IOTA: INTERNET OF THINGS ASSISTANT

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

Brandon Okumura

July 2017

© 2017
Brandon Okumura
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	IoTA: Internet of Things Assistant
AUTHOR:	Brandon Okumura
DATE SUBMITTED:	July 2017
COMMITTEE CHAIR:	Foaad Khosmood, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Seng, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Franz Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

IoTA: Internet of Things Assistant

Brandon Okumura

The Internet of Things is the networking of electronic devices, or "Things", that enables them to collect and share data, as well as interact with their physical surroundings. Analyzing this collected data allows us to make smarter economic decisions. These interconnected networks are usually driven by low-powered micro-controllers or cheap CPUs that are designed to function optimally with very little hardware. As scale and computational requirements increase, these micro-controllers are unable to grow without being physically replaced.

This thesis proposes a system, IoTA, that assists the Internet of Things by providing a shared computational resource for endpoint devices. This solution extends the functionality of endpoint devices without the need of physical replacement. The IoTA system is designed to be easily integrable to any existing IoT network.

This system presents a model that allows for seamless processing of jobs submitted by endpoint devices while keeping scalability and flexibility in mind. Additionally, IoTA is built on top of existing IoT protocols. Evaluation shows there is a significant performance benefit in processing computationally heavy algorithms on the IoTA system as compared to processing them locally on the endpoint devices themselves.

ACKNOWLEDGMENTS

Thanks to:

- Professor Foaad for his guidance and good advice.
- My Mom for supporting me throughout my college career.
- The entire Cal Poly Computer Science Department.

In Memory of:

- My Father, Michael.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER

1 Introduction 1

1.1 Description of the Problem 1

1.2 Overview of the Solution 2

1.3 Outline of the Thesis 2

2 Background 4

2.1 The Internet of Things 4

2.1.1 IoT Endpoint Devices 5

2.1.1.1 Power 5

2.1.1.2 Network Capability 5

2.1.1.3 Cost Effectiveness 5

2.1.2 Difficulties in IoT 6

2.2 FOG Computing 6

2.2.1 Benefits of FOG Computing 7

2.3 IoT Communication Standards 7

2.3.1 CoAP 8

2.3.1.1 CoAP Model 8

2.3.2 MQTT 9

2.3.2.1 MQTT Model 9

2.3.3 CoAP or MQTT? 9

2.4 Docker 10

3 Related Works 11

3.1 The Fog Computing Paradigm: Scenarios and Security Issues 11

3.2 Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing 12

3.3 Fog Computing: A Platform for Internet of Things and Analytics . . 13

3.4 Mobile Fog: A Programming Model for LargeScale Applications on the Internet of Things . . . 13
3.5 Processor Offloading in Cell Phones . . . 14
3.6 Offloading Benefits . . . 15
4 Design . . . 16
4.1 Goals . . . 16
4.2 Requirements . . . 16
4.2.1 Server Deployment and Maintenance . . . 17
4.2.2 Scalability . . . 17
4.2.3 Security . . . 18
4.2.4 Client Deployment . . . 18
4.2.5 Client Configurability . . . 19
5 Implementation . . . 20
5.1 Overview . . . 20
5.2 Server . . . 22
5.2.1 Server Requirements . . . 22
5.2.1.1 Hardware Requirements . . . 22
5.2.1.2 Software Requirements . . . 23
5.2.2 Architecture . . . 23
5.2.3 Requests . . . 23
5.2.3.1 POST . . . 24
5.2.3.2 PUT . . . 25
5.2.3.3 GET . . . 27
5.2.3.4 DELETE . . . 29
5.2.4 Database Architecture . . . 29
5.2.5 Module Overview . . . 32
5.2.5.1 Server . . . 32
5.2.5.2 Utils . . . 32
5.2.5.3 Machine . . . 32
5.2.5.4 IoTDB . . . 33
5.2.5.5 Command . . . 33
5.2.5.6 coapExceptions . . . 33

5.3 Client 33
5.3.1 Architecture 34
5.3.2 Config 34
5.3.3 Module Overview 34
5.3.3.1 IoTClient 35
6 System Evaluation 36
6.0.1 Performance Measurements 36
6.0.2 Accuracy 37
6.1 Results 37
6.1.1 IoTA Breakdown 41
7 Conclusion 43
8 Future Work 44
BIBLIOGRAPHY 46
APPENDICES
A IoTA Server Setup 48
A.1 Dependencies 48
A.2 Required Python Libraries (Not in Python's stdlib) 48
A.3 Running the Server 48
B Docker Commands 49
B.1 List all Docker Instances 49
B.2 Run a Command in Docker Instance 49
B.3 Copy File To Docker Container 49
B.4 Copy File From Docker Container 49
B.5 Get Container's Log Files 49
B.6 Portainer 49

LIST OF TABLES

Table Page

6.1 Matrix Results Table 40

6.2 Primality Results Table 40

6.3 Matrix Multiplication Breakdown Table 42

6.4 Primality Breakdown Table 42

LIST OF FIGURES

Figure		Page
5.1	An overview of the IoTA System	21
5.2	An overview of the server	24
5.3	An example of a POST Request	25
5.4	Network Diagram of a POST Request	26
5.5	An example of a PUT Request	27
5.6	Network Diagram of a PUT Request	28
5.7	An example of a GET Request	28
5.8	Network Diagram of a GET Request	29
5.9	An example of a DELETE Request	30
5.10	Network Diagram of a DELETE Request	30
6.1	Performance Benefit of using IoTA	38
6.2	Matrix Multiplication Runtime Results	39
6.3	Primality Runtime Results	39

Chapter 1

INTRODUCTION

"When wireless is perfectly applied the whole earth will be converted into a huge brain, which in fact it is, all things being particles of a real and rhythmic whole. We shall be able to communicate with one another instantly, irrespective of distance." -Nikola Tesla 1926 [6]

The invention of the Internet has enabled people to communicate and share data quickly across the globe. This gives the ability to access data that was not widely available in the past. The Internet of Things (IoT) aims to unlock even more data by connecting "things", or everyday objects, to the Internet. This next evolutionary step of the Internet allows for the collection and analysis of even more data, leading to a more connected and automated world.

1.1 Description of the Problem

Thanks to the recent advancements of certain technologies in the past couple of decades, the Internet of Things has been able to expand exponentially. These advancements include the expansion of wireless connectivity, better batteries, faster and more reliable Internet, cheaper hardware, as well as the popularity of the ARM based chip. This growth in the field has lead to a multitude of consumer endpoint devices to be created and manufactured.

These IoT endpoint devices are created to be optimally efficient and lightweight, while still able to function with decent performance. This, in turn, causes endpoint devices to be computationally weak but energy efficient. A problem arises when trying to add functionality to existing devices. The addition of computationally

heavier algorithms such as machine learning or image processing means they are not able to run effectively on constrained hardware. Existing devices would require better hardware or their processes will suffer from long turn-around times. The most common approach to this problem is to replace all endpoint devices with better hardware. This waste of endpoint devices can add up in cost for a connected business or connected smart home.

1.2 Overview of the Solution

The goal of this thesis is to present an in-depth implementation and evaluation of a scalable and practical solution to extend the functionality of current IoT devices without the need of replacing the hardware of all endpoint devices. This is done by creating a generalized computational unit, IoTA, that is specifically designed to assist the Internet of Things.

The contribution of this paper is to design and build a server and client using existing IoT standards that allows for seamless processing of jobs submitted by endpoint devices, effectively offloading the work to a shared computational resource. The final product will be a server to process these requests, as well as an API (Application Programming Interface) for IoT developers to utilize additional computational power if there is an available server. This model enables the Internet of Things to continuously expand without being constantly constrained by endpoint device hardware.

1.3 Outline of the Thesis

We discuss the background of fog computing in chapter 2. In chapter 3, we explore related works. Chapter 4 discusses the design goals of the IoTA system, while chapter 5 describes the implementation of this project. Chapter 6 presents the experiments

used to validate the IoTA system's performance. Chapter 7 concludes with a summary of our contributions. Lastly, chapter 8 concludes with potential future work opportunities.

Chapter 2

BACKGROUND

This chapter provides technical background research in areas related to the Internet of Things. It provides a brief description of concepts and utilized technologies critical to understanding the IoTA system.

