An Adaptive Architecture for Internet of Things Applications

Brian Ramprasad

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE
STUDIES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF ARTS

GRADUATE PROGRAM IN INFORMATION SYSTEMS AND
TECHNOLOGIES

YORK UNIVERSITY
TORONTO, ONTARIO
SEPTEMBER 2018

Abstract

The number of IoT devices has been growing exponentially as new products are being developed and legacy systems are becoming Internet enabled. As a consequence of this trend, the large amounts of traffic generated by IoT devices require new approaches to platform design and workload management. The primary challenge with managing IoT devices is that the traffic can be highly variable due to the type and the time of use. To be able to maintain quality of service standards on an IoT platform, these traffic patterns need to be modeled and understood so we can adapt the architecture dynamically. To address these challenges, we propose an adaptable architecture, a platform to emulate IoT devices, and a smart testing framework to detect bottlenecks that can predict the demand for computing resources. We show that in certain cases we can predict the demand for computing resources with a high degree of accuracy.

Acknowledgements

I would like to thank Dr. Marin Litoiu for his guidance over the years as my supervisor. He introduced me to academic research and has been a great mentor.

I would also like to thank Dr. Marios-Eleftherios Fokaefs for his assistance in developing this platform and providing me with knowledge about how to conduct experiments and how to become a better researcher.

My gratitude goes to Dr. Joydeep Mukherjee for helping me with designing the experiments and reviewing my thesis. His feedback was very much appreciated.

I would like to also like to give a special thanks to Lisa Nguyen. She provided many years of support and guidance throughout this thesis. I am eternally grateful for her advice and wisdom along this journey.

Lastly, I would like to thank my parents, family, friends and lab colleagues for providing support during this thesis.

Contents

List of Tables

List of Figures

List of Equations

Chapter 1

Introduction

As the volume and variety of connected devices grows, the demand on the supporting infrastructure is ever changing. An IoT device can be considered any device with a connection to the Internet that will likely produce data that is consumed by another service. An example IoT device would be a smart home thermostat that can upload its data to the cloud. Homeowners can then view this data on a smart phone and even control the thermostat remotely. This is an example of the value that Internet enabled devices provide because they have the ability to gather data about the operating environment so that that can be analyzed. Prior to the IoT era, non-connected devices kept this data local to the device. With the increased number of connected devices we must develop new ways of processing this data. IoT applications that ingest this data and make sense of the information require us to design an architecture that can handle a large number of devices. One of the challenges with IoT networks is maintaining quality of services because of the fluctuations in the number of active devices and the data they produce [1–3]. IoT networks are typically heterogeneous, meaning that there are many types of IoT devices that behave differently which are connected to the network.

This is contrast to homogeneous IoT networks where all the IoT devices are identical. Fluctuations occur because IoT devices may operate under time of use policies to save energy or devices fail in the environment or their up link to the Internet may be temporarily off line. Being able to reliably predict resource utilization in a dynamic environment can help overcome some of the challenges with precisely scaling IoT networks. Prediction with high accuracy allows us to plan ahead for changes in demand. For e.g., some critical IoT devices that are used in the medical field require that the data be processed with minimal latency [4]. If the CPU utilization becomes too high, it may impact the time to process the data because the CPU is busy handling many tasks.

1.1 Motivation

To investigate and learn about the behaviors of IoT networks at scale is not economically feasible and research in this area is largely fragmented as others have attempted to create simulation or emulation tools to test and evaluate only partitions of an IoT network and in some cases only for a specific scenario [5–8]. The primary challenge with designing these research tools is how to create a platform that can be used to evaluate the behavioral performance of a diverse set of applications and devices from end to end, in a uniform way [9]. Without collecting usage data in a uniform way, the resources metrics we collect may not be comparable because different sampling methods are used. For example, CPU utilization can be polled from the hardware, the operating system, the container level or the application level. This could yield different results depending on which processes are included in the calculation of the CPU usage metric. Also, each application will use memory and CPU in a different way because each application may have different processing patterns. Sometimes these applications may be installed on different hardware where the type of available computing resources is affecting the performance of the application. To the best of our knowledge there is no existing application that can provide an end to end evaluation of an IoT network that can also be installed on the production hardware. Most applications that provide IoT simulation/emulation are constrained to a particular test environment or local machine. Towards the goal of providing a solution to scale and evaluate IoT networks in a uniform way, we propose a virtual lab, Emulated Internet of Things or (EMU-IoT). EMU-IoT is an adaptable architecture and a smart testing framework that network designers can implement that will allow them to reliability scale an IoT network autonomically. Autonomic in this context is defined as system that is capable of continuously monitoring and having the ability to take corrective action to maximize the performance

goals of the application. In this case, corrective action would be detecting that a target utilization has been reached and a scaling decision needs to be made to maintain the quality of service on the network. A quality of service goal would be defined by the user. For example a corrective action could be, based the goal of preventing the CPU utilization from going over a certain value which has been known to affect response time, a decision could be to horizontally scale by adding a new virtual machine thus giving us more CPU cores. In this work, we focus on investigating the optimal scaling methods for the application infrastructure based on quality of service goals in IoT networks. The investigation is primarily about how to best to arrive at the trigger point for a scaling decision based on the number and types of IoT devices that are operating in the network. Our overall goal is to provide an emulation environment so that others can install our platform to be able to investigate how their IoT application will perform on their specific network.

1.2 Research Objectives

Our work is focused on understanding the behavior of highly dynamic IoT networks to discover the bottlenecks and being able to more precisely predict the resource utilization. The lack of knowledge regarding bottlenecks in the IoT networks can cause the quality of services of applications to degrade. Quality of service factors may be important to certain types of applications that require priority processing where users require fast answers to queries. For example, if it has been agreed that a query on a set of IoT data should take no longer than 3 seconds to complete, then we should be able to predict at what resource utilization level the query takes longer than 3 seconds complete. To do this we need to develop a method of experimentation and to create

a process to evaluate the smart testing framework. We can achieve these objectives through answering the following research questions:

- **RQ-1**: *Is it possible to build a system that can be used to experimentally identify the bottlenecks and test the capacity of large scale heterogeneous IoT networks?*
- **RQ-2**: *Can we trigger an adaptation by using a prediction algorithm to propose a corrective action before a service level violation occurs in a dynamic IoT network?*

Research Questions in Detail:

RQ-1:

Awareness of bottleneck points is usually unknown before a system is deployed. This is mostly due to the reason that once a system has been released into production the operating environment is not exactly the same as the development environment. Therefore it is crucial that we have tool and a process for evaluating these conditions by being able to replicate all aspects of the network infrastructure and the applications running on that network.

RQ-2:

Applications and the devices that are present on an IoT network are constantly subject to change. Since the volume and variety of IoT devices on a network can be dynamic, it becomes a challenge to predict the need for computing resources. Therefore an investigation and comparison of bottleneck prediction methods is an important process that IoT network architects must undertake before one can be confident that the system will perform according to established quality of service levels.

1.3 Thesis Contributions

In this section we discuss the contributions of this thesis to the field of IoT and the performance modeling of applications that support IoT networks.

We demonstrated that we can build a customizable virtual lab (EMU-IoT) that can be used to model heterogeneous IoT networks on an adaptable architecture. We are able to deploy all the necessary components to have a fully working solution that is reproducible by others. The software is executable on nearly any cloud computing service and can be installed without the need of any specialized hardware. EMU-IoT in its present state is capable of monitoring any IoT application as long as it is containerized and can easily be adapted to monitor non-containerized applications.

A methodology and specification was created to define what an emulated IoT device is. We provide a generic design and a minimum set of characteristics that an emulated IoT device must have. This allows EMU-IoT to be able to be extended so that others can create any software defined version of a physical IoT device to meet their custom requirements.

An IoT application resource utilization prediction engine was developed based on a Smart Testing Framework for Adaptation. This allows us to execute repeatable experiments to learn about IoT device resource utilization so that we can trigger adaptations to add or remove computing resources. The framework also allows for new prediction modules to be implemented including learning models through a systematic collection of historical data and analysis. For example, we can set the data collection feature of the framework to continuously monitor all changes in device count and type on the network. We can then store this information in a persistent database store to generate a large dataset. In theory, the more data we can use to train a model the more the accurate that model should be in predicting resource utilization.

Validation experiments were conducted to show that the system can detect bottlenecks and predict CPU resource utilization based on a set of requirements. We successfully replicated three common scenarios on IOT networks. First we modeled homogeneous IoT networks where we have a set of IoT devices that only produce small amounts of data. Then we experimented with only IoT devices that generate large amounts of data. Lastly we modeled heterogeneous IoT networks were we had a combination of devices that produce small and large amounts of data. We executed two types of experiments. First, we executed experiments using an exhaustive search method on three dimensions, with a Temperature and Light sensor, an IoT Camera and a combination of both IoT device types. The goal of these experiments was to determine what amount of devices would it take to reach a target CPU utilization. The experiments were executed successfully and we showed that we could detect bottlenecks for a chosen target. This data was then used as input data for a linear regression prediction model to find the device count for a unknown CPU utilization target. We were able to show that predicting the usage patterns of light weight IoT Devices can be done with linear regression models with a high degree of accuracy(within 0.02% of our goal). Based on these experiments we can also show that linear regression models become less accurate for prediction in heterogeneous IoT networks where we have a combination of devices with different characteristics.

1.4 Thesis Organization

This thesis is structured as follows. In Chapter 2 we discuss the background and related work of what others have done in the field of IoT research. In Chapter 3 introduce our design for a virtual lab (EMU-IoT) to execute experiments on adaptable architecture. In Chapter 4 we discuss the implementation of the both the hardware and software systems. In Chapter 5 we discuss the validation experiments and the results. In Chapter 6 we conclude and discuss future work.

Chapter 2

Background and Related Work

In this section we provide an overview of cloud computing, microservices architectures used for rapid instantiation of applications. We discuss technologies that will allow us to create emulated IoT devices and the supporting application infrastructure. We also discuss other simulation tools for IoT networks as it relates to our work. To evaluate our methodology, we discuss different algorithms that will be used to scale the heterogeneous IoT network so that we can achieve the goals described in our research questions.

2.1 Background

2.1.1 Cloud and Edge Computing

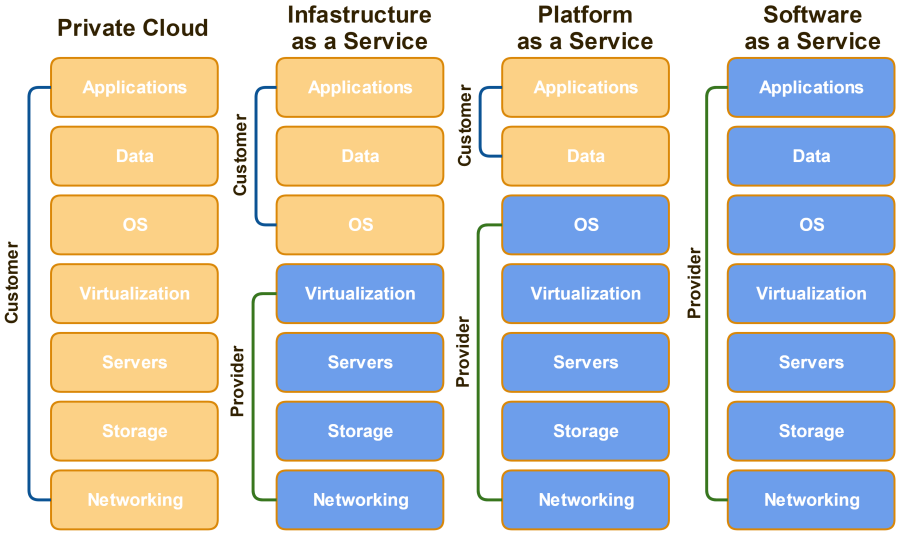

Figure 2.1: Cloud Computing Service Models
Adopted from[1]

Cloud computing refers to a combination of both software and hardware to deliver computing as service to the public [10]. The concept of the cloud is meant to represent an infinite set of computing resources that can be reserved and released on demand [11]. Users may create applications that run on the cloud. As shown in the figure 2.1, several services can be run in the cloud. In this case the type of cloud service is called Software as a Service. In other scenarios, users may have more fine grain control of the services and request compute resources such as CPU and memory to run a virtual machine or container. In this case the type of cloud service would be Infrastructure as a Service or Platform as a Service. In these cases both the hardware resources and the software is provided to the user without specific knowledge of the physical underlying infras-

[1] https://www.sevone.com/white-paper/monitoring-cloud-infrastructure\-performance-eliminate-visibility-gaps

tructure [12]. As the user is abstracted away from the physical machine, the resources can be provisioned from a diverse set of geographic locations. Clouds can be private or public. Private clouds are computing resources that are exclusively for the use of a single organization. In contrast to this are public clouds that provide computing resources to the public at large. Examples of public clouds are Amazon Ec2 and Google Compute Engine. Some clouds provide a testbed environment to support research into the behaviors of computing resources and provide a platform for the evaluation of next generation services and applications. The cloud that we have available to us is SAVI (Smart Application Virtual Infrastructure) that will provide computing resources for our experiments with IoT networks [13]. The SAVI cloud is a set of computing resources that is shared by several Canadian Universities located across Canada. It is a hierarchal design with a centralized core containing large amounts of computing power and smaller lower powered edges located at each university. This gives us the ability to build and execute experiments that require geographic computational awareness. Since IoT devices can be located in many different locations and require connectivity, this is an important feature to have when choosing a cloud environment.

