp. 23–29, 2016.

[32] I. Stojmenovic and S. Wen, "The fog computing paradigm: Scenarios and security issues," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, pp. 1–8, IEEE, 2014.

[33] S. J. Stolfo, M. B. Salem, and A. D. Keromytis, "Fog computing: Mitigating insider data theft attacks in the cloud," in *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, pp. 125–128, IEEE, 2012.

[34] R. Pike, "Go at google: Language design in the service of software engineering." `http://talks.golang.org/2012/splash.article`.

[35] R. Pike, "Less is exponentially more." `http://commandcenter.blogspot.com.au/2012/06/less-is-exponentially-more.html`, Jun 2012.

[36] "Go 1.4 release notes." `https://golang.org/doc/go1.4#runtime`.

[37] "pthread_create(3) - Linux manual page ." `http://man7.org/linux/man-pages/man3/pthread_create.3.html`.

[38] "Effective go." `https://golang.org/doc/effective_go.html`.

[39] "A tour of go." `https://tour.golang.org/concurrency/1`.

[40] "A tour of go." `https://tour.golang.org/concurrency/2`.

[41] "Package sync." `https://golang.org/pkg/sync/`.

[42] R. Beton, "Mutexorchannel." `https://github.com/golang/go/wiki/mutexorchannel`.

[43] "Downloads." `https://golang.org/dl/`.

[44] "Installing go from source." `https://golang.org/doc/install/source#environment`.

[45] "The internet of your things." `https://developer.microsoft.com/en-us/windows/iot`.

[46] "Frequently asked questions (faq)." `https://golang.org/doc/faq`.

[47] "Package net." `https://golang.org/pkg/net/`.

[48] "Stack overflow developer survey 2015." `http://stackoverflow.com/research/developer-survey-2015`.

[49] L. D. Paulson, "Developers shift to dynamic programming languages," *Computer*, vol. 40, no. 2, pp. 12–15, 2007.

[50] R. Hudson, "Go gc: Latency problem." `https://talks.golang.org/2015/go-gc.pdf`, 2015.

[51] M. Kevac, "Go gc times from 1.4 to 1.5 (tip)." `https://twitter.com/mkevac/status/620872308446625792/photo/1`.

[52] R. Hudson, "Go gc: Prioritizing low latency and simplicity." `https://blog.golang.org/go15gc`.

[53] H. Shi, N. Chen, and R. Deters, "Combining mobile and fog computing: Using coap to link mobile device clouds with fog computing," in *2015 IEEE International Conference on Data Science and Data Intensive Systems*, pp. 564–571, IEEE, 2015.

[54] C. Bormann, "Block-wise transfers in CoAP draft-ietf-core-block-17 Work in Progress." `https://tools.ietf.org/html/draft-ietf-core-block-17`.

[55] Z. Shelby, K. Hartke, and C. Bormann, "Messaging Model ." `https://tools.ietf.org/html/rfc7252\#section-2.1`.

[56] Z. Shelby, K. Hartke, and C. Bormann, "Cross-protocol proxying between coap and http." `https://tools.ietf.org/html/rfc7252\#section-10`.

[57] C. Bormann, "CoRE Resource Directory draft-ietf-core-resource-directory-07." `https://tools.ietf.org/html/draft-ietf-core-resource-directory-07`.

[58] Z. Shelby, K. Hartke, and C. Bormann, "Request/Response Semantics." `https://tools.ietf.org/html/rfc7252\#section-5`.

[59] Z. Shelby, "Constrained RESTful Environments (CoRE) Link Format." `https://tools.ietf.org/html/rfc6690`.

[60] Z. Shelby, K. Hartke, and C. Bormann, "CoAP URIs." `https://tools.ietf.org/html/rfc7252\#section-6`.

[61] K. Hartke, "Observing Resources in CoAP draft-ietf-core-observe-08 Work in Progress." `https://tools.ietf.org/html/draft-ietf-core-observe-08`.

[62] D. Sallings, "go-coap." `https://github.com/dustin/go-coap`, 2012.

[63] "Package pprof." `https://golang.org/pkg/pprof/`.

[64] J. Armstrong, *Programming Erlang: software for a concurrent world.* Pragmatic Bookshelf, 2007.

[65] The Computer Language Benchmarks Game, "Java programs versus Go." http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=java.

[66] "Oracle java se embedded downloads." http://www.oracle.com/technetwork/java/embedded/embedded-se/downloads/index.html.

[67] "Oracle java me embedded downloads." http://www.oracle.com/technetwork/java/embedded/javame/embed-me/downloads/java-embedded-java-me-download-2162242.html.

[68] "Build massively scalable soft real-time systems." https://www.erlang.org/.

[69] "Raspberry Pi 2 Model B." https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.

[70] "Lisp overview...." http://lisp.cisco.com/lisp_over.html.

[71] R. Moskowitz, T. Heer, P. Jokela, and T. Henderson, "Host identity protocol version 2 (hipv2)," tech. rep., 2015.

[72] N. Togashi and V. Klyuev, "Concurrency in go and java: Performance analysis," in *Information Science and Technology (ICIST), 2014 4th IEEE International Conference on*, pp. 213–216, IEEE, 2014.

[73] C. Jardak, J. Riihijärvi, F. Oldewurtel, and P. Mähönen, "Parallel processing of data from very large-scale wireless sensor networks," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 787–794, ACM, 2010.

[74] "Cilkplus." https://www.cilkplus.org/.

[75] "The chapel parallel programming language." http://chapel.cray.com/.

[76] "Release history." https://golang.org/doc/devel/release.html.