2.1 The Internet of Things

The Internet of Things is the concept of networking everyday devices to enable devices to collect and share information between themselves. This information can be used to make smarter decisions by analyzing and reacting to this new data. This new form of computing has the potential to revolutionize the world by creating a better user experience. Consider the following example in a connected smart home: When your alarm clock goes off, it is able to communicate with other devices. It can tell your bedroom lights to turn on, or your coffee maker to start brewing a fresh pot, and even tell your shower to start. Inter-device communication automates and streamlines your daily morning routine. The Internet of Things enables devices to work together to achieve a common goal.

The Internet of Things has already expanded greatly in the last decade. IoT can already be found in multiple industries, such as heath care, agriculture, energy, transportation, and home automation. Current IoT solutions are built on the concept of cloud computing.

2.1.1 IoT Endpoint Devices

Endpoint devices are any networked device that includes sensors or actuators to interact with it's environment. These devices can be designed to function in a wide array of environments. Therefore, when developing devices for the Internet of Things, we must consider a few different factors that impact the device's functionality and design:

2.1.1.1 Power

All IoT devices need access to power. Unfortunately, running power distribution lines to all devices can be inconvenient and unnecessary, especially at scale. The most common approach is to design the endpoint device to use a battery. Using a battery can let the device run in more environments, but users will have to keep maintenance costs in mind.

2.1.1.2 Network Capability

The Internet of Things relies on the ability to communicate with other devices through the Internet. However, not all environments have access to a stable wifi connection, or may utilize a very low bandwidth network. As a result, some protocols may unnecessarily flood limited networks.

2.1.1.3 Cost Effectiveness

Endpoint devices are designed to be deployed in volume. Naturally, this requires the devices to be cost efficient as expensive devices would be impractical. To achieve cost effectiveness, IoT devices must be designed carefully to limit the hardware so it may function at an acceptable level while not being too overqualified for its task at hand.

Understanding the use cases of these endpoint devices becomes crucial in designing efficient and cost effective hardware. Since most of the end point devices are designed to run in constrained environments, they are designed to be lightweight embedded devices.

2.1.2 Difficulties in IoT

In the past decade, the Internet of Things has grown exponentially, and is predicted to continue this growth pattern for the foreseeable future [14]. Unfortunately, the rapid expansion of IoT has lead to competing standards and non-scalable solutions.

For example, current IoT market solutions utilize cloud computing for data management. However, cloud computing is not always optimal, especially in constrained network applications. As an alternate approach, we will look into FOG computing in section 2.2. In addition, since IoTA is built using pre-existing IoT technologies, we will explain our choices in using the COAP protocol over MQTT in section 2.3.

2.2 FOG Computing

FOG Computing, also known as edge computing or fogging, is a term coined by Cisco [5]. This computational model aims to improve cloud computing by bringing the computational intelligence closer to where the data is being collected. FOG computing is not a replacement for cloud computing, but instead extends the cloud for improved performance.

2.2.1 Benefits of FOG Computing

In the current market solution of utilizing cloud computing, end point devices need to communicate with servers outside of the local network. This communication model suffers from a reduced quality of service (QOS) due to network latency. In real time applications, high network latency from the endpoint device to the cloud can greatly affect the end product. By utilizing FOG computing, endpoint devices would communicate with edge network devices, which may allow an application to complete faster, resulting in an overall better QOS.

In addition, FOG computing reduces the amount of data being sent to the cloud. Instead of sending all of the data collected by many endpoint devices, FOG computing servers allow users to condense and analyze data before sending it out to external networks. This solves the current growing issue of running out of physical bandwidth as scale increases.

All in all, FOG computing reduces latency and bandwidth by extending the cloud to edge networks, which improves the overall quality of service.

2.3 IoT Communication Standards

The rapid expansion of IoT has pushed the development of standards that are better suited for this user space. As a result, multiple communication standards have been proposed and are now competing to be the single standard. Two of the most used communication standards are CoAP and MQTT. This section describes each protocol and analyzes the best protocol for the IoTA system.

2.3.1 CoAP

Created in 2015, CoAP, or Constrained Application Protocol, is a new networking protocol that was designed specifically for the Internet of Things [8]. It is designed to have a low overhead and small footprint, so it can be effectively utilized in machine to machine communication in constrained network applications. To achieve the ability to run sufficiently on low power or lossy networks, CoAP uses UDP (User Datagram Protocol) for transport layer communications.

CoAP is also designed to operate with very little hardware requirements. An example of a lower power IoT endpoint device is an 8-bit micro-controller. CoAP's 4 byte fixed header and small packet size allows for packets to be parsed in place without needing extra RAM.

2.3.1.1 CoAP Model

CoAP is designed to be easily translatable to HTTP using proxies. To facilitate this, CoAP uses a REST like model, where devices can submit POST, PUT, GET, and DELETE requests to a server resource.

As previously mentioned, CoAP is built on top of UDP. Utilizing UDP for communications eliminated the overhead associated with TCP. UDP is optimal for IoT in streaming continuous sensor data. However, to maintain quality of service, CoAP requires a "Confirmable" flag in the header, in which the recipient is required to respond back.

For communication security, CoAP utilizes DTLS (Datagram Transport Layer Security).

2.3.2 MQTT

MQTT, or Message Queue Telemetry Transport, was originally designed by IBM, but is now an open source standard. [4] MQTT was also designed to be a lightweight messaging system. However, MQTT utilizes a public/subscribe model, unlike HTTP's request/response model. This architecture is optimal for many to many communications, unlike CoAP's one to one model.

2.3.2.1 MQTT Model

MQTT requires a message broker, or a server to distribute messages to clients that have subscribed to a certain topic. This requires each MQTT client, or endpoint device, to keep a constant TCP connection open to the message broker server.

For transport layer communications, MQTT utilizes TCP (Transmission Control Protocol). This requirement on TCP may prevent MQTT from being utilized in small micro-controllers.

For communication security, MQTT does not offer any additional levels of security outside of using TLS (Transport Layer Security).

2.3.3 CoAP or MQTT?

Both MQTT and CoAP are lightweight communication protocols that are focused on running in constrained environments. Both are open standards, run on IP, and support asynchronous communication.

While both standards can be used for this application, the IoTA system utilizes CoAP for it's communication standard. CoAP's one-to-one model is more optimal than MQTT's many-to-many model for transferring state information between the endpoint device and the server. In addition, CoAP has a much smaller code footprint

and network usage than MQTT, making CoAP usable on a wider range of lightweight devices.

There are many implementations of CoAP in many different languages, including Java, Python, C/C++, Go, Javascript, and Ruby. For IoTA, we chose the aiocoap library in Python 3.

2.4 Docker

Docker [2] is an open source project designed to run applications in separate 'containers'. These containers contain source code, dependencies and libraries necessary for applications to run. This allows developers quickly develop applications, without the need to worry about hardware or library requirements on a server.

A Docker instance mimics the functionality of a virtual machine. However, Docker was designed to share a virtual operating system. This architecture is more efficient than having a hypervisor managing system resources. This results in each Docker instance containing a small container with an application. This small container footprint allows Docker to efficiently run many "virtual machines" at once.

Chapter 3

RELATED WORKS

Fog computing has been a recent area of study and development. There have been analysis of architectures and implementations, both of which were considered and incorporated into IoTA. These projects are outlined below with a critique on their implementations and designs.

3.1 The Fog Computing Paradigm: Scenarios and Security Issues

This paper [12] explores the advantages and disadvantages of fog computing, as well as some applications in real life scenarios. Smart grids, for example, would be able to load balance the energy grid. Fog computing would be able to collect all of the sensor data from edge nodes or networks, and process them locally. Once processed, the information can be used to send direct commands to the actuators on the network for near real time results. This can reduce latency by reducing the amount of data sent to the cloud for processing.

This time saving aspect could also be used in traffic lights as well. Traffic lights could sense oncoming traffic, as well as pedestrians. A network of smart traffic lights could be used to reduce traffic by allowing a cascading effect of green lights to allow traffic to flow easier. Fog computing would again reduce unnecessary network traffic and latency of cloud processing.

This paper also goes into some of the issues of fog computing as well. Security is a huge issue with fog computing. This paper provides an in-depth example of a man-in-the-middle attack on a fog computing system. In this type of attack, fog systems that are compromised could be replaced with fake nodes to be utilized for

malicious activity. A man-in-the-middle attack would be easy to implement, because of the distributed nature of the systems.

All in all, this paper explains the need for fog computing by giving multiple examples of where it could be used. This paper only explained possible architectures, and did not have a current working system.

3.2 Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing

The evolution of certain technologies has enabled the Internet of Things to grow rapidly over the few years. [13] This paper evaluated these contributing factors, as well as explains main challenges faced by the Internet of Things.