An increasingly common use case for cloud computing has been to connect a large variety of everyday devices to the Internet so that they can transmit their data. For example thermostats, fridges, etc. These are also known as IoT devices. The usage pattern for these devices is the data is sent to the cloud for processing and in some cases a response is sent back to the device. Depending on where the cloud is located, this can increase latency. The proposed solution is to bring the processing power closer to the device [14]. This has created the concept of edge computing where applications are designed to take advantage of computing power on smaller clouds that process data

[2] `https://hackernoon.com/edge-computing-a-beginners-guide-8976b6886481`

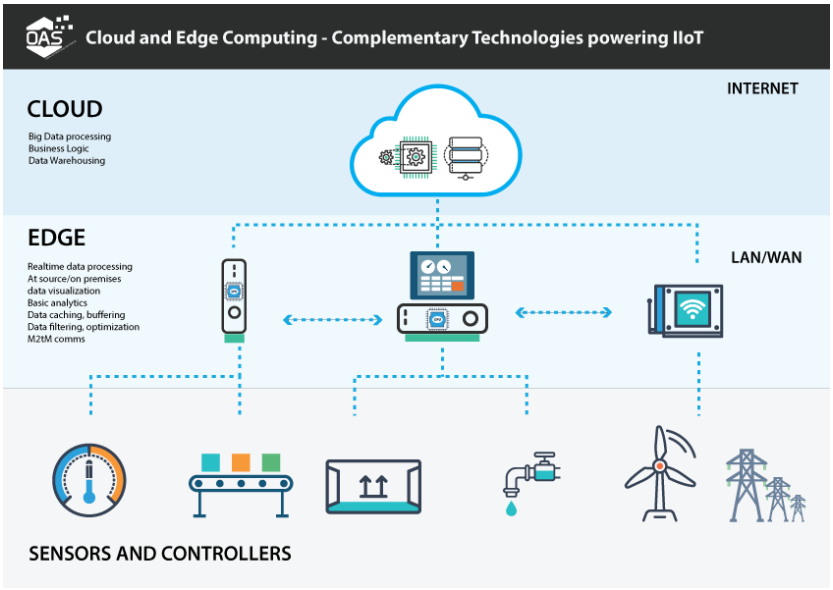

Figure 2.2: Edge Computing Architecture
Adopted from [2]

locally. As shown in 2.2 the edge clouds also aggregate data and send the information back to the central cloud for long term storage and future processing.

2.1.2 Internet of Things (IoT)

As an emerging subfield in computing, the concept of the Internet of things (IOT) can be defined differently depending on the application domain and the perspective of the participants. For example, for end consumers (the public at large) IoT can mean having new smart home devices such as fridges that reorder food and thermostats that stream data to cloud and can be controlled with smart phones [15]. In contrast to this perspective, IoT in industrial applications can mean upgrading legacy systems to become connected the Internet or the creation of new devices that are developed with Internet connectivity from inception [16]. IoT devices can be broadly placed into two categories, devices that emit data and devices that can also actuate based on instruction from an internal state change or a remote controller. In both cases these devices

produce data that can be used to improve the value that the IoT device provides over traditional non-connected devices. For example, a non-IoT enabled security camera that records video is limited because the video cannot be analyzed for a pattern detection. Whereas an IoT camera can stream its data to a remote location where analytics can be performed, and an action can be taken as a result of that analysis. Another facet of IoT can be described as the underlying infrastructure that supports the devices themselves. Device-to-device communication and external connectivity are common characteristics for IoT devices. New network topologies and protocols have been created to specifically support the communication needs of IoT devices as they do not behave like typical computing devices such as personal computers and servers. Devices may be low powered, geographically dispersed and be located in the external environment [17]. Another characteristic of IoT networks is that they can be heterogeneous. Heterogeneous in the context of IOT networks can be defined as having devices with different characteristics operating on the network simultaneously. IoT devices vary widely in terms of function and behavior. We can have sensors that produce small amounts of data and very fast frequencies. In other cases we have can have devices that produce large amounts of data infrequently. We may also have devices that continuously stream data on a constant basis. Due to this variety of data stream patterns, the network that supports these devices must be able to adapt to changing requirements as new devices are added and removed from the network. One approach to solving this issue has been to use a containerized method which is a software based approach to allow different gateways to be added dynamically when a new IoT device is added to the network [18]. The gateway is a service that provides connectivity between IoT devices and the processing infrastructure. The processing infrastructure consumes, stores and makes the data available for use by analytics services or other systems. Another

facet of the IoT network that can be heterogeneous is the gateway hardware itself since low powered devices such as Raspberry Pi's and Arduinos' can be placed closer to the IoT devices compared to full sized servers [19]. Exploring different techniques to determine which is the best type of device to use in terms of cost reduction, while maintaining quality of service, is an important factor when designing an IoT network.

IoT devices themselves may also become part of the processing infrastructure as devices gain more CPU power [20]. This is seen as new generation of IoT devices that gives a device dual purpose as opposed to the typical scenario where a device has a single purpose. An example of this would be a simple sensor that reads temperature which has almost no processing power and just emits data. Our design will need to have the capability to accommodate these new devices.

2.1.3 Microservices

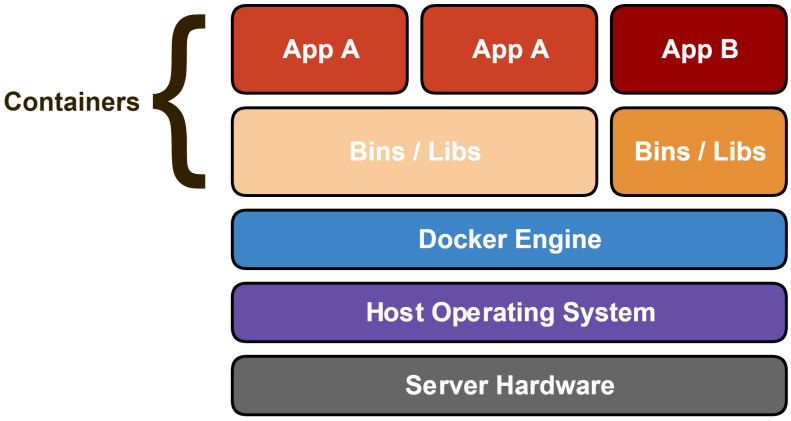

Figure 2.3: Microservices Architecture
Adopted from[3]

A key challenge when deploying applications in a cloud-based network is provisioning time. Traditionally this would be done by first creating a virtual machine on a

[3] http://www.bigdatatraining.in/wp-content/uploads/2016/06/dockerc.png

physical server. Once this was completed all the supporting operating system and application frameworks needed to be installed. This can take a significant amount of time and it also presents another challenge of how to reliably run many applications simultaneously on the same physical server/virtual machine. A microservices architecture uses a containerized approach and has been proven to be faster to provision compared to physical and virtual machines. As shown in figure 2.3 containers essentially encapsulate a process with the required dependencies so that it can be run in isolation inside of a virtual machine. In IoT networks, we can have many different types of applications running inside containers that share the resources which is more optimal compared to deploying one application per virtual machine or server. [21]. Another key feature of using microservices is the ability to orchestrate the creation and destruction of these processes. Docker is currently the most widely used container platform and has been used for simulation projects by others [22, 23]. It provides an API that allows you to programmatically remotely instruct clusters of servers to execute functions related to managing containers. The simulation tool we seek to develop will require such capabilities.

2.1.4 Big Data

Current projections for the number of IoT devices to be connected by the year 2020 is in the tens of billions [24]. This represents a massive increase in the amount of data that is generated from these devices. One of the primary drivers for connecting these devices is to be able to perform analytics to gain insight into trends and detect patterns so that timely action can be taken. Achieving these goals require that the corresponding infrastructure be able adapt to such queries by maintaining quality of services. For example, a video camera can generate gigabytes of data in a single hour, which is a

tremendous amount of data to process if you have hundreds of cameras on a single network A number of database and streaming technologies have emerged ever since Google began to index the web in the early 1990's. Variants of this technology exist today in the form of distributed databases and processing frameworks such as Hadoop, HDFS, Storm, Spark and Cassandra [25]. Selecting and evaluating a technology for a given application will become even more important as the traffic continues to grow. IoT traffic tends to be very dynamic and distributed applications that have a big data component need to adapt to these changes through scaling mechanisms to optimize available computing resources [26]. It is critical that researchers have a methodology that can accurately model their big data needs. In our work we seek to provide a platform that will enable these types of experiments.

2.1.5 Prediction Algorithms

A common trait in IoT networks is the variability of traffic patterns due to the diverse types of devices and fluctuations in the number of devices. As previously mentioned the goal of this work is to determine how we can adapt changes in a heterogeneous IoT network while maintaining a set quality of service goals. To be able to achieve this goal we need a methodology to examine how the different IoT devices affect the performance, and to be able to identify the bottlenecks when the volume of the workload (the number of IoT Devices) and the mix (the type of IoT Devices) change. For this, we can use algorithms that predict the utilization as a function of workload (mix or/and intensity)

Linear Exhaustive Search

To drive the following prediction algorithms a complete search of the solution space is needed. Although this form of search can be highly compute intensive

it is good for finding small variations in the variable under observation [27]. The dataset that is generated through this method will be used to train other models.

Linear Regression

Using a linear regression approach, we will attempt to predict the CPU utilization for a given set of IoT devices. This would include a configuration where we have a single IoT device type such as a temperature sensor and a scenario with a mixture of device types, for example a temperature sensor and a camera that streams images. This method of predicting CPU utilization is a widely used statistic when evaluating cloud-based systems [28].

2.2 Related Work

There have been several initiatives towards creating a tool that can reliably and accurately simulate IoT networks. Chernyshev, identified 3 types of IoT simulators that are actively being researched [29]. Full stack simulators, Big Data Processing simulators and Network simulators. Full stack simulators are defined as simulators that provide end-to-end support for devices and applications. Big Data simulators focus on using cloud computing resources for big data processing in an IoT context. Lastly Network simulators focus on network traffic and evaluating different protocols that are typically used in IoT networks. We discuss the current state of the work done in these categories and compare them to the goals of our IoT Emulator. Our approach is to combine all three types of simulators to create a comprehensive solution.

In the full stack category, we have Devices Profile for Web Services Simulator(DPWSim) which allows users to define and create simulated IoT networks [6]. It provides a robust set of tools, however the platform is limited to using DPWS(Devices Profile for

Web Services) standards which is based on WSDL(Web Services Description Language). Our goal is to create a tool where any communication standard can be used. In addition to singular protocol being used in DPWSim, it also implements the SOAP which is mainly used in defining message exchange rules between enterprise applications. SOAP messages incur a large amount of overhead by design. IoT protocols are typically designed to be light weight due to limited available processing power and bandwidth. In our work, the goal is to implement emulated devices that use IoT type protocols such as messaging over MQTT or HTTP. Also, in the full stack category we have iFogSim which allows for end to end simulation of devices, edges and the processing infrastructure for IoT networks [30]. While this toolkit provides all the features of a simulation environment it does not use the actual hardware to execute the simulation making the results of experiments difficult to compare with a physical system. Our work improves upon this model by providing a tool that can be executed on the actual network where the real system will run on.

In the big data simulator category there is IoTSim and SimIoT. In the case of IoTSim, it does provide the capability to simulate large scale IoT networks but has two major drawbacks [8]. The first limitation of of IoTSim is that the workload generation process is only based on the MapReduce model. While this is a common scenario when processing IoT data, many other types of scenarios exist. For example, image recognition of live video streams requires a completely different set of steps required to analyze the data as compared to batch processed numerical data such as those generated from environmental sensors. Moreover this limitation leads to the inability to emulate heterogeneous IoT networks since we can only support devices that generate MapReduce type data. In our approach we have an emulator that is capable of evaluating the performance of *any* IoT big data application as long as it can be containerized.

The second limitation of IoTSim is that it cannot be executed in the actual environment as it is also extension of CloudSim meaning it has same limitation as the previously mentioned simulator iFogSim. This is a significant drawback because big data applications all behave differently depending on what type of hardware is available to them. For example, Spark is a memory intensive application and Cassandra is a disk intensive application. If we only have machines with large amounts of disk space and small amounts of memory, Spark will perform poorly compared to Cassandra. We need a way of identifying these types of bottlenecks which our platform can detect. In the case of SimIoT it also lacks the support for heterogeneous IoT networks [31]. This is a key characteristic of IoT networks where many types of devices are present. As previously mentioned, in our approach the platform will currently support any type of software defined version of and IoT device without requiring changes to the underlying platform. SimIoT also lacks the ability to track QoS metrics to drive network optimization such as adding or removing computing resources. In our approach we solve this by implementing a smart testing framework.

Lastly, we have the network simulator category. This category has the most activity. Network simulation is a well-researched area that predates IoT research by several decades, so it makes sense that tools that were designed to simulate network traffic are being extended and repurposed to IoT network research. One such popular tool is CupCarbon which has been extended to include IoT features for emulation [5]. While this emulator can execute a simulation on real hardware it is limited to running on Raspberry Pi's and requires modification to work on each new platform. One of our goals is to make our emulator platform agnostic so that it can be installed on any cloud provider. Another popular tool is Cooja which is used primarily for simulating wireless sensor networks [7]. This simulator overcomes the limitations of CupCarbon but it only

allows simulation of the devices and not the applications that process the sensor data. You would still need another simulator to handle the data ingestion functions making it difficult to evaluation an end to end scenario on an IoT network.

2.3 Chapter Summary

After conducting the background research and literature review, we have discovered several approaches and key technologies that will support our goals of answering our research questions. First, we discovered using that using microservices, namely the Docker implementation will allow us to create and run experiments as others have successfully have used the technology for this purpose. We also learned about the variations in the composition of IoT networks. We determined that our approach needs to be flexible enough to handle wide variety of IoT devices and different topologies. This knowledge will guide our approach in designing a system that can be easily extended to meet to the future needs of IoT networks. An important component of our system is the resource prediction mechanism since we want to answer our research question regarding be able to provide an optimal way to trigger an adaptation in a dynamic IoT network. Lastly, we investigated the related works in IoT simulation and discovered that there are three major types of simulators/emulators, namely Full stack simulators, Big Data Processing simulators and Network simulators. We found that there are many solutions that provide good ways to simulate or emulate various subsets of an IoT network but there appears to be no system that provide functionality to evaluate and end to end solution. Additionally, many of the simulators/emulators don't allow others to make use of the actual hardware that the system will run on. This is an important gap that our work will seek to fill.