The evolution of bigger and better batteries has increased the possibility of where electronic IoT devices can work. This coupled with the increased power efficiency of SOCs (System on a Chip) has created long life sensors that are not required to be attached to a power grid. In addition to better batteries, renewable energy devices are already starting to emerge.

Since the Internet of Things is expanding at such an exponential rate, more and more devices are being added to networks. Older networks are not able to handle the amount of traffic produced by all of these devices. Moving to 4G LTE/EPC for cellular networks would be able to expand the bandwidth for edge networks, increasing the feasibility of fog and cloud computing. Lastly, this paper ends off by analyzing some potential challenges for fog computing systems. This includes computational and storage limitations, security, and standardization of a protocol or system. Privacy and security were also issues that heavily emphasized. This paper also did not have a working fog computing system.

3.3 Fog Computing: A Platform for Internet of Things and Analytics

This paper [7] also analyzes fog computing as a whole. However, this paper provided much more possible specific system architecture details than the previous two papers. This paper also gave examples of how fog computing can be utilized. The two examples provided were smart traffic lights, and a wind farm. In addition to analyzing the benefits of fog computing in these situations, they also provided key requirements and system outlines for these situations.

In order for fog computing to be utilized to its fullest potential, it must be used in tandem with cloud computing. The proposed architecture had a fog system managing a sensor/embedded system network, while the cloud managed applications and data warehousing. Fog computing would be an abstraction layer between IoT devices and cloud computing. This would provide a generic API for cloud applications to control IoT devices.

This paper proposed a simple API. This, however, was not implemented. This paper did outline a policy-based framework that would allow greater control for load balancing, power management, and security.

3.4 Mobile Fog: A Programming Model for LargeScale Applications on the Internet of Things

This is the only paper [9] that actually produced and evaluated a fog computing system. This system was focused on providing a high level programming model that simplifies the concept of distributed computing. It also was focused on dynamically scaling application to optimally use available resources.

The API produced by this paper was well documented. It explained what each function does, and how it would be used by a node to utilize a fog computing system.

After explaining the API, it evaluated the system with vehicle tracking cameras. In vehicle to vehicle data streaming, fog computing always beat cloud computing, because it was able to utilize local networks to send data instead of sending the data through the cloud. However, when testing query range, cloud computing outperformed fog computing only at high ranges. This is due to aggregating sensor data at edge nodes before evaluating user queries.

This paper seemed to mostly be concerned about fog computing and its impact on network traffic. Utilizing fog computing reduces latency and outperforms cloud computing at lower query ranges.

3.5 Processor Offloading in Cell Phones

The concept of offloading heavy application loads has already been proven to be beneficial for low powered battery devices. The Cuckoo system [10] created a framework for smart phones to offload computations. The Cuckoo system was able to increase the computational speed and reduce energy consumption of Android devices by utilizing an external computational server.

This system was also built to run in mobile environments on low powered devices. They successfully proved that offloading certain computational tasks would be beneficial. Their system, however, is designed for developers to have full control over both local and remote executions, which lets developers implement different functions for local and remote executions. This system does not allow for existing code to be remotely executed, causing a need to rewrite an entire application to utilize Cuckoo.

3.6 Offloading Benefits

Not all applications can benefit from being offloaded to an external server [11]. The energy usage and reliability of cloud computing were evaluated for different applications in this paper. Energy usage was wasted when the cost of communication power was much greater than the cost of power needed when processing an application locally.

This paper also evaluated other problems with offloading computations to cloud computing such as real time data and security. This paper concludes that before offloading, applications should be evaluated for reliability, privacy, and energy overhead.

Chapter 4

DESIGN

This chapter discusses the goals and requirements of the IoTA system. These requirements will focus on how endpoint devices will utilize the IoTA system rather than focusing on technical specifications.

4.1 Goals

Our primary goal with IoTA is to increase the functionality of end point devices without the need of the physical replacement of the device. This should be done while maintaining or improving the good quality of service. Therefore, if the IoTA system is not available, the functionality of the endpoint device should not fail. Instead, the endpoint device would complete the task without utilizing the benefits of the IoTA system. In addition, IoTA should be easy to apply to an existing system, as well as scale easily.

4.2 Requirements

In this section, we formalize the list of requirements in how the system would need to be structured. The requirements discussed are: server deployment and maintenance, scalability, security, client deployment, and client configurability.

Server deployment describes the ease of setting up new systems and maintaining existing ones. Scalability is the ability to expand to support more endpoint clients. Security is concerned with the secure transport of data from the endpoint device to the server. Client deployment describes the ease of setting up and tearing down clients. Lastly, client configurability describes the ease of adapting the client to a

range of applications.

4.2.1 Server Deployment and Maintenance

The server must be easy to deploy and maintain. IoTA is designed to be an add-on to any IoT network to boost performance of the system. Naturally, this requires IoTA to be easily integrable with any system without the need to rewrite an existing codebase.

The formalized requirements are noted below:

- Server must run on *nix box
- Server must require little maintenance once deployed
- Server must not generate unnecessary traffic
- Server must support many clients
- Server must run a variable amount of executables
- Must have documentation of all server requirements and dependencies

4.2.2 Scalability

IoTA must be scalable to allow the system to grow and evolve with the Internet of Things. Since IoT is expanding exponentially, upgrading IoTA to support more clients should require minimal effort. One simple approach to expanding IoTA is to create more server instances. However, this requires more configuring on the client side by manually allocating the endpoint devices to each server. Thus, scalability must be abstracted from the clients.

The formalized requirements are noted below:

- Server must be able to expand the number of supported clients
- Server must be able to monitor each endpoint device to observe system utilization
- Expansion of the server must be transparent from the client

4.2.3 Security

Security is critical in IoT solutions. The server must prevent eavesdropping of sensitive data, as well as ensuring the transported data is not tampered with.

The formalized requirements are noted below:

- Server must maintain security by encrypting packets

4.2.4 Client Deployment

Client deployment is concerned with the ease of setting up new endpoint devices. We require that the clients should not have to rewrite existing endpoint software as to easily integrate with existing IoT networks. Also, we require a small code footprint to interface with IoTA. This allows the IoTA system to be compatible with many low power endpoint devices.

The formalized requirements are noted below:

- Client should be easy to deploy
- Client should require little, if no, maintenance
- Client should be light weight to run on embedded devices
- Client should run source or compiled code.

4.2.5 Client Configurability

All IoTA clients should be easy to configure. Ideally, IoTA will require a small amount of settings to run properly with the server. IoTA is also required to be dynamic in the sense that the client should able to be adapted to any endpoint device use case.

The formalized requirements are noted below:

- Client must be easily configured
- Client must be able to run various executable, including source code and pre-compiled sources.

Chapter 5

IMPLEMENTATION

This section describes the technical implementation of IoTA. The goal of this section is to document the program flow and system architecture of IoTA. This section also provides a general overview of the codebase and its organization.

5.1 Overview

IoTA is composed of two different components. The server handles all requests submitted by the endpoint devices. The clients are any endpoint device that utilizes the IoTA system. For a proof of concept, the client and the server are both written in python3. However due to the many implementations of CoAP, a client can be written in many different languages.

The IoTA system essentially allows endpoint devices to take advantage of a shared computational resource. This shared resource creates a single virtual machine for each physical endpoint device. Each endpoint device communicates with the server using a client API utilizing CoAP. The API allows endpoint devices to send executables, source code, or data to the server for faster processing and storage. The overview of the system is shown in figure 5.1