Chapter 3

EMU-IoT System Design

The surge in the number and variety of devices has created a challenge for the systems that support the data streams from IoT sensors. To transform raw data into meaningful and actionable intelligence we require fast methods to ingest and process IoT device data. The variety of IoT devices adds to the complexity of designing an architecture that can be flexible enough to handle new devices and provide good scalability as the number of devices increases. Designing a suitable architecture to meet specific quality of service goals requires that we have a model that can be analyzed so that we can determine if those goals are being met [32]. The feasibility of executing large scale physical models can be very costly and often experiments rely on analyzing a scaled down version of the system to be [29]. The primary issue with using a scaled down physical model is that results may not be the same as running the experiments on the actual system. Due to the limitation in the value of the results generated by a scaled down model, an architecture cannot be deemed reliable for implementation because it has not been shown to support that system. We propose a virtual IoT lab (EMU-IoT) which is lightweight, platform agnostic and easily configurable to execute experiments

that nearly replicate the system to be. When deploying an architecture to support sensor streams, we typically look to evaluate that architecture based on the performance bottlenecks of reading those sensor streams. This also includes processing the data and executing queries. In the proposed virtual lab, we address these challenges by providing the ability to create software versions of IoT devices and the ability to configure workloads for a given use case, that is executable on any computing infrastructure (ex. local network, Amazon Ec2, or Microsoft Azure). For example, if we have environmental sensors, we may want to see how many sensors we can support using a specific set of hardware and then run queries on that data. A model using both the true number and type of sensors is much more meaningful because it is a closer representation of the system to be.

3.1 A Customizable Virtual Lab

In this section we describe the properties that a virtual lab should have to be able to emulate real world IoT networks. The aim of this work is to be able to emulate the topology and behavioral patterns that may occur in an IoT network.

A) *Geographical Distribution*

The very nature of IoT devices is that they are dispersed throughout the environment meaning that they are externally located away from the centralized computing infrastructure. For example, you may have a centralized analytics application located in one city, but you may have IoT devices spread across many cities and countries. This is done to reduce the cost and complexity of the IoT network. Therefore, to replicate this scenario and get results similar to a production deployment we need an emulator that is capable allowing others to install

the software onto their physical production hardware rather than in a test environment where the results generated could be different.

B) *Temporal Distribution*

IoT devices are typically configured to suit the usage patterns of the environment in which they are deployed. For example, during the day a temperature sensor may be configured to take more frequent readings because during work hours more humans can mean more temperature fluctuations in the space. Since we have more frequent readings, the load on the network becomes larger. Conversely at night, the temperature is much more stable because everyone has gone home and now we can configure the sensor to take readings every hour instead of every 5 minutes during work hours. An emulator for IoT networks will need to have this feature to replicate this scenario.

C) *Heterogeneity*

Perhaps the most obvious characteristic of IoT networks is the variety of device types that exist in the environment. Many new devices are being deployed and others are being decommissioned on a regular basis. This presents two main challenges. First, we need an emulator that is easily extendible and generic enough so that someone can create new emulated IoT devices. Second, we need to design the emulator in such a way that it does not disturb the stability of the existing network when we remove and add new devices. This is a critical feature for the emulator, because for real world networks downtime is not acceptable.

D) *Network Connectivity Variety*

As previously mentioned there could be many types of sensors on an IoT network. These devices may also use different technologies to establish connections

to the cloud network. For example, you could have Bluetooth, Wi-Fi, Ethernet etc. These different types of connectivity have different bandwidth speeds. An IoT emulator would also need to have this feature so that we can compare the delay and bottlenecks between using different network connectivity types.

E) *Network Protocol Variety*

IoT networks can also be diverse in terms of the protocols that are used to provide communication between devices and the computing infrastructure. Additionally, existing protocols are always being updated and new ones may appear in the future. One of the main goals for our emulator is to abstract the design in such a way that you can use any communication protocol. No reconfiguration should be necessary to implement a new protocol or to have multiple protocols running at the same time.

F) *Infinitely Scalable Design*

A common theme appearing in the research of IoT networks and is that the number of connected devices is growing exponentially and this trend is expected to continue for the foreseeable future. A major concern for application developers and network designers is how to prepare for the growth ahead. There are two major concerns in this problem space. First how do we quickly scale to meet the demand and how do we avoid having idle resources in our infrastructure that will unnecessarily increase cost. An emulator must have the capability to deal with these two problems. Towards this goal we develop a prediction engine that will help to better understand these usage patterns in IoT networks.

3.2 Device Properties

A virtualized IoT device must fully emulate the behavior and characteristics of the actual device. In this section we describe in the context of IoT emulation what are the three generic properties that a virtualized IoT device should have.

1. *Connectible*

   IoT devices vary in terms of their ability to connect to a receiving device that will read the emitted data. The typical connection types are over, Bluetooth, WIFI, and hardwired (serial, Ethernet). A receiving device could be a controller that aggregates several sensors or other IoT devices that communicate in a peer to peer topology. The consequence of this is that different connectivity modes may have different transmission rates of speed. For example, transmitting data over Ethernet is faster compared to WIFI and Bluetooth. To deal with this variation, a virtualized IoT device must have the ability to set its transmission type so you can model the actual physical device. When instantiating an IoT device in the EMU-IoT environment, the connectivity type can be set at run time.

2. *Configurable*

   Since IoT devices also vary greatly in terms of the types of on board sensors (temperature, movement, video, audio, etc.) developing a software-based component that is configurable is more practical rather than several individual devices. In the first iteration of our design we focused on creating a virtualized IoT device with the ability to configure the emission rates to simulate different connectivity types and setting the reading ranges for a temperature and luminosity sensor. In future iterations we hope to expand the number of generic sensor types that are supported.

3. *Deployable*

   The most important feature of the EMU-IoT platform is the ability to rapidly instantiate IoT devices. To achieve these goals, we use a microservices approach by containerizing the software component so that it can be quickly activated. A virtualized sensor is a small lightweight application that begins emulating the behavior of an IoT device once a container has been started. This primary advantage of this that an unlimited number of virtualized IoT devices can be activated and they can be selectively dispersed on different networks according to the needs of the user. Another advantage of using microservices is that you can have a common way to communicate with the containers even though they may be running different IoT software components inside the container.

3.3 Virtualized IoT

In this section we describe the two main components that are fully virtualized which enable workloads to be executed on EMU-IoT. These components can be rapidly instantiated and deployed at scale.

3.3.1 Device

A virtualized IoT device in the context of the EMU-IoT lab is an encapsulated service that emits data. It adheres to the three properties as described in section 3.2 which are connectible, configurable, and deployable. As shown in figure 3.1 we have three different components inside of a virtualized IoT device. At build time, parameters to modify the behavior of the device can be passed to the service running inside of the container. Once the configuration has been set, the data generator service begins to

produce the emulated data. This data is then encoded in the appropriate format for the transport protocol being used, and then the data is transmitted to another service that is external to virtualized IoT device. A virtual IoT device can also be configured to accept new data at runtime if the device needs to obtain new information once the data generation service has been started.

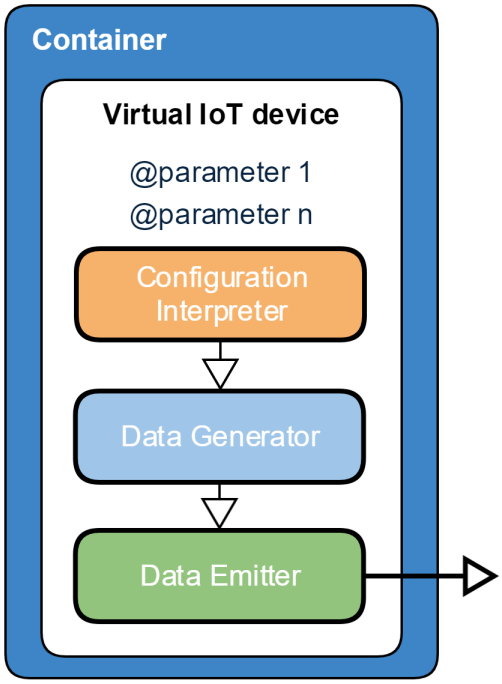

Figure 3.1: Virtualized Sensor

3.3.2 Gateway

A virtualized IoT gateway in the context of the EMU-IoT lab is an encapsulated service that receives, formats, and forwards data onward to an external service that collects data from many gateways. Virtual gateways may also support physical IoT Devices.

As shown in figure 3.2 we have four different components inside of a virtualized IoT gateway. At build time, parameters to modify the behavior of the gateway can

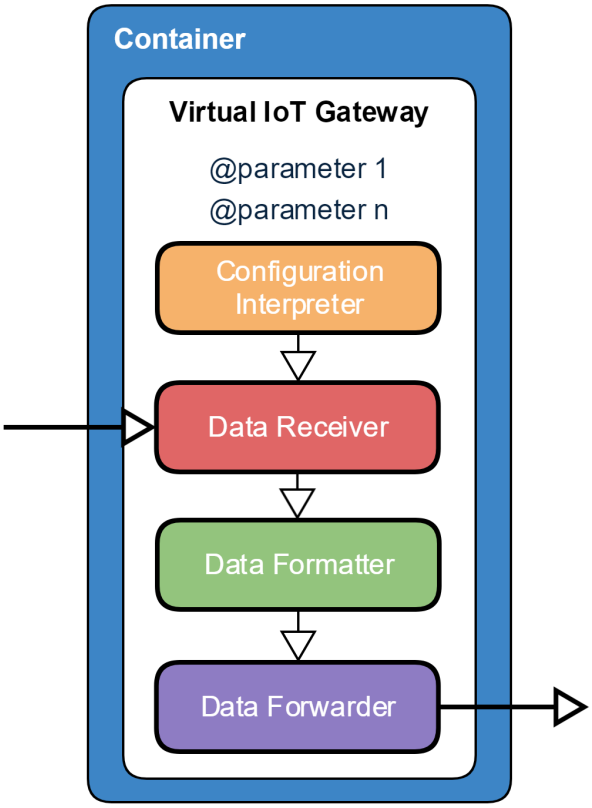

Figure 3.2: Virtualized Gateway

be passed to the service running inside of the container. Once the configuration has been set, the data receiver service starts and waits for incoming data from the physical or virtualized IoT devices. This data is then checked to make sure it is in the correct format and then each reading from the IoT device is formatted into a common message standard that is used on the IoT network. The data forwarder then makes a connection to a data aggregation service that is waiting for this information.

3.4 Network Architecture

In this section we discuss the network design that can support an emulated IoT environment that is comprised of virtual machines and both physical and virtual IoT devices. We also discuss the data pipelines that allow the movement of data between the participants in the IoT network.

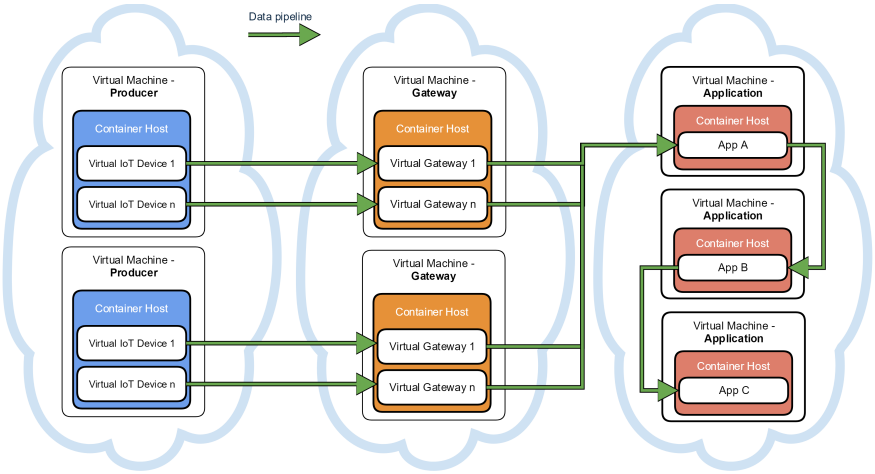

Figure 3.3: Network Architecture

In the IoT network we have 3 separate segments that operate independent of each other. Across all three segments the applications are running on a containerized architecture which we described in section 2.1.3. The use of containers is a mandatory requirement for both the producers and gateways, but the applications can be run directly on the virtual machine or directly on the hardware. As shown in figure 3.3 can see that the virtual sensors push their data to the gateways, then the data is forwarded to the application side. In this example, on the application side, the data is passed from App A to App B then to App C. These could also be standalone services that are not dependent on each other. We could also have more than three App's. The design in this part of the architecture is based on the needs of the users particular IoT application.

3.4.1 Producer Host

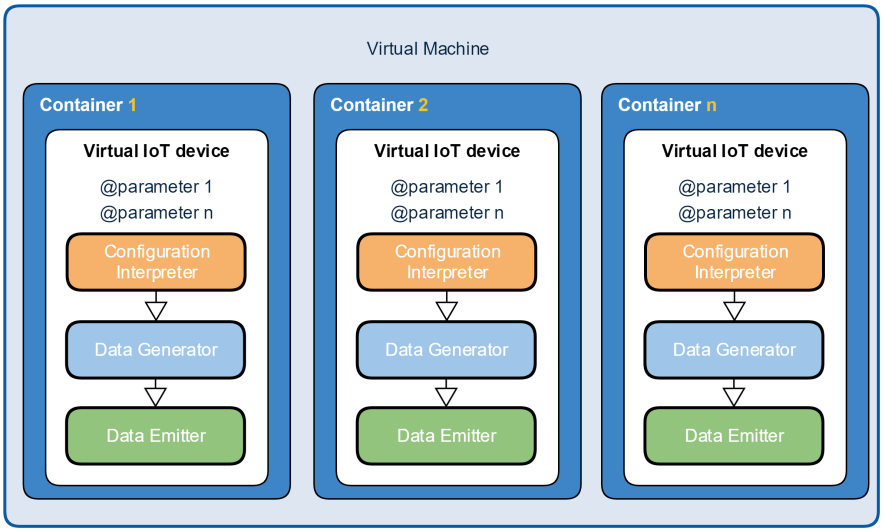

Figure 3.4: Producer Host

A producer can be defined as an virtualized IoT device that *produces* data that is emitted and read by an outside service. As shown in figure 3.4 we have a virtual machine that can host many instances of the virtual IoT device. The purpose of this design is to allow us to deploy IoT devices easily on any host as needed. The containers are designed to have a very small footprint so that large numbers of virtual IoT Devices can be hosted on a single VM. The producer once started, will emit the readings and send the messages to the receiver gateway (the controller). Creating IoT devices as one per container allows us to better emulate a standalone device which is the way physical devices operate. This is as opposed to creating several devices as threads sharing resources inside a single container when doesn't truly emulate a standalone device scenario . For example, a smart watch has its own base set of services and its communication component only provides connectivity for that one smart watch. It doesn't share its Internet connectivity link with other smart watches

3.4.2 Gateway Host

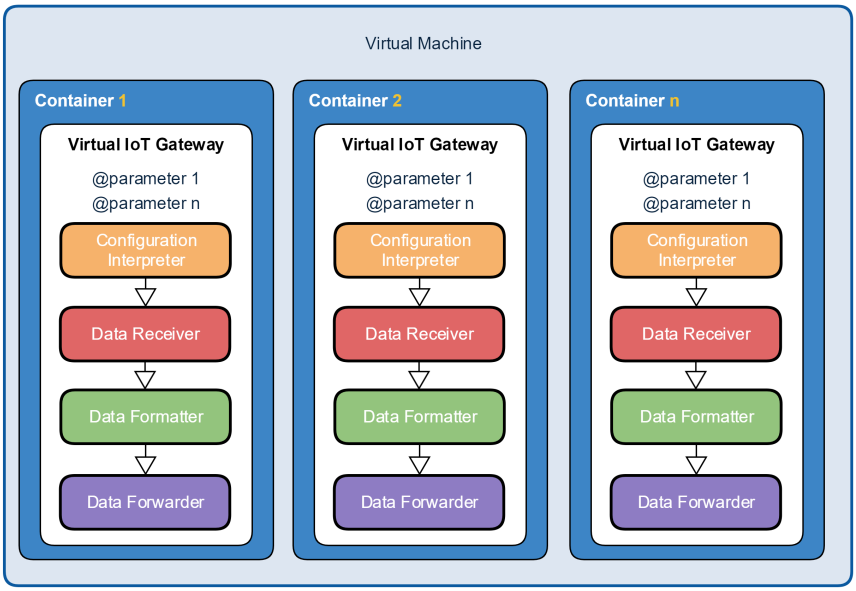

Figure 3.5: Gateway Host

Similar to the Producer Host we have a virtual machine that will host the gateways where IoT devices will transmit their data to. The gateways can be configured to receive data from many IoT devices. Therefore you do not need as many virtual gateways as virtual IoT devices. This is a key feature of EMU-IoT as we can experiment with different workloads on the gateways to determine the optimal number of IoT devices that can be supported on a given virtual gateway. This is an important feature because the fixed resource requirements for a virtual gateway are much higher compared to a virtual IoT device so we would like to optimize our computing resources by limiting the number of virtual gateways.

3.4.3 Application Host

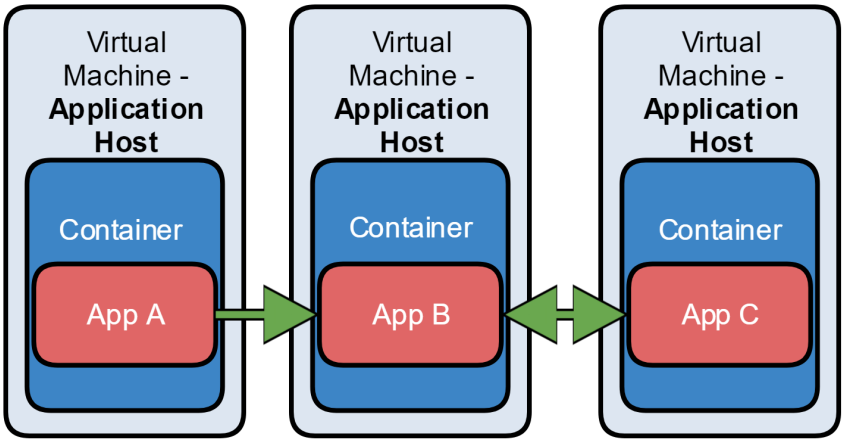

Figure 3.6: Application Host

The Application hosts in EMU-IoT represent the environment for the containerized applications that support the IoT services running on the network. Typically they are services that ingest, process and store the incoming data from the various IoT devices on the network. As shown in figure 3.6 they can run inside containers similar to the producers and gateways. Generally the resource usage is what we observe when evaluating IoT applications. Performance measures such as CPU utilization are monitored to maintain the quality of services on the network. As mentioned in our research goals we aim to build a system that will allow us to understand the behavior of IoT networks. Towards this goal we will be executing experiments designed to find the bottlenecks of these applications based on the CPU utilization.

3.5 Smart Testing Framework

In this section we describe the Smart Testing Framework and how it will help us to achieve our research goals of being able to intelligently trigger scaling adaptations in highly dynamic IoT networks. First we describe the Smart Testing State machine and the purpose of each state. Then we define what a bottleneck is in the context of an IoT network and how the Smart Testing Framework can be used to find it. Lastly we describe the how test cases drive the bottleneck detection process using different goals and decision algorithms. Once a bottleneck as been detected we can use this as threshold to execute an adaptation in the environment by adding more computing resources or removing computing resources. The actual process of scaling is a feature that is external to the Smart Testing Framework.

3.5.1 State Machine

The primary component of the Smart Testing framework is the state machine. It controls the process of creating workloads and detecting the bottlenecks. There are three distinct states that the machine can be in until it exits.

1. *Generate Test Case*

   Based on the type and configuration of a test case, a new case is created each time this state is reached. This means that a new set of IoT devices is created or removed and there will be a resulting change in the traffic on the network that can be observed and monitored. The test case can be generated based on a set of predefined rules or a set of rules generated by the prediction engine that will be discussed in section 3.5.5.

2. *Collect Resource Utilization*

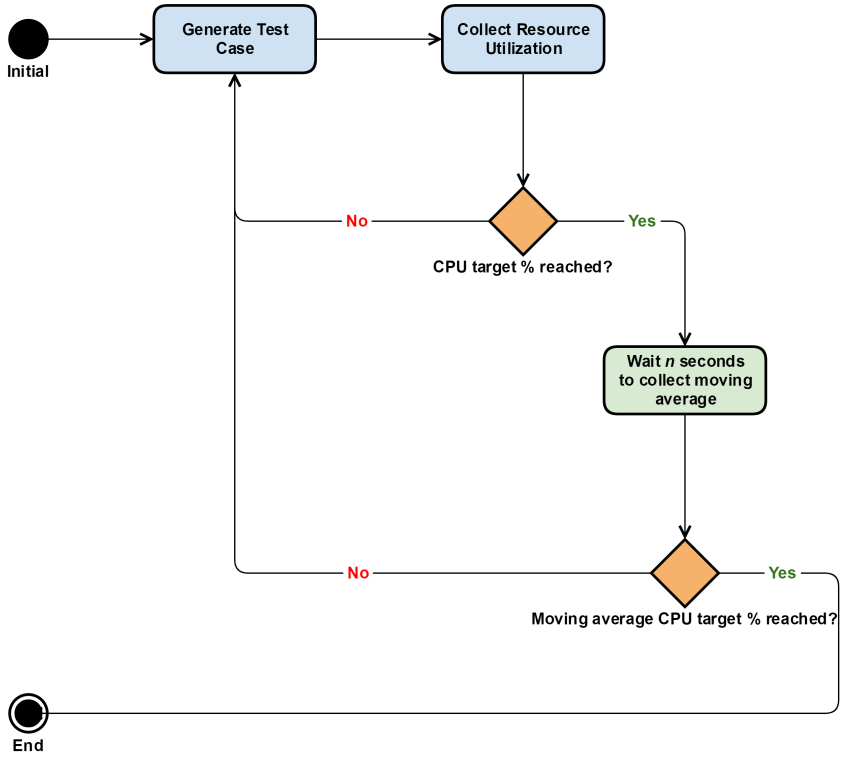

Figure 3.7: Smart Testing State Machine

In this state the live resource utilization is checked to see if the defined goal target has been reached. For example, if CPU utilization is the resource we want to check, then this information is polled from the server and checked to determine if it is greater than or equal to the goal (if we are looking at the upper boundary). We can also look for a lower boundary meaning a test case would now remove IoT devices instead of adding them.

3. *Collect Resource Utilization (Moving Average)*

   The purpose of this state is to temporarily pause the generation of new test cases to observe the resource utilization under more stable conditions. This is because after the test case has created new IoT devices they may require sometime to

initialize and begin creating traffic. This state is also important to average out random resource spikes in the IoT network due to application background processes that are unrelated to the IoT traffic.

3.5.2 Bottleneck Detection

The triggering of the end state in the state machine is that a bottleneck in the IoT network has been detected. A bottleneck in the context of the IoT network is any point in the application infrastructure which has breached the established quality of service level threshold. Bottlenecks are a threat to application stability. Since many applications operate on the network we can have multiple bottlenecks.

3.5.3 Test Case Definition

Test cases are designed to allow us to model and investigate the behavior of an application in response to a particular scenario on the IoT network. For example, in a smart building we may want to activate IoT environmental control devices during the day and shut them down at night. So, to emulate this scenario we would define the rules to generate a test case that creates IoT devices at certain times of the day. These test case rules are discussed in the next section.

3.5.4 Test Case Types

In section 3.1 we described several features that EMU-IoT will have to emulate different IoT usage scenarios. To recap these are Geographical Distribution, Temporal Distribution, Heterogeneity, Network Connectivity Variety, Network Protocol Variety, and Infinitely Scalable Design. We can define each of these as a test case to emulate the scenarios.

```
Generate "n" IoT devices on Producer host "A" located on Edge "1"
Generate "n" IoT devices on Producer host "B" located on Edge "2"
```

Figure 3.8: Rules for Geographical Distribution

In figure 3.8 we model the scenario where we create IoT Devices on different edges. These edges would be located in different locations such as different cities

```
Check Time of Day
If time of day == "business hours"
    Generate "n" of IoT Device Type "A"
If time of day == "after hours"
    Remove "n" of IoT Device Type "A"
```

Figure 3.9: Rules for Temporal Distribution

In figure 3.9 we model the scenario where we create IoT Devices during the day and then remove them during the night time to conserve resources.

```
Generate "n" of IoT Device Type "A"
Generate "n" of IoT Device Type "B"
```

Figure 3.10: Rules for Heterogeneity

In figure 3.10 we model the scenario where we create different types of IoT Devices. EMU-IoT is capable of supporting an unlimited number of IoT device types.

```
Generate n of IoT Device Type "A" ("configure with throttle to limit
    bandwidth usage")
```

Figure 3.11: Rules for Network Connectivity Variety

In figure 3.11 we model the scenario where we create IoT devices with the configuration parameters passed to the service so that the user can select which network

connection type to use, eg. Wifi, Bluetooth, etc.

```
Generate "n" of IoT Device Type "A"(configure to use "MQTT")
Generate "n" of IoT Device Type "B"(configure to use "HTTP")
```

Figure 3.12: Rules for Network Protocol Variety

In figure 3.12 we model the scenario where we create IoT devices with different protocols. HTTP is the most common protocol on the Internet. MQTT is a protocol commonly used in IoT networks.

```
while CPU Utilization <= "80%"
    Generate "n" of IoT Device Type "A", every "5" seconds
if CPU Utilization in Application "A" > "80%"
    Trigger scaling action to add more resources.
```

Figure 3.13: Rules for Scaling

In figure 3.13 we model the scenario where we create IoT devices based on finding a bottleneck and then we trigger a adaptation scaling action. This is done via a prediction algorithm or workload generator which will be described in further detail in section 3.5.5.

*At the moment we must hard code these scenarios, in the future we hope to build a rule-based processing engine to generate the test cases.

3.5.5 Prediction Engine

Another part of the smart testing framework is the utilization prediction engine. Based on the related works there are many approaches that can be used for prediction. In this section we will discuss the primary components of the engine. The data processor, the predictor and the test case generator. Together they provide users with the ability build custom prediction algorithms and execute test cases to detect bottlenecks.

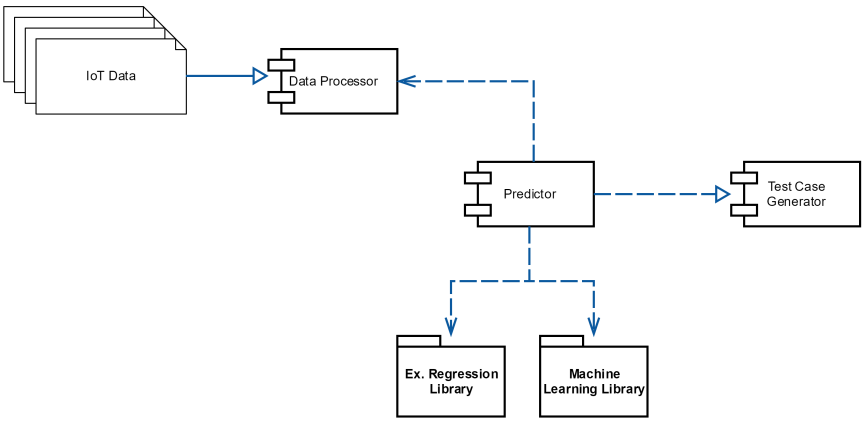

Figure 3.14: Prediction Engine

1. *Data Processor*

   The role of the Data Processor is to consume the historical usage information that has been collected. We can use this data to make predictions about usage patterns that have not yet been observed. The processor ingests, validates, formats the data and passes it to the Predictor

2. *Predictor*

   The role of the Predictor component is to apply a prediction algorithm on the incoming data that has been passed to it. For example as shown in 3.14 we can import a regression library and use it to generate a predicted value. In our experiments we used CPU utilization as the value we want to predict based on the number of active IoT devices on the network. This will be explained in detail in the experiment chapter.

3. *Test Case Generator*

   The role of the Test Case Generator is to pass the action to EMU-IoT. This action would be for example create a specific number of IoT devices as specified in the rules generated defined by the Predictor.