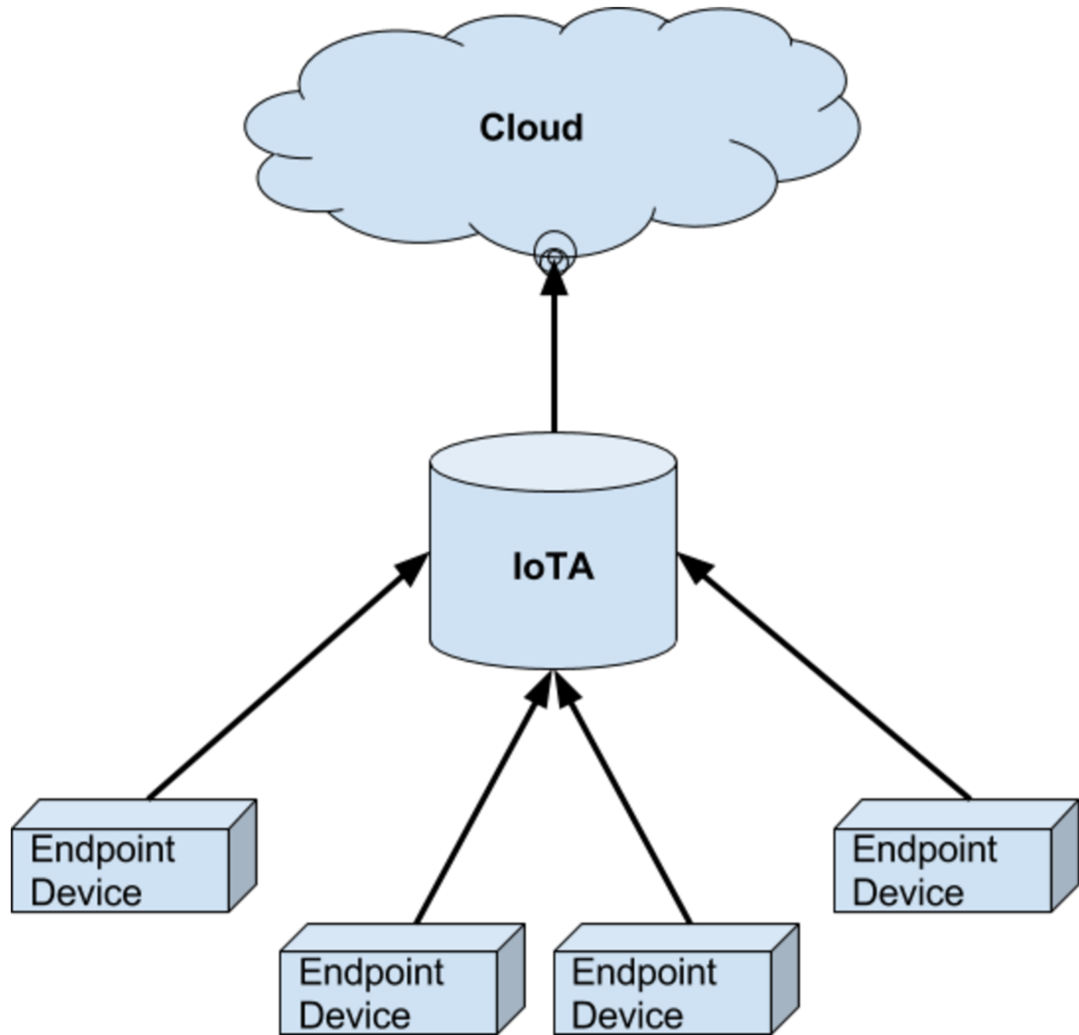

Figure 5.1: An overview of the IoTA System

5.2 Server

5.2.1 Server Requirements

5.2.1.1 Hardware Requirements

The hardware requirements for the IoTA system keep scalability and flexibility in mind. However, for the IoTA system to run optimally, it is crucial that the server have better hardware than the endpoint devices. There will be little to no performance benefit if the server hardware is computationally equivalent to the endpoint devices.

As a result of using virtual machines, the server has no specific formal requirements on server hardware. This flexibility allows endpoint clients to run different architectures and instruction sets.

Additionally, users are able to choose how much hardware to allocate to the IoTA system. Due to the diversity of IoT network structures and sizes, we cannot generalize formal server system requirements. Variables such as network size, endpoint utilization, and server functionality must be considered into choosing optimal hardware for a specific IoT network.

It must be noted that the inclusion of multi-core CPUs (Central Processing Units), GPUs (Graphics Processing Units) or co-processors have the potential to increase IoT network performance depending on the application. Again, these peripherals are not formally required for the IoTA system to function.

Lastly, due to the rapid growth of the Internet of Things, it is crucial that we keep scalability in mind for this system. One of the benefits of utilizing Docker for virtual machines is that we can take advantage of Docker Swarm to expand the system. Docker Swarm allows us to cluster Docker nodes, and abstract them as a single virtual system. If an IoT network outgrows its current configuration, users

can append additional Docker nodes easily without reconfiguring the entire system or network.

5.2.1.2 Software Requirements

The server requires Docker as it is a core component of the system. IoTA also requires a database and a CoAP library. As a proof of concept, IoTA uses MySQL for the database, and Aiocoap as the CoAP library. However, IoTA is constructed in such a way where the database can easily be changed to support a wider range of IoT applications. Unfortunately, the CoAP communications can not be hot swapped at this moment due to limitations of the available CoAP libraries and how they are structured.

5.2.2 Architecture

The architecture is designed to expand the functionality of endpoint devices. This is achieved by letting endpoint devices run jobs on virtual machines. In this system, there is a one-to-one relationship between each physical device and a virtual machine (VM). The VM is used to store source code, pre-compiled executables and data. The overview of the server architecture is shown in figure 5.2

5.2.3 Requests

The processing resource can be accessed using CoAP's four methods. In all methods, the IoTA system expects properly formatted JSON requests to be processed correctly.

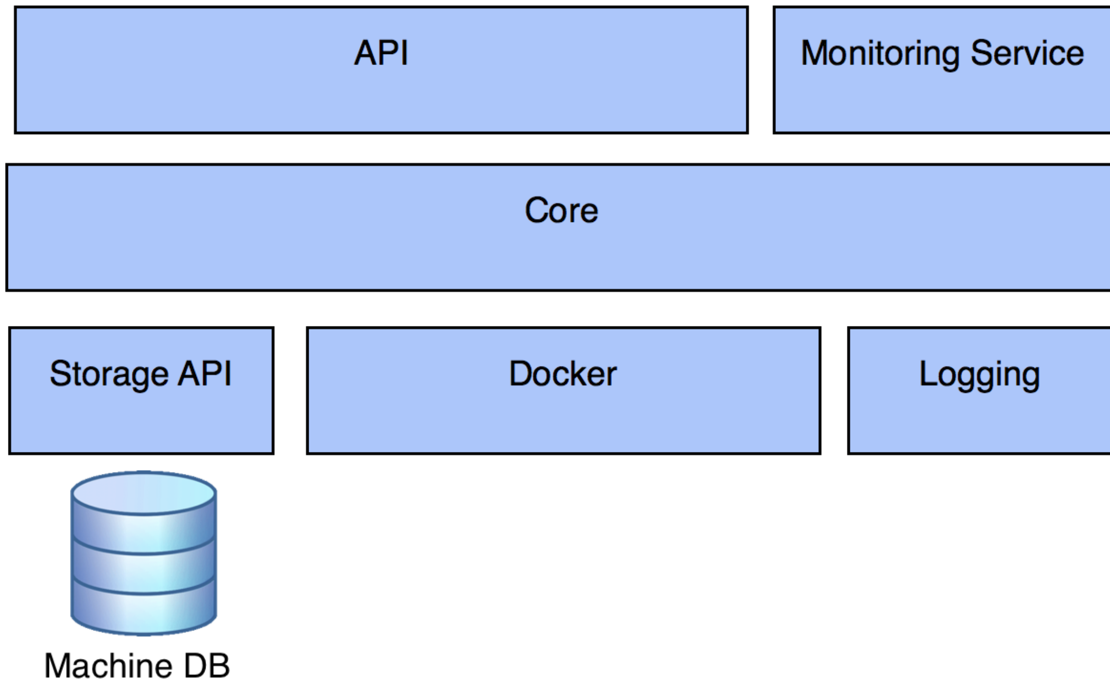

Figure 5.2: An overview of the server

5.2.3.1 POST

The POST request, an example shown in 5.3, is responsible for creating and configuring a virtual machine for the end point device. The request requires a few fields to properly instantiate the VM. The "name" and "device" are descriptive information given by the device. The "MAC" address of the device is used as a unique identifier for each machine. The "DockerImage" is the name of the VM that the endpoint wishes to use. Lastly, "Update" is a flag used by the client to update any information about a device that is already in the system.

If correctly configured, the server will respond with a unique token that the client will use for the rest of the requests. This unique token is the containerID of the Docker container. If the server encounters an error or receives an invalid POST request, the server will respond with an error message. Figure 5.4 shows the network diagram of

```
{
  "name" : "Endpoint Device 1",
  "device" : "Raspberry Pi 2",
  "MAC" : "FF:FF:FF:FF:FF:FF",
  "DockerImage" : "python",
  "Update" : false
}
```

Figure 5.3: An example of a POST Request

a POST request.

The usage of Docker allows us to support a wide range of applications. Developers can either choose existing Docker images (from Docker Hub), or even create their own for specific applications.

It should be noted that if the IoTA system encounters a valid Docker image that has not been cached on the server, this request can take longer than expected since Docker will have to download the image before being utilized by IoTA.