3.6 Chapter Summary

In this chapter, we have discussed the architecture of the system, the major system components and how they are related to meeting our research goals of building a system that can emulate IoT networks to drive adaptations. We introduce the concept of a Smart Testing Framework for IoT networks. This will allow use to execute experiments that can detect bottlenecks which helps with our research goals with optimizing resource usage by using intelligent scaling algorithms. The Smart Testing Framework also allows the system to learn from each iteration as it has more data to work with when using the Prediction engine to drive the test cases. The design is modular and allows for future extendibility while maintaining stability. This was a big challenge for us because we needed to identify subsystems and types of objects that needed to be configurable and yet not be heavily depended on other subsystems. In the end we were able to provide a true end to end solution (from the device side through all application layers) for evaluating IoT networks. In the next section we will provide more detail about how the design was implemented as a software system.

Chapter 4

Implementation

In this section, we describe the approach that was used to transform the design into a working system. We discuss how we met our research objectives of being able to produce a working emulator, the smart testing component and a tool to measure the performance of our system. The microservices platforms, infrastructure, programming language libraries, and the hardware devices that we used as a guide to implement our virtualized devices is also briefly discussed in this chapter. Lastly, the implementation of the main modules of EMU-IoT are discussed. The results of the experiments will be presented in section 5.

4.1 Requirements

To meet our research objectives and to carry out our experiments as described in the introduction section we require the following:

- A fully functioning set of virtual machines with Docker + all required applications installed.

- We need to use a programming language with suitable libraries.
- We need tools to measure the performance of the IoT network.

4.2 Hardware Configuration

IoT devices come in a variety of types and legacy devices are being retrofitted with Internet connectivity. In this section we describe the process of transforming two common IoT device types and define the data structures that they emit. We also discuss the virtual machines that were used to support our IoT application.

4.2.1 IoT Temperature and Light Sensor

As mentioned in the background section, environment monitoring is a common use case for IoT devices. Therefore to meet our research objectives of providing a platform for emulating IoT networks, we decided to implement an environmental IoT device. In our design we used the Texas Instruments SensorTag as the model for our virtualized IoT device that produces temperature and luminosity data. A physical model of the IoT device is show in figure 4.1. The TI SensorTag has several embedded sensors that emit readings with various data types as shown in table 4.1. The data types are all numerical which means that from a size perspective, each set of emitted readings it is quite small.

Resource Name	Unit	Type	Description
pressure	hPa	float	Pressure sensor air pressure
pressure_t	C	float	Pressure sensor temperature
humidity	%RH	float	Humidity sensor relative humidity
humidity_t	C	float	Humidity sensor temperature
objtemp	C	float	IR temp sensor object temperature
light	Lux	float	Light sensor illuminance
battery	mV	float	Battery voltage level

Table 4.1: SensorTag embedded sensors

Figure 4.1: Physical Sensor Tag

- *Features*

  The device works by simply emitting readings for each sensor type at a given interval, for e.g. every 1 second or 10 seconds. Since we know the data types and emission rates, we can reliably reproduce this data without the need for the physical device by designing a software version of the sensor.

- *Message Format*

  To be able to process the data in a standardized way we have defined a message format that is based on the JSON standard. This allows us to validate the message integrity as it gets passed between applications on the IoT network. Figure 4.2 is a an example of a temperature and luminosity reading. Since we use a common message format, that also allows us to store the information in the same database schema.

```
{
    sensorID:"IoTProducer1_IoT_temperature_sensor_65_3001"
   ,sensorType:"room_temperature"
   ,value:"16"
   ,timestamp:"1533663203"
   ,daydate:"20180807"
 }
```

Figure 4.2: Temperature Sensor Message Format

```
{
    sensorID:"IoTProducer1_IoT_temperature_sensor_65_3001"
   ,sensorType:"lux_meter"
   ,value:"192"
   ,timestamp:"1533663203"
   ,daydate:"20180807"
 }
```

Figure 4.3: Lux Sensor Message Format

4.2.2 IoT Camera

Another type of IoT device that poses challenges for network designers are devices that generate large amount of traffic. An example of such a device is a camera. Cameras are an important tool in monitoring the environment and cameras produce video streams that can be analyzed in real time. To understand the behavior of this type of IoT device, we created a prototype camera using the Raspberry Pi Kit that came with a camera module as show in figure 4.4.

- *Features*

  To create a virtualized representation of the IoT camera we documented several necessary characteristics for a fully working camera. The camera must produce video, must have a connection to the Internet and must be able to stream the video to a recipient. Towards this goal we created an application that stores a video file, we implemented several libraries for image processing and created a connector to a database to store the video data. The application creates streams by looping through the video file, breaking the video down into frames, encoding the images into a string based format and then writes this data to a table in Cassandra.

- *Message Format*

  The messaging format shown in figure 4.5 was created to transmit and store the data. We store each image in the database with a frame id. This is a concatenation of the

Figure 4.4: Raspberry Pi with Camera Module

camera id and timestamp. This allows us to reconstruct the video frame by frame if necessary. The advantage of storing the images by frame allows other applications to perform image recognition tasks without have to repeatedly break down the video into frames each time the camera data is queried.

```
{
    camera_id:"IoTProducer2_IoT_camera_68_3010"
   ,frame_id:"153467408132"
   ,value:"base64_encoded_value_here"
   ,timestamp:"1533663203"
   ,daydate:"20180807"
 }
```

Figure 4.5: IoT Camera Data Format

4.2.3 Virtual Machines

The EMU-IoT application is able to run directly on server hardware and also on virtual machines. As mentioned earlier our implementation was on virtual machines provided by the SAVI network. The specification of these machines and the software stack will be described in greater detail in section 5.1.

4.3 Software System Implementation

In this section we describe the software components that are present in EMU-IoT. As described in section 3.1 we have several applications that marshal the data from the IoT device to the database. We describe the process of creating virtualized controllers that allow the virtualized IoT devices to connect to the network. The controllers act as aggregation points in the network which is a hierarchical architecture that reduces the number of connections to the IoT network. We also describe an example big data application that we implemented using Apache Kafka[1], Spark[2] and Cassandra[3], that reads the IoT data and streams it into the database.

4.3.1 Applications

The applications in EMU-IoT can be split into two layers. In the first layer we have the microservices infrastructure that provides the hosting environment for our virtualized devices, gateways and business applications. The second layer is the actual virtualized IoT devices and the business applications that process the data from the virtualized IoT devices.

[1] `https://kafka.apache.org`
[2] `https://spark.apache.org`
[3] `https://cassandra.apache.org`

- *Docker*

  The the first layer is where we have the container provider service. We have previously discussed the role of microservices and how it helps our design, please refer to section 3.4. Docker was installed onto the virtual machines and was controlled remotely by our IoT Experiment Manager application that will be discussed in section 4.4.4. This part of the implementation was not customized beyond the standard features provided by Docker so we omit the configuration details. The instructions for setting up the environment can be found in Appendix A.

- *Kafka*

  Apache Kafka is a distributed application that is based on the publish and subscribe

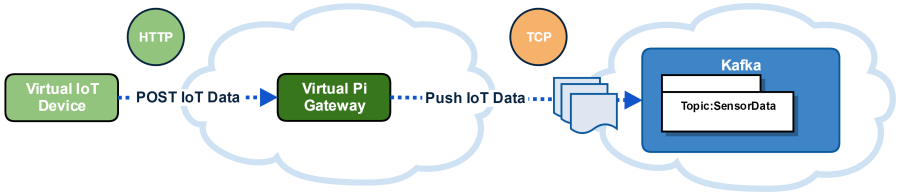

Figure 4.6: Kafka Implementation

data model. To realize this data model, Kafka nodes store *topics* that producers of information *push* data to and anyone interested in that data can *subscribe* to the topic and check the topic periodically to determine if new messages are available to be pulled. Kafka serves as a broker of information. The advantage of using Kafka in this context is that it has been designed to deal with large volumes of messages. In our IoT application we will be generating hundreds of messages per second. Another advantage of using Kafka is that the data can be set to be stored for a specified in period of time in case we need to reread the data. In our case we mark the message as read once the streaming job has downloaded the message. As shown in figure 4.6

the messages are first posted from the virtualized IoT device to the gateway. This transmission is done over HTTP is done via a POST transaction. The virtual Pi has a web server that reads the messages. The messages are then pushed to the topic we created called SensorData. The topic manager inside Kafka will assign the incoming message to the correct topic. The pull or *subscribe* process is discussed in the Spark section.

- *Spark*

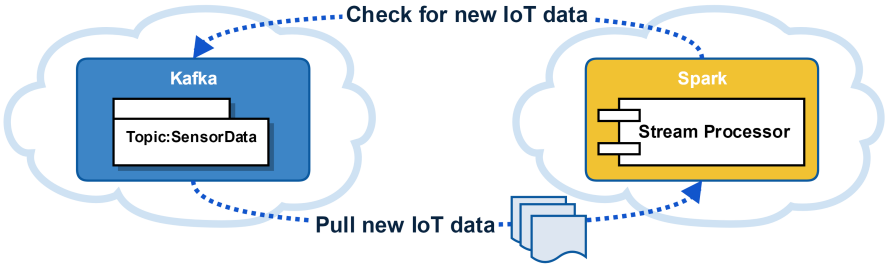

Figure 4.7: Spark Streaming Implementation

Apache Spark is distributed data processing framework that can be used to transform data from various sources. Transformations could be computations, movement of data from a source to sink. Generally any type of data manipulation can be performed. Users or applications submit jobs to Spark and the task is then executed in memory. The role that Spark plays in our application is a data stream processor. As shown in figure 4.7 the job runs on a 5 second interval where it checks the SensorData topic in Kafka. If there are new messages since the last check, the job pulls the new messages in and marks these messages as read. Once Spark has downloaded these messages it transforms them into objects that mirror the data format used in the Cassandra database schema. For here the last step in the job executes an `Insert`

operation so that the data can be serialized to the Cassandra database. The Cassandra database will be discussed in the next section.

- *Cassandra*

  Apache Cassandra is a NoSql distributed database that is suited for storing large

Figure 4.8: Cassandra DB Implementation

volumes of data. For our application, Cassandra serves as a persistent data store for the IoT messages. As shown in figure 4.8 the second task of the Spark streaming job is to format the data to match the schema we have created inside Cassandra. The details of the schema can be found in Appendix A.

- *Node Red*

  Node-Red[4] is an interactive web based visual development tool for wiring together devices. It is built on top of node.js and makes use of the hundreds of thousands of existing Node.js packages that have already been developed. When we create an application in Node-Red, essentially, we compose our application from many types of node libraries and make minor changes to suit our needs. Node-Red is all about the flow of information between nodes. Some nodes produce data so they only output data. Some nodes transform and transmit data so they have both and input and output. Some nodes consume information to transmit out of the Node-red environment (eg. print to console, forward to TCP socket, etc.) so these nodes

[4] `https://nodered.org/`

only have an input point. We have chosen to have a local instance of the node-red application on our raspberry pi to establish direct connectivity with the sensors. We can use this simple drag and drop tool to get our sensors connected to our Kafka messaging queue. Once the application has been configured it no longer requires human intervention to connect to the IoT devices.

Gateway Implementation

In the first phase of our experiments we used a physical Raspberry Pi and a physical sensor which was connected to the Pi over Bluetooth as shown in figure 4.9. After developing this prototype we virtualized these two components to create EMU-IoT. On the virtualized Pi we installed a containerized instance of IBM's Node-Red tool. The gateway component that we built using Node Red will make the connection to the Zookeeper to obtain the address of the Kafka broker nodes so that the messages can be transmitted.

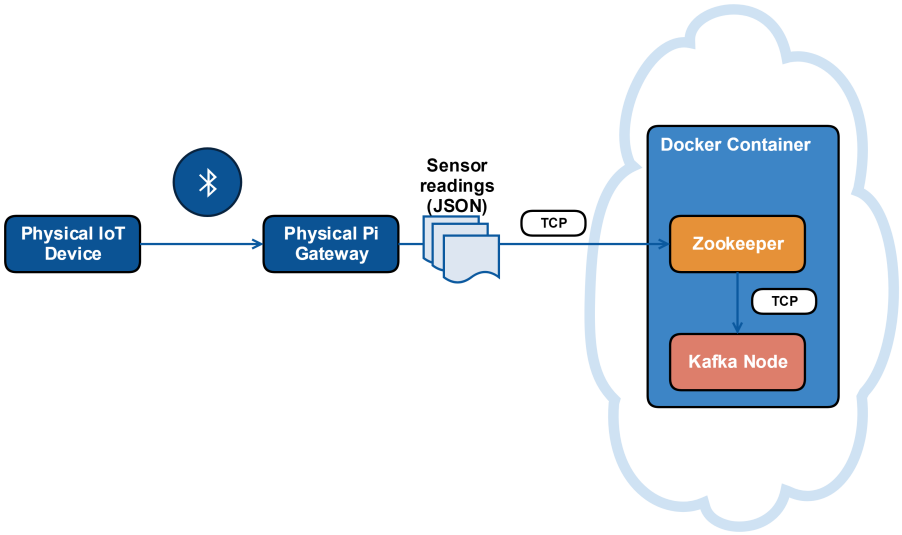

Figure 4.9: Physical IoT Device to Raspberry Pi Prototype

As shown figure in 4.10 to be able to get the data from the virtual IoT device we

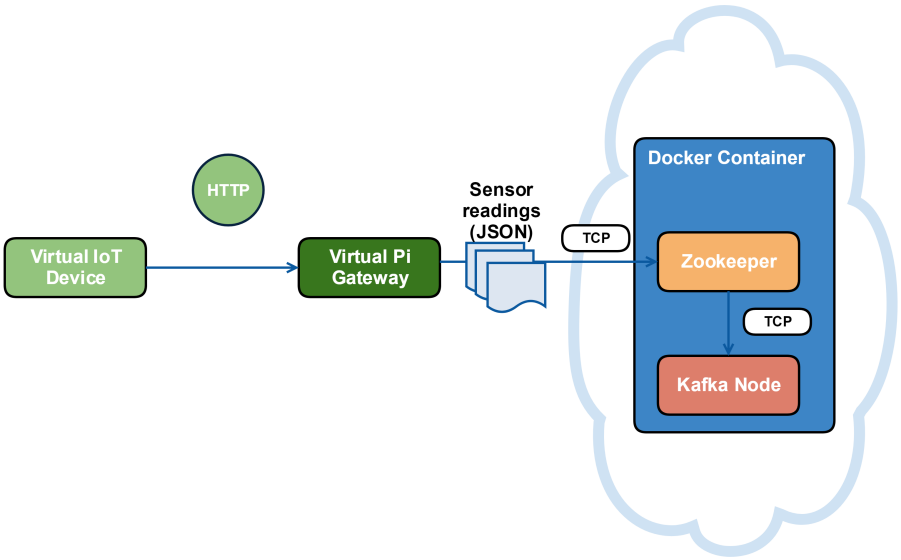

Figure 4.10: Virtual IoT Device to Virtual Raspberry Pi

replaced the Bluetooth receiving function with an HTTP web server that receives messages from the IoT device. When the data is received from the IoT device we pass the data onto Kafka.

Gateway Multiplexing

One of the import features of a virtual gateway is to take advantage of its connection to the IoT network. As mentioned in section 3.4.2, we designed the gateways to handle many incoming connections. For example a single virtualized gateway may accept messages from 10 individual IoT devices and a virtual machine can host up to 3 virtual gateways as show in figure 4.11 This requires us to only maintain 3 connections to the IoT network but we are able to serve 30 devices.

Gateway Configuration

The virtual gateways were designed to be created without changes at run time so the configuration is set at build time. As show in figure 4.12 we created a flow with several *nodes*. This will enable the gateway to forward the data to the big data IoT

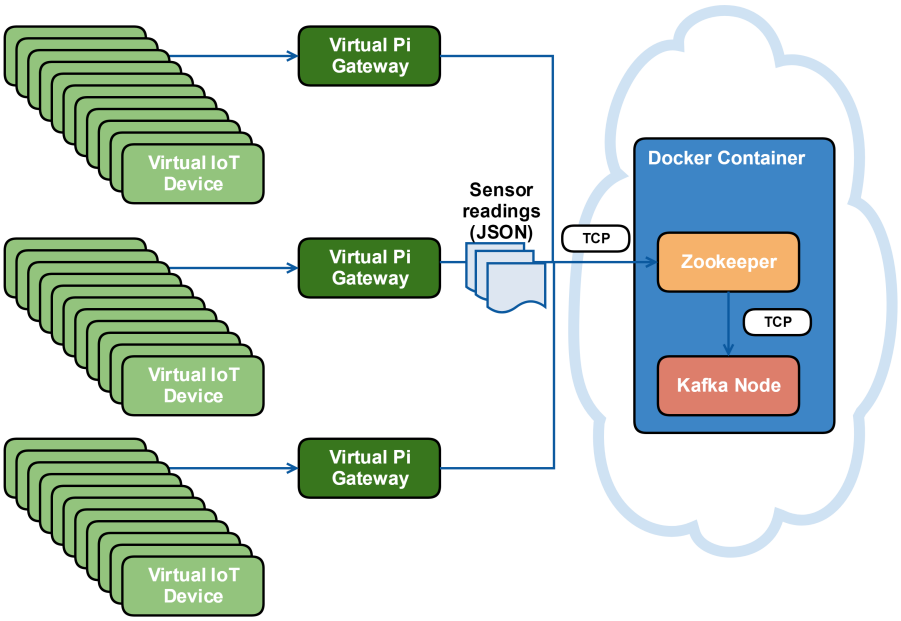

Figure 4.11: Virtual Raspberry Pi Multiplexed

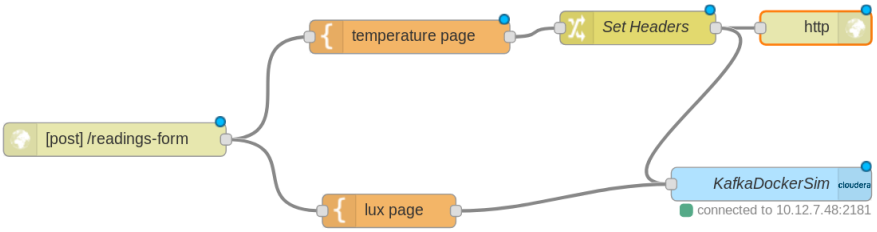

Figure 4.12: Node Red Flow Virtual Raspberry Pi

application we created. First we have the leftmost node which represents our HTTP web server that receives the incoming IoT messages. Then we have two nodes in orange that represent different pages that accept HTTP POST transactions from the IoT devices. In our example application, we have one page for the temperature readings and one for the lumen readings. The rightmost node (labeled KafkaDockerSim) is where the information flows out of the gateway to the messaging broker (Kafka). For this we have to add the Cloudera Node-Red library that contains the driver's to make the connection to the database. As previously mentioned, in Kafka the fun-

damental concepts are that we have producers and consumers. Therefore we need to use a producer node in our Node-Red application to send messages to our Kafka cluster.

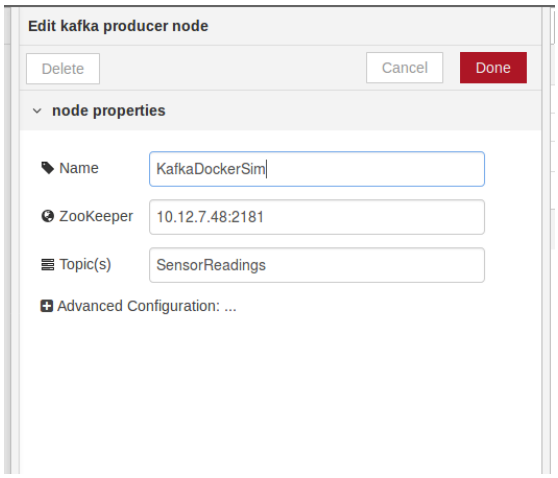

Figure 4.13: Node Red Kafka Configuration

As shown in figure 4.13 we provide the IP address of the Zookeeper than maintains a list of the Kafka nodes where the messages containing the sensor data can be sent to. The Zookeeper then returns a list of Kafka nodes. Once the we have the address of the Kafka node we can then begin to send the messages.

4.4 Key Modules

In this section we discuss the most important modules that were created to support the major functions of the EMU-IoT application. These key components are: the IoT Monitor that gets the performance data, the IoT Device Service that enables virtual IoT devices to be created and destroyed, the IoT Load Balancer that manages the decisions about how the virtual IoT device is placed on the network, IoT Experiment which executes configurable experiments, the Smart Testing Function that is used to find bot-

tlenecks and the example regression library that provides the Smart Testing function with its inputs. There are many more modules that provide vital services in the EMU-IoT application and they can be viewed at the Github repository. Please see the user guide in Appendix A for the link.

4.4.1 IoT Device Service

The IoT Device service provides the primary functionality for creating and destroying virtualized devices in EMU-IoT. The service abstracts many layers of the device creation process that eventually reach the container service provider libraries(Docker). This function will carry out the actual tasks of creating the containers. This allows us in the future to change container service providers if necessary. In our example application we created IoT Temperature and Light Sensor and IoT Camera. In each of these cases the IoTDeviceService class is extended as a subclass to represent the new type. This allows future users of EMU-IoT to create their own custom IoT device types while not requiring any further changes in other classes because they are forced to implement the same methods. The key methods of this service are:

```
addVirutalIoTDevice( IoTLoadBalancer, MonitorManager)
removeVirtualIoTDevice (IoTLoadBalancer, MonitorManager)
```

4.4.2 IoT Load Balancer

The IoT Load Balancer provides several functions for orchestrating the management of IoT Nodes and the assignment of IoT devices to those IoT Nodes. IoT Nodes are objects that represent physical/virtual machines which are the computing resources from the cloud service provider. IoT Nodes belong to IoT Network which is an object that represents the relationships between IoT Nodes. The orchestration functions of

IoT Load Balancer ensure that whenever a request to create a IoT device or Gateway is made, the correct IoT Node is provided based on the Load Balancing policy. This is an important feature for EMU-IoT because we want to be able to execute experiments that are geographically disbursed. For example we can set a policy to force all newly created IoT Devices to a particular IoT Node in a certain geographic location. IoT Load Balancer is also responsible for maintaining the overall health of the IoT network. IoT Load Balancer can perform cleaning and reset functions in case an experiment did not execute completely and remove lingering IoT devices on the network. Lingering IoT Devices are emulated IoT devices that are no longer needed. The key methods of this service are:

`iot_network_cleanup()`

`get_free_IoTProducerHost()`

`set_distribution_policy()`

4.4.3 IoT Monitor

The IoT Monitor provides all of the data gathering capability from across the entire IoT Network. The gathered data can be statistics of the resources in use on a particular node, such as CPU, Memory, Disk and Network bandwidth utilization. We can also gather information about the container services such as the number of containers and whether they have terminated or are still running. The resource utilization information that IoT Monitor gathers is the input data for the prediction engine. In our implementation we collect all of the metrics mentioned above, but we only use the CPU% utilization as the primary data source to make predictions. At the moment the data is stored in separate data files which allow us to map the files to specific instances of experiments so we know the genesis of the data. This became very useful when we

wanted to separate out the training data to be used by the prediction engine. Another major benefit of IoT Monitor, is that the application was designed to be multi-threaded, therefore we can monitor many nodes and services independently and specific monitors can be disabled if not needed. To implement this feature we created the Monitor Manager that allows for the coordination of the all monitoring functions. There are several key methods that allow the monitor to function, which are:

`iot_network_cleanup()`

`get_free_IoTProducerHost()`

`set_distribution_policy()`

4.4.4 IoT Experiment

The IoT Experiment Module is the main access point for the user to run coordinated experiments based on a set of configuration parameters that can be provided at build time. This module is extendible depending on what type of experiments you want to run. For example in our case we ran experiments based on exhaustive search. This means we are targeting a particular CPU% utilization point to find a bottleneck. In another case, we implemented a linear regression based experiment that learned from the exhaustive search data and then used the Smart Testing function to make predictions about unknown bottlenecks in an IoT application.

- *Smart Testing Function*

  The Smart Testing function that is implemented in IoT Experiment allows us to generate test cases based on the rules described in section 3.5.4. For example in our experiments we created a Heterogeneous experiment type combined with the Scaling type experiment where we defined 3 different test case types. Case 1: only temperature and light devices, Case 2: Only IoT Camera and Case 3:

Where we create 2 temperature and light devices for every 1 IoT camera sensor. A more detailed discussion of the experiments can be found in Chapter 5 We use the method `configureExperiment(experiment_type)` to set the type of experiment we want to run. This function will set the correct monitors based on what application we want to monitor to detect a bottleneck. For example with temperature and light devices we want to examine Kafka for bottlenecks so we turn off the Spark and Cassandra Monitors. The method `__executeWorkload()` then takes these configuration settings and triggers the `__generateTestCase()` until the target utilization is met.

- *Regression Library*

  As an add-on to the IoT Experiment module, we implemented a Regression library that is used to generate the test cases. This is based on the `Python statsmodels`[5] package. We set the target utilization which is a value that we want to find the bottleneck for and then the regression library loads the training data and computes the regression function. This function is then fed into the test case generation logic. The regression library implements an already tested and open sourced module provided to the Python community so we treat the computational methods as a black box.

[5] `https://www.statsmodels.org`

4.5 Chapter Summary

In this chapter we described how the major system components were developed. We discussed the process of taking a physical IoT device and transforming it into a virtualized IoT device that could be replicated at scale. We also described the key modules that are used in EMU-IoT and we discussed the example IoT application that was created to validate our design using the virtualized IoT Devices. There were significant challenges during the implementation phase of this project. For example, while developing the IoT camera it took several iterations and experiments to select an appropriate image resolution and frame rate to emulate a typical IoT camera. This because if the frame rate and resolution was very high, we would easily overwhelm the network by consuming all of the available bandwidth with just a dozen cameras. This would have made running experiments with large numbers of cameras difficult since we would only be able to create a small number of them.

Another one of the main challenges was developing IoT Monitor which collects the utilization data from IoT Nodes on the IoT network. Since we needed to run the various monitors in parallel, keeping the threads synchronized was challenging. To overcome this issue we implemented several thread management functions that monitor the threads and ensure they are terminated in a orderly fashion to prevent the overall application from crashing. This feature took a substantial amount of time to debug, but given that it was a critical function for collecting the data from our experiments, the testing and validation time was justified. With the camera configuration and multi threading issues solved, we were able to fulfill our research goals of having a system that is capable of detecting bottlenecks in IoT networks.

Chapter 5

Performance Evaluation

In this chapter we describe the methods we used to evaluate the bottleneck detection features of EMU-IoT. In section 5.1 we discuss the methodology for evaluating the results of our experiments by defining the measures that we use to compare the performance of each experimental case. Then in section 5.2 we discuss how the system was validated using control methods to make sure that the results of our experiments were reliable. In section 5.3 we discuss the variables that we will manipulate and the variables that we will observe in an attempt to show that EMU-IoT can be used for reliable bottle neck detection and prediction. We define a set of experiments and discuss the reasons for why we chose them. Then in section 5.4 we discuss the results of the experiments and attempt to determine if the bottlenecks were found and the degree of the accuracy using our prediction mechanism. This will allow us to see if certain scenarios are harder to predict with respect to target utilization compared to other scenarios.

5.1 Evaluation Methodology

In this section, we define and describe the choices we made regarding the types of metrics to collect, and how they are measured. For the metrics component, we decided to focus on CPU utilization as it is a widely accepted measure identified in the literature review for evaluating application performance.

The evaluation consisted of two phases. In the first phase, we collected the CPU utilization metrics from running the application where we search for a target utilization using the exhaustive search method. In the second phase of the experiment we use the collected metrics to derive a regression function that is used for prediction. We then execute a set of experiments to see if the predicted utilization it close to the actual utilization of the test case. In the context of our application, CPU usage is defined as the percentage of the available CPU cycles that is consumed by the container being monitored.