5.2.3.2 PUT

The PUT request, an example shown in figure 5.5, is responsible for transferring dependency information, source files, executables, and data files to the VM. As previously mentioned, this request requires the unique token provided by the POST request. This request also requires a list of commands to be executed.

A command is defined by the command type, the fileName, and the payload. There are two types of commands currently supported in a PUT request: CopyTo

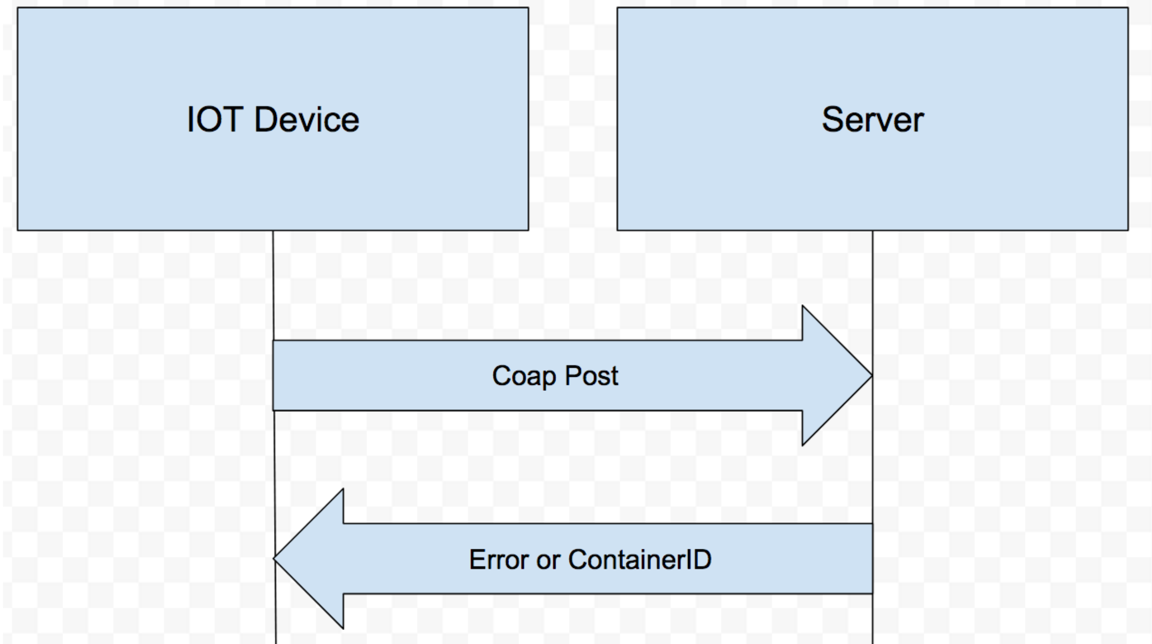

Figure 5.4: Network Diagram of a POST Request

and Exec. The CopyTo command will copy an file to the virtual machine with the provided file name and contents from the payload field. The Exec command will execute the given payload in the virtual machine. The Exec command can ignore the "filename" field in the JSON request.

To assist endpoint devices, dependencies and requirements for jobs are communicated to the server via a text file and automatically installed. This text file is language specific. For example, a Python container would require a file named "requirements.txt", while a Java container would require a file named "pom.xml". It should be noted that installing dependencies in this fashion is slow and not recommended. The better way of installing dependencies is the use of a custom Dockerfile [3]. This allows Docker to cache images for faster VM initialization times.

The CopyTo commands will return an acknowledgement if the file has been successfully transfered to the VM. The Exec command will return a unique ID for that job. This JobID is used in the GET request to retrieve data. Figure 5.6 shows the

```
{
  "UID" : "1234567890",
  "Commands" : [
  {
  "Command" : "CopyTo",
  "FileName" : "requirements.txt",
  "payload" : "707963727970746f3d3d322e362e310a"
  }
  ]
}
```

Figure 5.5: An example of a PUT Request

network diagram of a PUT request.

5.2.3.3 GET

The GET request, an example shown in figure 5.7, is responsible for retrieving the output from a job. This request requires the unique container ID, as well as the unique job ID. This request will return an error if the job has not completed yet. If the job has successfully been completed, the request will return the raw output.

It should be noted that most end point devices will not need to retrieve the information from the server. In most applications, data is sent to the server for processing before being sent elsewhere, such as the cloud. Figure 5.8 shows the network diagram of a GET request.

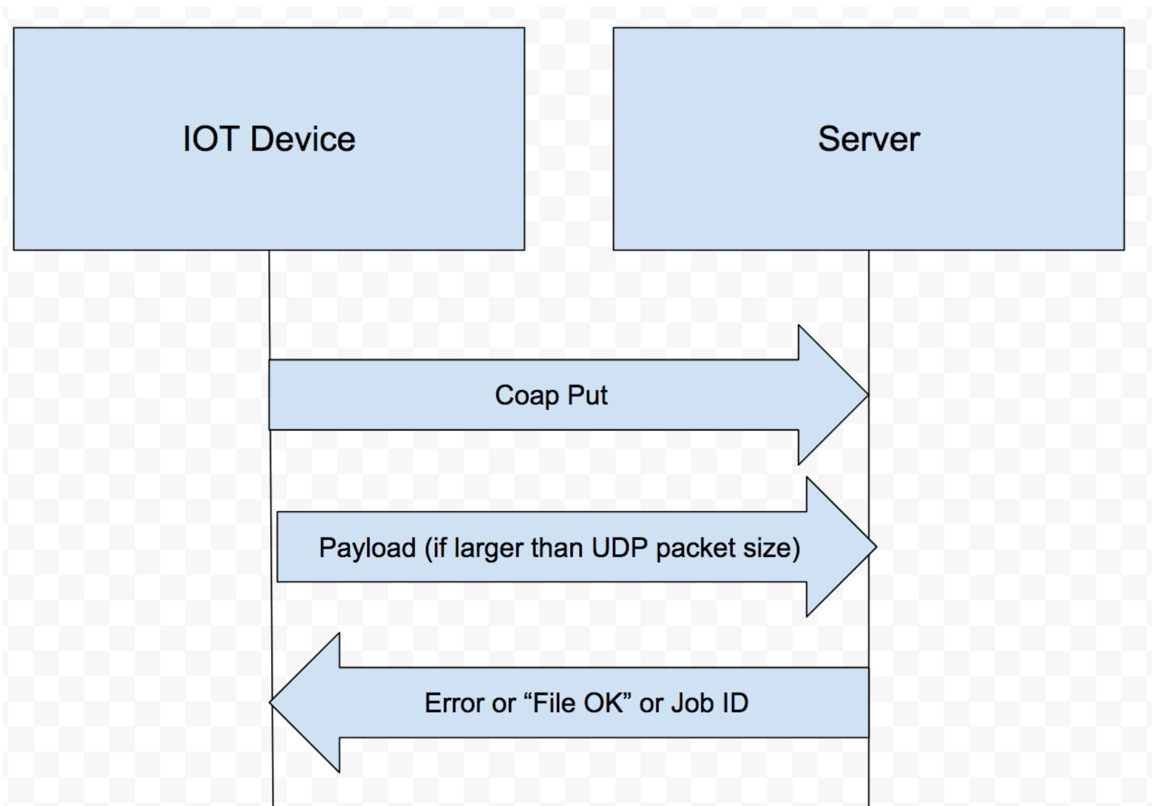

Figure 5.6: Network Diagram of a PUT Request

```
{
  "UID" : "1234567890",
  "JID" : "a1b2c3d4-a1b2-a1b2-a1b2-a1b2c3d4e5f6"
}
```

Figure 5.7: An example of a GET Request

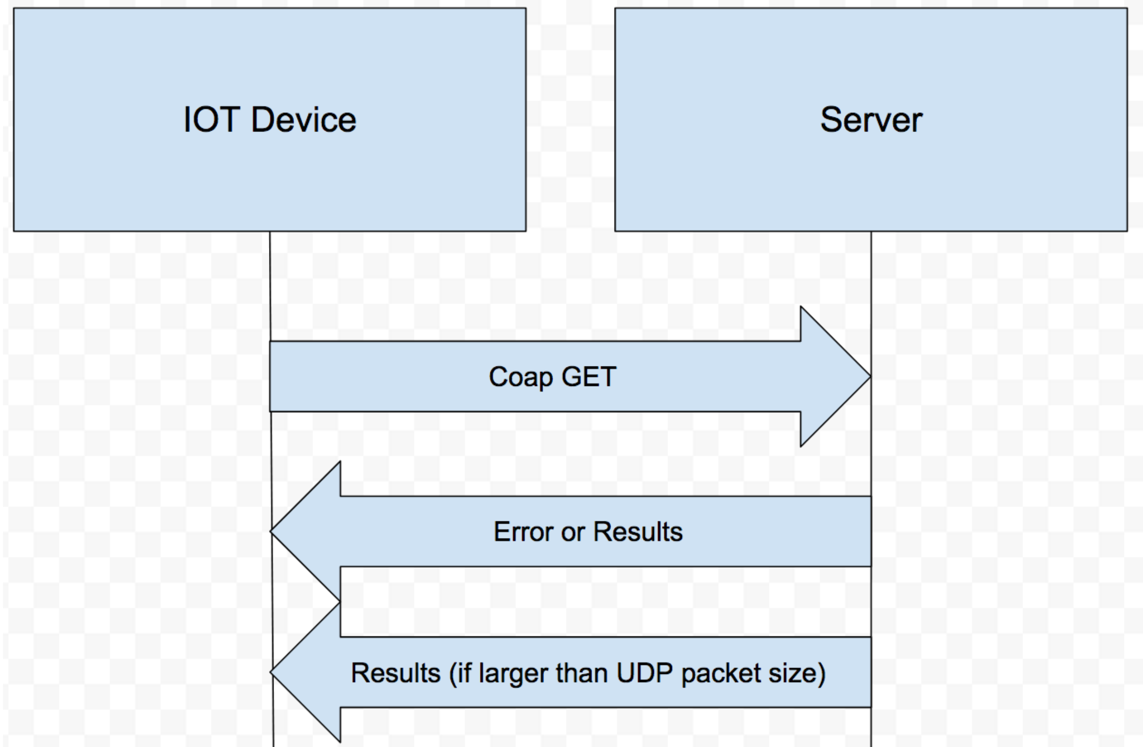

Figure 5.8: Network Diagram of a GET Request

5.2.3.4 DELETE

Lastly, the DELETE method, an example shown in figure 5.9 is used to remove a virtual machine from the IoTA server. This request only requires the unique container ID. This method cleanly removes the virtual machine and any temporary files related to this container. It should be noted that the server can be configured to automatically remove any VM that has not been used recently. This allows the server to free up resources when possible. Figure 5.10 shows the network diagram of a DELETE request.

5.2.4 Database Architecture

The database is structured to store data about each endpoint device and the jobs that each endpoint device submits. The database is composed of two tables:

```
{
    "UID"  :  "1234567890"
}
```

Figure 5.9: An example of a DELETE Request

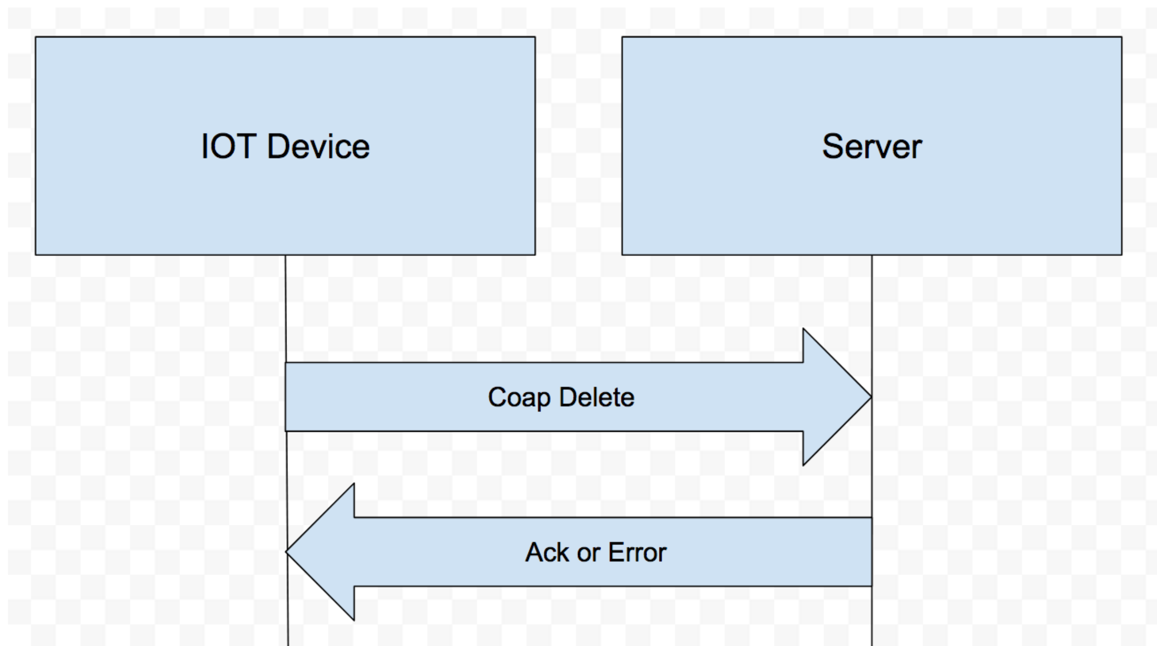

Figure 5.10: Network Diagram of a DELETE Request

The table to store the endpoint device is as follows:

- Name VARCHAR(30)
- Device VARCHAR(30)
- MAC VARCHAR(30)
- DockerImage VARCHAR(30)
- ContainerID VARCHAR(10)

Both the name and device fields stored by the server are used as descriptors submitted by each client. The name and device fields are not meant to be used as unique identifiers by IoTA. The unique identifier for the endpoint device is the MAC address of the client.

The DockerImage field contains the name of the Docker image that the endpoint devices uses. This allows us to use any virtual machines and languages supported by Docker.

The Job table's columns are as follows.

- JID VARCHAR(36)
- CID VARCHAR(10)
- Completed INT
- StartTime INT
- FinishTime INT

Both the start and end time recorded by the system are in milliseconds. This provides us the ability to analyze all of the jobs submitted to see what device takes the most

computational resources. This data can then be used for analysis to load balance the server.

5.2.5 Module Overview

This section describes each of the components in the codebase to aid in anyone trying to understand it. IoTA was designed with modularity in mind for the ease of implementing new features. This section describes each part of the server source code.

5.2.5.1 Server

This file contains the main logic for the server and the server resources. It starts up a CoAP server and starts listening on port 5683. The server has one resource that is reachable at "coap://(server IP)/processing". Lastly, the server has options for a clean boot, which clears all database tables, as well as resets the IoTA file structure.

5.2.5.2 Utils

This file contains general utility functions that are used by the server. This includes simple file IO, simple json parsing, as well as generating unique IDs.

5.2.5.3 Machine

This file contains an object model for an endpoint device. This file also contains functions to interface with the Docker container that the device is associated with. These functions include copying files to and from the VM, restarting or pausing the VM, and running commands on the VM.

5.2.5.4 IoTDB

This file contains a simple interface the interact with the database. This implementation includes MySQL commands to create the tables and store/retrieve machines and jobs from the database. By abstracting the database to these few functions, this allows for a seamless transition if a user would like to use a different type of database by simply re-implementing these methods.

5.2.5.5 Command

This file contains an object model for a Command. A Command is essentially any job that an endpoint device submits to the server. The command is responsible for running, timing, and storing the result of each job.

5.2.5.6 coapExceptions

This file contains all of the exceptions that this application may encounter. These exceptions provide developers with meaningful errors to assist in debugging.

5.3 Client

The client is a simple API used to interface with the IoTA server. The API simply formats the data, and sends them in the correct CoAP requests. Our implementation was written in Python 3. However, since the client API is essentially a CoAP wrapper, it would be a trivial exercise to implement a client API in a different language.

5.3.1 Architecture

The client enables end point devices to send over files to the virtual machine. To ensure usability, developers do not need to rewrite existing code. Instead, developers would send over pre-compiled or source code to be ran on the IoTA server. The API includes methods to copy files and execute jobs.

To ensure the best possible quality of service on a dispatched job, the client API runs a local version concurrently with the version running on the IoTA server. The client API then returns the output of whichever job completes first. This eliminates the network overhead of using the IoTA system on small jobs, where running locally is more optimal. This also has the additional benefit of still completing jobs in the event that the IoTA system is unavailable or unresponsive.

5.3.2 Config

The configuration of clients was kept very simple for this application. Assuming a developer has utilized the API correctly, the client only requires the IP address of the server to function properly. Server discovery functionality was omitted in this proof-of-concept for simplicity and to reduce latency for evaluation.