5.2 System Testing and Validation

In this section, we explain the various methods that are used to determine that the system is functioning correctly. First, we define what is a correctly functioning system and the various ways that it might not function correctly.

To be able to measure the performance of the system we built a monitor to collect the metrics from the various Docker containers that exist in our architecture. This monitor performs data collection on a predefined timed interval. Since we collect this information regularly this allows us to determine if there are any components that are not functioning in our system. If a component is not reporting any statistics, it has gone down. Therefore, we can describe a fully functioning system as operating as normally

if we can read data from the IoT devices, pass that data to the gateway and process that information stream to be ultimately stored in the database.

5.2.1 Data

The data produced by physical IoT devices are readings sensed from the environment. Since these values have a fixed type and size we can reliably reproduce them synthetically. Since we are interested in the volumes of messages, not the meaning of the contents, using the synthetic data will allow us to produce workloads that can be used to evaluate the performance of the architecture.

5.2.2 Data Integrity

In our IoT network, we can only process correctly typed and complete messages for a given sensor reading. As described in previous sections regarding message design, the integrity of the messages is checked at the processing point in Spark. This is done to ensure that valid information that meets our schema design is written to the database. Any messages that does not meet our specifications are ignored.

5.2.3 Message Delivery

When the system receives a message from the IoT device we must make sure that this information can successfully make it through the pipeline and into the database. To ensure delivery we can count the number of messages that are generated and compare that count to what is written to the database. These message counts can be checked at the Kafka and the Cassandra Database.

5.3 Experimental Plan

In this section we describe the types of experiments, the configuration and how the data is collected. As identified in section 2 we, use the CPU utilization as a measure for comparing experiments. The experiments are divided into two phases. In the first phase as shown in table 5.1 we execute the exhaustive search up to a chosen CPU% target. This means that we obtain the CPU readings for all cases until we reach the target. We identified 3 types of experiments that cover 3 typical IoT scenarios. First we have the Temperature and Light sensor experiment which represents IoT devices that generate very small amounts of data. Second we have IoT Camera which represents an IoT device that will generate large amounts of traffic. Lastly, we have a combination of the two types that models the scenario of a heterogeneous IoT network as this was one of our research goals to provide a platform with this capability. We repeat this same process ten times for three levels for each of the 3 experiment types. The exhaustive search experiments therefore yields 10 x 3 = 30 runs.

Device	Runs	Type	Application	Target CPU %
Temperature and Light	10	Exhaustive	Kafka	15.00%
IoT Camera	10	Exhaustive	Cassandra	30.00%
Temperature and Light(2) + IoT Camera(1)	10	Exhaustive	Cassandra	25.00%

Table 5.1: Experiment Plan Exhaustive Search

In phase two of the experiments we will attempt to make predictions based on the data that is gathered in the first phase of the experiments. As shown in table 5.2 we chose prediction targets that are beyond the data collected so that we can predict unknown values. For example, for IoT camera we collect data up to the 30% utilization point and then we try to predict how many IoT Cameras are required to increase the CPU utilization up to 32.50%, 35.00%, and 37.50%. We repeat this same process five

Device	Runs	Type	Application	Target CPU %
Temperature and Light	5	Regression	Kafka	17.50%
Temperature and Light	5	Regression	Kafka	20.00%
Temperature and Light	5	Regression	Kafka	22.50%
IoT Camera	5	Regression	Cassandra	32.50%
IoT Camera	5	Regression	Cassandra	35.00%
IoT Camera	5	Regression	Cassandra	37.50%
Temperature and Light(2) + IoT Camera(1)	5	Regression	Cassandra	27.50%
Temperature and Light(2) + IoT Camera(1)	5	Regression	Cassandra	30.00%
Temperature and Light(2) + IoT Camera(1)	5	Regression	Cassandra	32.50%

Table 5.2: Experiment Plan Linear Regression Search

times for three levels for each of the 3 experiment types. The regression experiments therefore yields 5 x 3 x 3 = 45 runs.

5.3.1 Testing Instruments

Hardware

The hardware specs are described in the implementation section. They are physical Raspberry Pi's that are similar to devices used in industry for interconnecting IoT Devices to networks. Since we are using a virtual test bed environment we instead have virtual machines that host our Docker containers which run all applications in our network.

Software

At the platform level, the software stack that is used across the network is Ubuntu 16.04 and the latest version of `Docker Community Edition 18.03.1-ce`. At the application level we have `Apache Kafka 1.0.0, Spark 2.1.0, and Cassandra 3.1` which are all deployed as containers on Docker. For the linear regression prediction function we used the `Python statsmodels` package.

5.3.2 Experiment Configuration

Experimental Setup

In this section we described all of the experiments that will be executed on the both the exhaustive search and the prediction experiments using linear regression. The variables that will be used across all experiments are presented in the following sections.

Dependent and Independent Variables

In the exhaustive search experiments we want to see if there is a fluctuation in the number of IoT devices for a given CPU target so therefore the device count becomes our dependent variable and the CPU becomes our independent variable. In the regression experiment we want to measure the changes in the CPU utilization based on the number and type of IoT devices. So the CPU utilization now becomes our dependent variable and the count and type of device becomes our independent variables. Please see table 5.2 for a complete list of experiment permutations that will be executed.

Experiment Constants

In the exhaustive search experiments we are interested only in observing the fluctuations in the IoT device count, so therefore we hold the CPU utilization target(the bottleneck) at a fixed amount. This was fixed across all 10 runs of the experiment for a given device type.

5.4 Results

In this section we present the results of all the experiments that were executed on EMU-IoT. These experiments were intended to show that based on our research goals we could first measure the performance of an IoT application from end to end, and second, be able to perform bottleneck detection that would trigger an adaptation for a given

IoT application. We discuss the experiment results for both the exhaustive search and linear regression.

5.4.1 IoT Temperature and Light Device

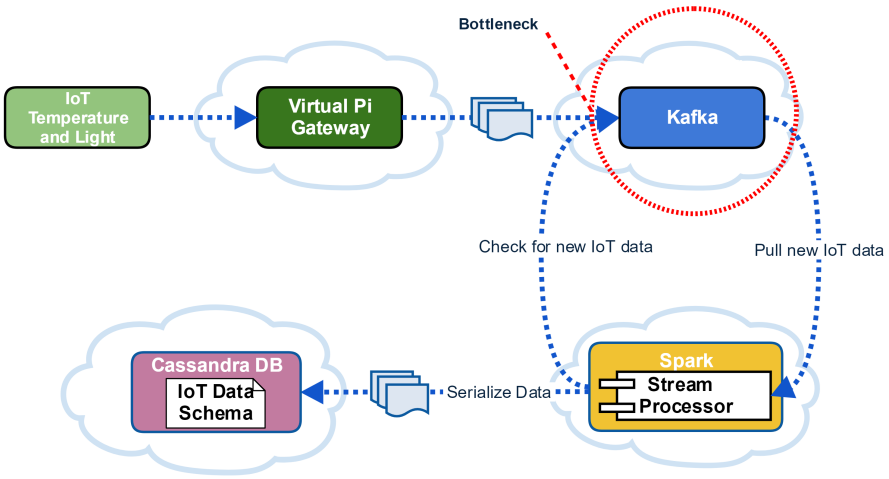

Figure 5.1: Bottleneck Point - IoT Temperature and Light

In the IoT Temperature and Light experiment our goal is to find the bottleneck when running a lightweight type of device. Once the bottleneck is found this would in theory trigger an adaptation in the infrastructure(scale up i.e. add more computing resources). For this type of IoT device the traffic is emitted and then sent to gateway where it is then forwarded to Kafka which is our aggregation point. From there the data is read by Spark and then stored in Cassandra. Prior to running these experiments we examined all three of these applications in the architecture and found that the CPU utilization rises the most with Kafka compared to the other applications. This is why we chose to examine only Kafka when experimenting with this IoT device type as shown in figure 5.1.

IoT Temperature and Light Discussion: ***Exhaustive Search***

After executing the experiment 10 times, the results of the IoT Temperature and Light

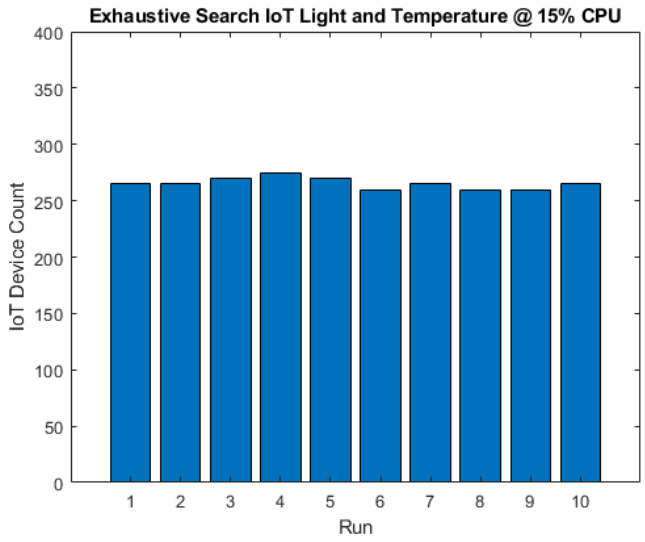

Figure 5.2: Exhaustive Search Results IoT Temperature and Light

device type experiment show that there is little variation in the number of IoT devices required to hit the 15% CPU utilization target. As shown in figure 5.2, for example the lowest number of IoT devices in the results is 260 and the highest number of devices required to hit the target is 275. The average number of IoT devices over the 10 runs is approximately 265. We can compute the standard deviation and it is within approximately 2% of the mean. This suggests that there is little variance in the data. The most likely cause of the minor variation in the results is due to cloud variability and other background processes running inside the container.

IoT Temperature and Light Discussion: *Linear Regression*

After executing the 10 runs from the exhaustive search we can derive the regression function (Equation 5.1) from the data in an effort to predict values beyond the 15% CPU utilization target. We plot the data from the exhaustive search experiments shown in figure 5.3 and we can see that there is strong positive linear relationship between the

IoT Temperature and Light device count and the CPU utilization.

$$(Number\_of\_Cores_i) \times (Utilization_i) = 0.2184 \times (Device\_count_i) + 3.1088 \quad (5.1)$$

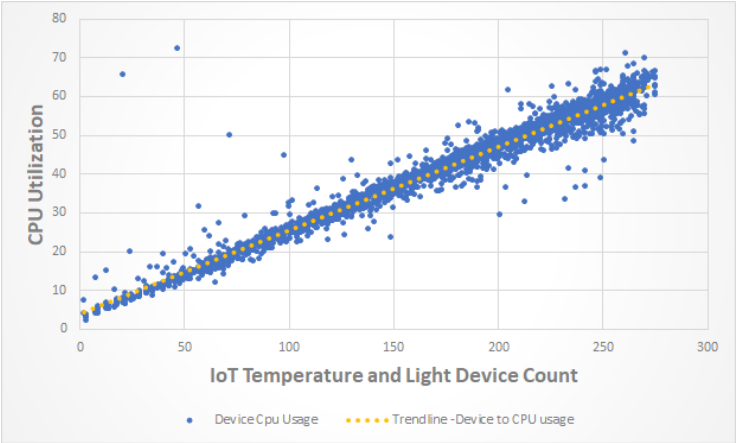

Figure 5.3: Regression IoT Temperature and Light

Using the regression function we execute three experiments by setting the $Utilization_i$ to 18%, 20%, and 23% and solve for $Device\_count_i$ which is the number of IoT Temperature and Light devices required to reach the chosen target utilization. We then ran each of the experiments 5 times. We also calculated the standard deviation to determine how close the results of our experiments were to the mean. The results are presented in table 5.3.

Predicted number of Devices	CPU Target	Actual Average CPU	Std Dev
306	**17.50%**	**17.30%**	0.1328
352	**20.00%**	**19.85%**	0.3996
397	**22.50%**	**21.70%**	0.3884

Table 5.3: Prediction Summary for IoT Light and Temperature

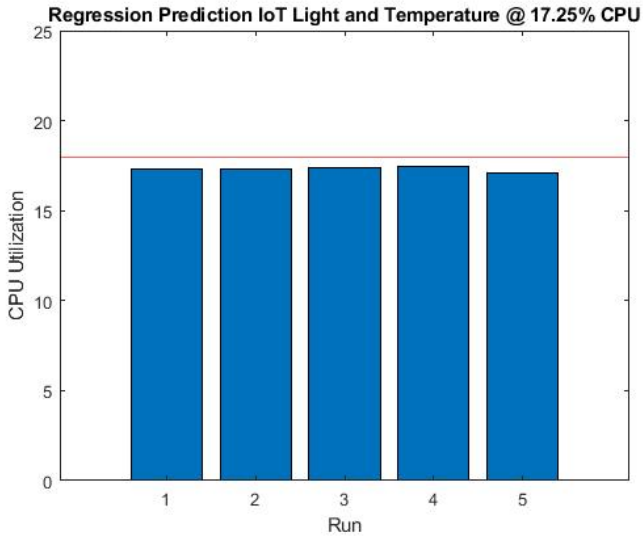

Figure 5.4: IoT Light and Temperature Prediction @ 17.50% CPU

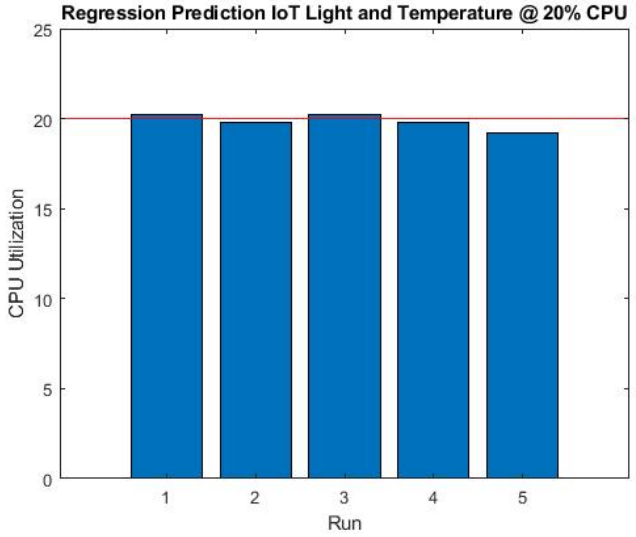

Figure 5.5: IoT Light and Temperature Prediction @ 20.00% CPU

In the first experiment we attempt to predict based on a target utilization of 17.50%, with 306 IoT Temperature and Light devices required. As shown in figure 5.4 the results were very consistent and we were able to reach an average target of 17.30%. In the second experiment we attempt to predict based on a target utilization of 20%, 352 IoT Temperature and Light devices are required. As shown in figure 5.5 the results