5.3.3 Module Overview

This section describes each of the components in the codebase to aid in anyone trying to understand it. IoTA was designed with modularity in mind for the ease of implementing new features. This section describes each part of the client API source code.

5.3.3.1 IoTClient

This file contains all necessary logic to send valid CoAP requests to the server. The client was kept very simple, since it is meant to run on an endpoint device. Most of the logic for this system is implemented in the server itself.

Chapter 6

SYSTEM EVALUATION

As the main goal of this thesis is to obtain better performance in computations, the hope is that the IoTA system would assist the endpoint devices in this task.

The results were gathered in this thesis were all performed on one system. The IoTA server is a 2010 MacBookPro with an Intel i7 chip clocked at 2.4GHz. This system also includes 16GB of RAM, and access to a dedicated graphics chip. The client is a Raspberry Pi 2 with a quad-core ARM processor running at 900MHz with 1GB of RAM. The Raspberry Pi 2 was chosen as the endpoint device because of it's use in DIY IOT applications and it's weak computational power when compared to the server.

To simulate a real IoT application, both the server and the client ran on a local WiFi network (802.11n) with other networked clients.

We evaluated the IoTA system's ability to run on two different algorithms: matrix multiply and prime number factorization. These algorithms were chosen because of their ability to be translated into real world applications.

6.0.1 Performance Measurements

We executed each benchmark locally on both the server and client for our base timings. We then tested the IoTA system in two different configurations. In one instance, all communication data was sent uncompressed, while the other instance compressed all data before being sent.

In the matrix multiplication tests, different size matrices were sent to the server, multiplied against itself, and sent back to the client for validation. The matrix mul-

tiplication program was written in Java, so the client sent over compiled Java byte code. 6.1 shows the average of ten run-times for each matrix size.

The prime factorization test was written in Python, and sent over as source code. In this test, clients sent a single prime integer to be evaluated to the server. The prime factorization program is a naive implementation to check if a number is prime or not. This benchmark iterates from 2 through n-1, where n is the input number. 6.2 shows the average of ten run-times for each tested prime number.

It should also be noted that the local execution of programs on the client side were disabled when using the IoTA system to measure the network overhead of the system. If full functionality of the IoTA client were tested, the processing time would be the faster time between the local execution time or the server execution time.

6.0.2 Accuracy

To ensure that our results from the IoTA system were correct, outputs from the IoTA server and local execution were both compared. In all tests, the outputs were the same, which indicates that the IoTA system produces the same results.

6.1 Results

In figure 6.2, we see that for lower matrix sizes, processing locally on the client was faster than using the IoTA system. As predicted, the network overhead is too large in this application to achieve any benefits. However, for larger matrix sizes (512x512 and 1024x1024), we observe a massive performance benefit of using the IoTA system.

We observe that our current bottleneck is how fast we can transfer the files between the client and the server. In typical IoT applications, most clients would not need to utilize the GET request, but instead send the data elsewhere. This essentially cuts

$$LocalProcessingTime > FileTransfer(data) + ServerProcessingTime + FileTransfer(results)$$

Figure 6.1: Performance Benefit of using IoTA

down our endpoint device network overhead in half for this specific example. We have generalized our performance in the formula in figure 6.1. As data size increases, we experience decreased performance. In the matrix multiplication tests, this large file transfer overhead was offset by the amount of computations needed for such large matrix sizes.

In the prime factorization benchmark in figure 6.3, we observe that the performance benefits are different than in the previous benchmark. From the graph, we observe a linear relationship between the number of iterations versus the time of computing the output when running the algorithm locally on the client. However, when running on the IoTA server, we experience a near constant run time.

This constant run time is due to the nature of the prime factorization algorithm. This benchmark sends very little data between the client and the server, essentially reducing our file transfer overhead to a minimum. In this example, using the IoTA system is beneficial for any client computations taking over 8 seconds.

From the graph, we also observe an interesting relationship between transferring all data compressed or uncompressed. In this benchmark, we do not see any performance benefit or drawback from compressing the data. This is due to the minimal data being sent between the server and client. Since there is no drawback from compressing the data, the IoTA system should be configured to compress all transferred data.

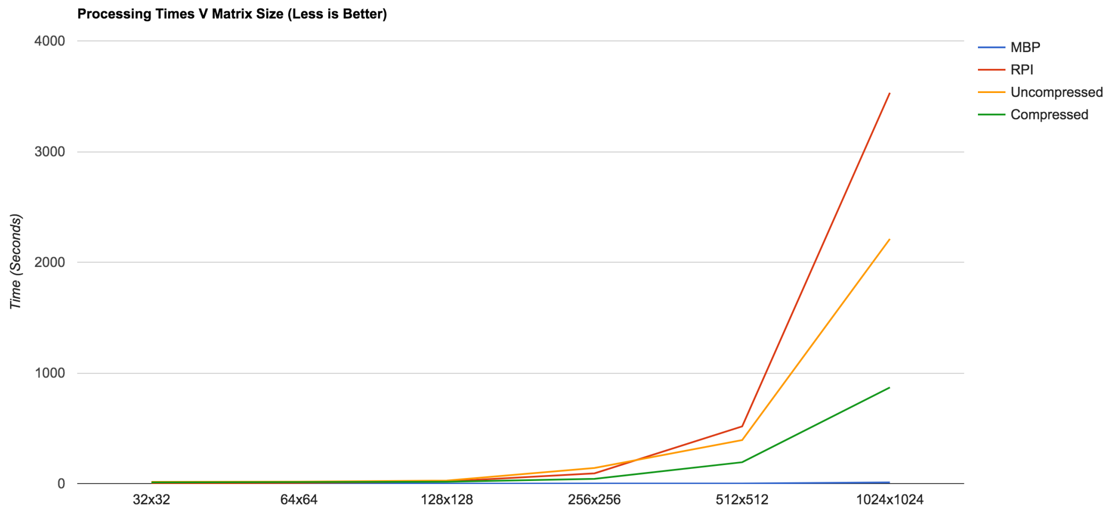

Figure 6.2: Matrix Multiplication Runtime Results

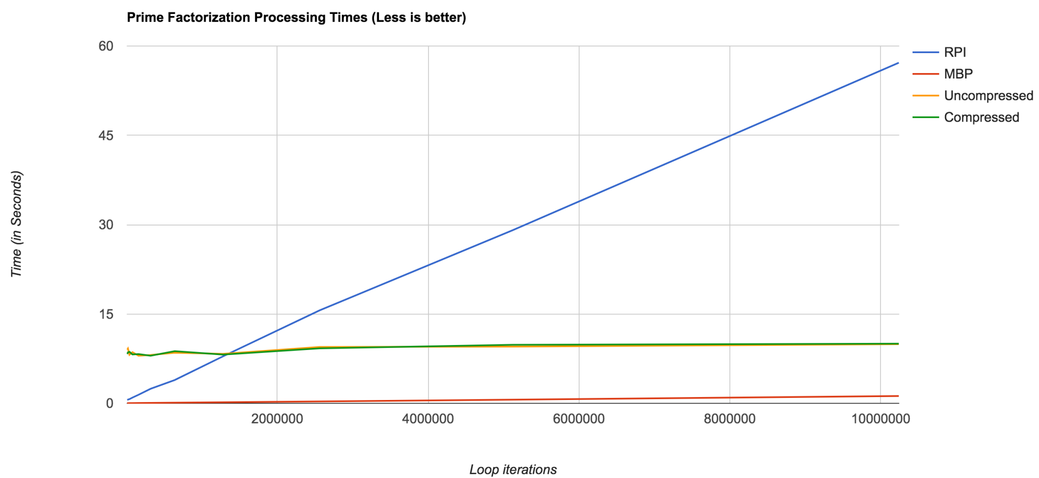

Figure 6.3: Primality Runtime Results

Table 6.1: Matrix Results Table

Matrix Size	MBP	RPI	Uncompressed	Compressed
32x32	0.1133333333	2.306666667	15.46333333	10.498
64x64	0.18	5.37	17.06666667	13.582
128x128	0.3033333333	19.58	27.34666667	17.02
256x256	0.5233333333	92.43	140.8866667	42.724
512x512	1.29	516.9033333	393.16	193.134
1024x1024	10.38333333	3531.04	2210.703333	868.464

Table 6.2: Primality Results Table

Number of Iterations	MBP	RPI	Uncompressed	Compressed
10007	0.027	0.588	8.99	8.278
20011	0.026	0.616	9.197	8.564
40009	0.031	0.701	8.142	8.646
80021	0.036	0.961	8.593	8.21
160001	0.046	1.435	7.995	8.274
320009	0.066	2.439	8.119	7.985
640007	0.102	3.91	8.494	8.763
1280023	0.176	7.887	8.31	8.181
2560021	0.323	15.599	9.466	9.23
5120029	0.604	29.071	9.528	9.828
10240033	1.219	57.193	9.922	10.01

6.1.1 IoTA Breakdown

For further evaluation, we timed each step of the IoTA system so we can observe which steps took the longest to execute. This will identify any bottlenecks in the system. The IoTA system can be broken down into five steps:

1. POST Request: Creating a virtual machine (Docker Instance) and returning a container ID
2. PUT Request: Copy source files
3. PUT Request: Copy data files (if any)
4. PUT Request: Run the command and return a Job ID
5. GET Request: Get the data from the server

The time elapsed per IoTA request was recorded using python's time library. Each benchmark was averaged over ten runs. Table 6.3 shows the breakdown of the matrix multiplication test using the IoTA system (with compressed data transfer).

From this breakdown, we observe that starting a VM, sending source code, and sending a run request were constant time through all the different matrix sizes. As matrix size increased, both sending data files and getting the results from the server also increased. This, again, suggests that our bottleneck is network overhead when sending over large files to and from the server. It is interesting to note that the GET request consistently took about 50% of the total time across all matrix sizes.

Since the primality test is linear in nature when using the IoTA system, we ran this test once as the results would be the same for all previously tested prime numbers. Table 6.4 shows the breakdown results for the primality benchmark. Compared to the matrix multiplication breakdown, we can observe that starting a VM and submitting

a run request are of constant time. Again, we can see that the GET request takes over 50% of the total time. This massive bottleneck can be avoided if data is not sent back the endpoint device for evaluation.

Table 6.3: Matrix Multiplication Breakdown Table

Step	32x32	64x64	128x128	256x256	512x512	1024x1024
POST	0.74 (15.6%)	0.75 (12.7%)	0.78 (7.6%)	.73 (2.8%)	0.73 (0.7%)	0.83 (0.2%)
PUT Source	0.88 (18.6%)	0.89 (15.1%)	0.87 (8.4%)	0.92 (3.5%)	0.89 (0.9%)	0.89 (.2%)
PUT Data	0.62 (13.1%)	1.08 (18.3%)	3.49 (33.9%)	11.81 (44.6%)	50.58 (48.3%)	221.21 (44.9%)
PUT Run	0.16 (3.4%)	0.16 (2.7%)	0.16 (1.6%)	0.16 (0.6%)	0.16 (0.2%)	.16 (0.0%)
GET Result	2.38 (50.3%)	2.89 (49.1%)	5.04 (48.9%)	12.97 (48.9%)	52.28 (50.0%)	269.23 (54.7%)
Total Time	4.73	5.89	10.3	26.5	104.25	492.35

Table 6.4: Primality Breakdown Table

Step	Time (in Seconds)
POST	0.70 (20.3%)
PUT Source	0.39 (11.3%)
PUT Data	0.0 (0.0%)
PUT Run	0.15 (4.3%)
GET Result	2.21 (64.0%)
Total Time	3.45

Chapter 7

CONCLUSION

We have developed an approach to improve the functionality of endpoint devices without the need of physical replacement. In order to complete this project, a set of requirements were constructed that would ensure the systems usability, flexibility, and scalability in existing IoT applications.

The evaluation results of IoTA show that we achieved faster processing times with a reasonable amount of network overhead. In addition to improving performance, IoTA unlocks greater potential for IoT networks. This system has the ability to store and condense data before it gets sent to the cloud. This aggregation of data has the potential to reduce outgoing bandwidth from a local network. Also, this system enables lightweight endpoint devices to utilize the full power of multi-threaded applications and GPU support.

This application has endless use cases in both smart homes and industrial applications. Using IoTA has the potential to improve the quality of service of any existing IoT network. Overall, the IoTA server successfully built and tested a FOG computing proof of concept. Most importantly, the IoTA system preserves the semantics of the Internet of Things while extending functionality.

Chapter 8

FUTURE WORK

As with all systems, there is a lot of potential to expand the functionality and improve performance of this project. The results of this paper showed significant performance increase when jobs were processed on the server, rather than locally on an endpoint device. The IoTA system demonstrated that overhead can be reduced by compressing data before being transferred. However, we expect that the overhead of the IoTA system can be reduced even further.

Additionally, as stated in section 5.3, it would be simple to implement a version of the client that runs on C. For this proof of concept, we did not run our system on an embedded platform. However, we predict that we would see similar performance benefits if ran on lower powered devices. Naturally, if a client were to run embedded C code on the IoTA server with a different architecture, a system emulator such as QEMU will have to run in a Docker container.

Another potential area for improvement lies in VM optimization. Currently, the IoTA system keeps all instantiated virtual machines alive, so that they may be used at any time. However, keeping VMs alive ties up system resources on the server. To extend the scalability of this system, VM usage can be monitored and analyzed to predict when a VM may be used. This analysis can then be used by a VM scheduler to dynamically sleep and restore VMs to free up resources.

Although not tested, this system has the ability to allow micro-processors to utilize GPU and co-processor resources. IoTA was built with this concept in mind, but should be tested to ensure seamless access to the server's other resources.

The IoTA system also does not have a callback function for submitted jobs. In the

current system, if an endpoint device needs to know the status of a job, the endpoint device would have to query the server. If a job has not completed yet, the endpoint device would have to repeatedly query the server until it has completed. Adding a callback feature may reduce unnecessary network traffic from this repeated polling.

Finally, this implementation can add additional security features. Security in IoT is crucial for its success. Future iterations would have to include some type of authentication system to use the IoTA server. Additionally, effort will have to be put in to ensure that VMs are secure to keep data safe. This would further improve the system's usability.

BIBLIOGRAPHY

[1] Cal Poly Github. http://www.github.com/CalPoly.

[2] Docker. https://www.docker.com/.

[3] Docker. https://jpetazzo.github.io/2013/12/01/docker-python-pip-requirements/.

[4] Information technology – message queuing telemetry transport (mqtt) v3.1.1. https://www.iso.org/standard/69466.html.

[5] IoT, from cloud to fog computing. http://blogs.cisco.com/perspectives/iot-from-cloud-to-fog-computing.

[6] Nikola Tesla's incredible predictions for our connected world. http://paleofuture.gizmodo.com/nikola-teslas-incredible-predictions-for-our-connected-1661107313.

[7] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[8] C. Bormann, K. Hartke, and Z. Shelby. The Constrained Application Protocol (CoAP). RFC 7252, June 2014.

[9] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.

[10] R. Kemp, N. Palmer, T. Kielmann, and H. E. Bal. Cuckoo: A computation offloading framework for smartphones. In *MobiCASE*, pages 59–79. Springer, 2010.

[11] K. Kumar and Y.-H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010.

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

[13] L. M. Vaquero and L. Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[14] O. Vermesan, P. Friess, P. Guillemin, S. Gusmeroli, H. Sundmaeker, A. Bassi, I. S. Jubert, M. Mazura, M. Harrison, M. Eisenhauer, et al. Internet of things strategic research roadmap. *Internet of Things-Global Technological and Societal Trends*, 1:9–52, 2011.

APPENDICES

Appendix A

IOTA SERVER SETUP

A.1 Dependencies

```
$apt update;
$apt install docker.io; #installs Docker
$apt install mysql-server; #installs mysql
```

A.2 Required Python Libraries (Not in Python's stdlib)

- aiocoap
- asyncio (Part of python's stdlib after 3.4)
- pymysql

A.3 Running the Server

The IoTA system runs on CoAP's defined default port (5683).

```
$python3 server.py
```

Appendix B

DOCKER COMMANDS

B.1 List all Docker Instances

```
$docker ps -a
```

B.2 Run a Command in Docker Instance

```
$docker exec (containerID) (command)
```

B.3 Copy File To Docker Container

```
$docker cp (file) (containerID):(path in container)
```

B.4 Copy File From Docker Container

```
$docker cp (containerID):(path in container) (host path target)
```

B.5 Get Container's Log Files

```
$docker logs (containerID) > stdout.log 2>stderr.log
```

B.6 Portainer

Portainer is a simple UI manager for Docker. Used for debugging containers. To start portainer on port 9000:

```
$docker run -d -p 9000:9000 -v /var/run/docker.sock:/var/run/docker
    ↪ .sock portainer/portainer
```