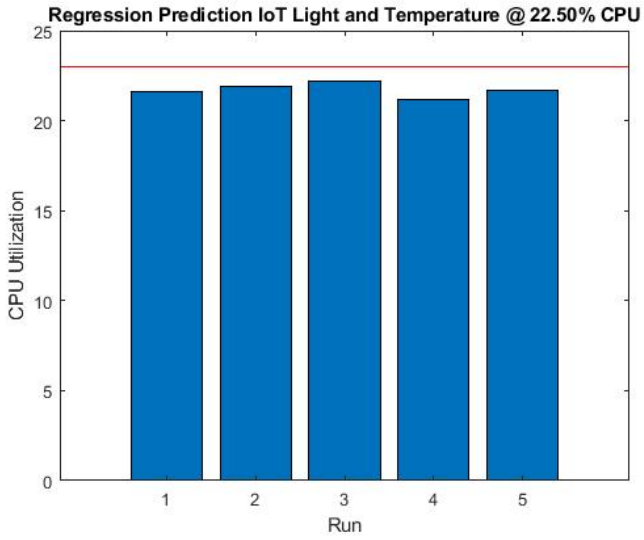

Figure 5.6: IoT Light and Temperature Prediction @ 22.50% CPU

were mostly consistent compared to the other two experiments and we were able to reach an average CPU utilization target of 19.85%. In the third experiment we attempt to predict based on a target utilization of 22.50%, where 397 IoT Temperature and Light devices are required. As shown in figure 5.6 the results were less consistent compared to the first experiments but more consistent compared to the second experiment and we were able to reach an average CPU utilization target of 21.70%. Generally the actual average CPU was quite close but always below the intended CPU target. This suggests maybe we need to always add an extra number of IoT Devices to reach our target.

5.4.2 IoT Camera

In the IoT Camera experiment our goal is to find the bottleneck when running only devices that create heavy traffic. For this type of IoT device the traffic is emitted and then sent directly to the storage application as shown in figure 5.7. There is no data being transmitted between the IoT Device, Kafka or Spark so the connector arrows in this figure are omitted to represent this concept. This in contrast to the IoT Temperature

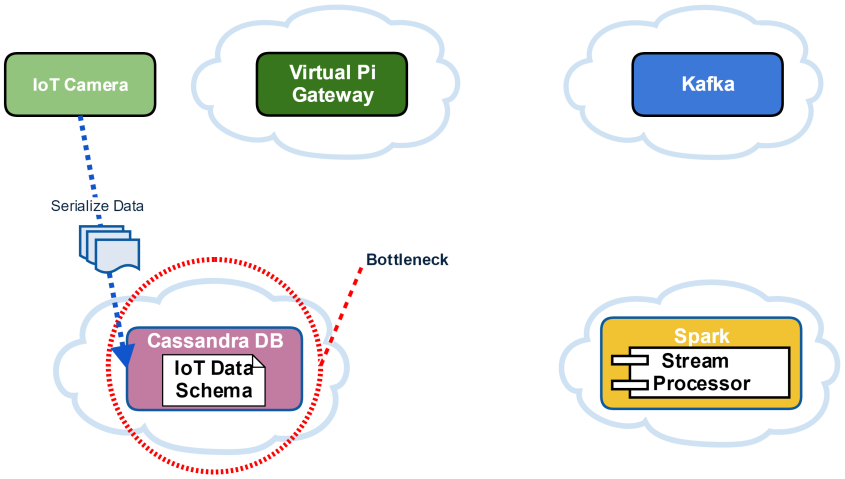

Figure 5.7: Bottleneck Point - IoT Camera

and Light device which sends its data first through the gateways. Prior to running these experiments we examined all three of these applications in the architecture and found that the processing of image data at the gateway would be unreasonable give the computing power at the gateways is rather low compared to a small sized cloud server. Another reason to send the data directly to the database is so that it can be read immediately by other services once the record is serialized to the table. This is why we chose to examine only Cassandra when experimenting with this IoT device type.

IoT Camera Discussion: *Exhaustive Search*

After executing the experiment 10 times, the results of the IoT Camera type experiment show that there is more variation compared to the IoT Temperature and Light experiment when looking at the number of IoT devices required to hit the 30% CPU utilization target. As shown in figure 5.8, for example the lowest number of IoT Camera devices in the results is 110 and the highest number of devices required to hit the target is 130. The average number of IoT Camera devices over the 10 runs is 122. We can compute the standard deviation and it is within approximately 6.46% of the

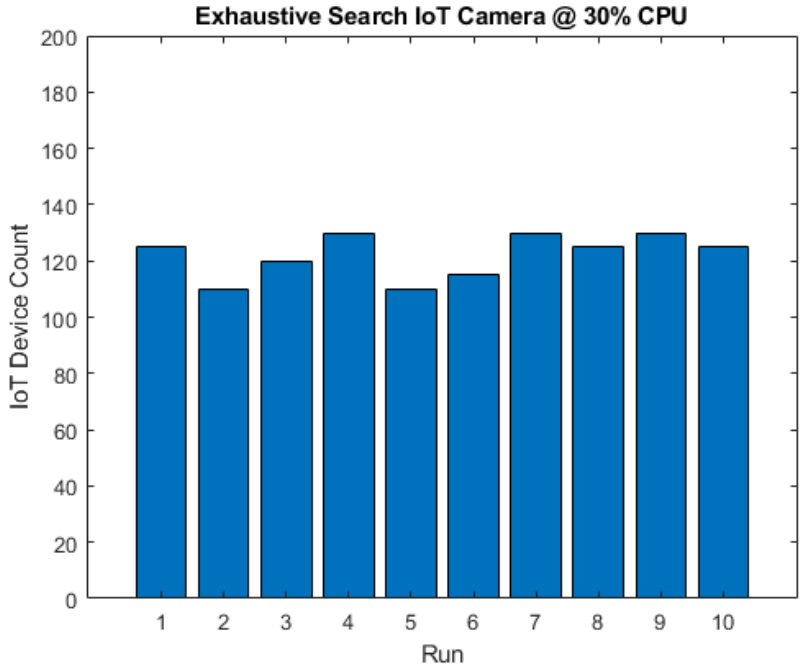

Figure 5.8: Exhaustive Search Results IoT Camera

mean. This suggests that there is moderate variance in the data. Likely the reason for the slightly higher variance is that Cassandra appears to have more background process that are CPU intensive as compared to for example Kafka.

IoT Camera Discussion: ***Linear Regression***

After executing the 10 runs from the exhaustive search we can derive the following regression function (Equation 5.2) from the data in an effort to predict values beyond the 30% CPU utilization target. We plot the data from the IoT Camera exhaustive search experiments shown in Figure 5.9 and we can see that there is a positive linear relationship between the IoT Camera device count and the CPU utilization. However there is a significant number of data points that are far from the trend line. As compared to the IoT Temperature and Light regression experiment, the line does not fit as well.

$$(Number\_of\_Cores_i) \times (Utilization_i) = 0.8416 \times (Device\_count_i) + 21.3387 \quad (5.2)$$

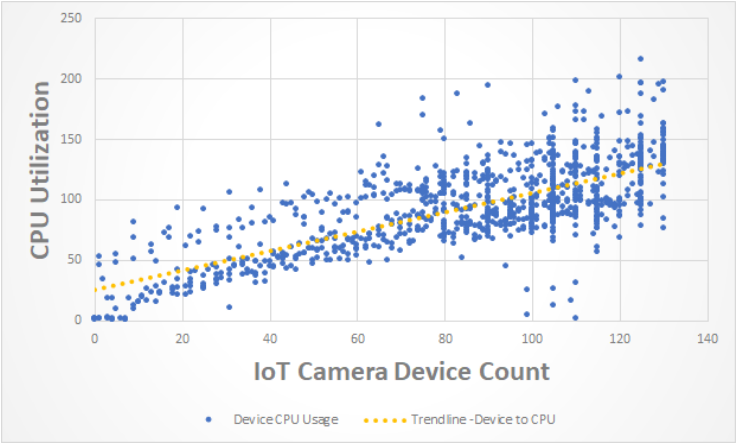

Figure 5.9: Regression IoT Camera

Using the regression function we execute three experiments by setting the $Utilization_i$ to 32.5%, 35%, and 37.50% and solve for $Device\_count_i$ which is the number of IoT Camera devices required to reach the chosen target utilization. We then ran each of the experiments 5 times. We also calculated the standard deviation to determine how close the results of our experiments were to the mean. The results are presented in table 5.4.

Predicted number of Devices	CPU Target	Actual Average CPU	Std Dev
129	**32.50%**	**33.56%**	1.2369
141	**35.00%**	**34.70%**	3.3461
152	**37.50%**	**36.44%**	3.0473

Table 5.4: Prediction Summary for IoT Camera

In the first experiment we attempt to predict based on a target utilization of 32.5%,

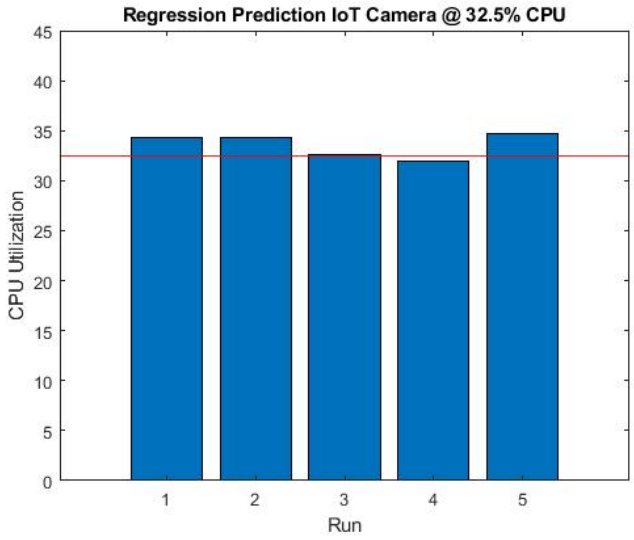

Figure 5.10: IoT Camera Prediction @ 32.5% CPU

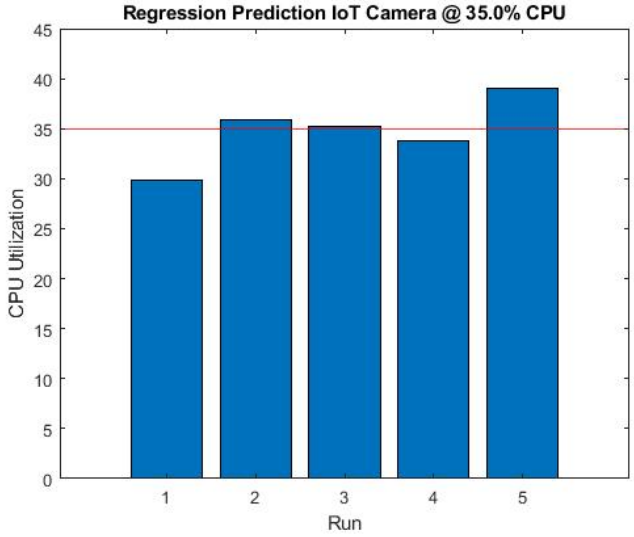

Figure 5.11: IoT Camera Prediction @ 35% CPU

with 129 IoT Camera devices required. As shown in figure 5.10 the results were fairly consistent with some minor variation in runs 3 and 4 and we were able to reach an average target of 33.56%. Since this average target is over the amount required this would be considered a service level violation. The adaption action to scale (i.e. add more computing resources) in this case would have been executed too late. In the

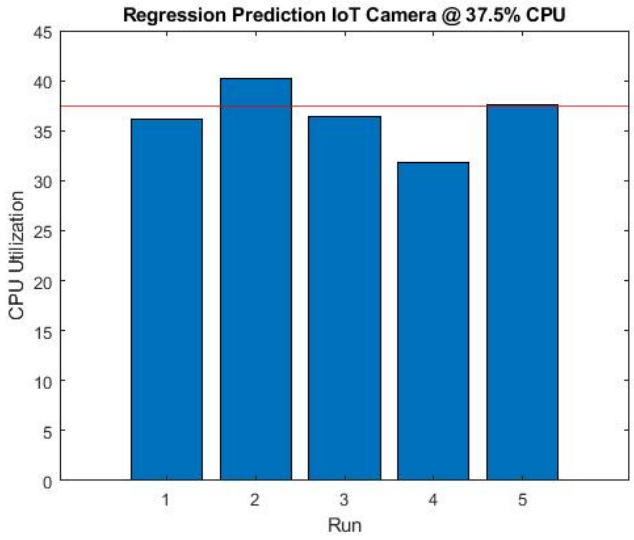

Figure 5.12: IoT Camera Prediction @ 37.5% CPU

second experiment we attempt to predict based on a target utilization of 35%, 141 IoT Camera devices are required. As shown in figure 5.11 the results were the least consistent compared to the first experiment and we were able to reach an average CPU utilization target of 34.70%. While this average amount was below and close to the required target(which is better than being over the target, there is a large amount of variation in run 1 compared to run 5. For example in run 1 we only reached a CPU utilization of 29.80% but in run 5 we overshot the target by hitting a utilization of 38.97%, this is a large discrepancy, resulting in a signification service level violation because the system would wait too long to adapt. In the third experiment we attempt to predict based on a target utilization of 37.50%, where 152 IoT Camera devices are required. As shown in figure 5.12 the results were quite close to the target.

5.4.3 IoT Camera + Temperature and Light Device

For the IoT Camera + Temperature and Light Device experiment our goal is to find the bottleneck when running a mix of devices that create both light traffic and heavy

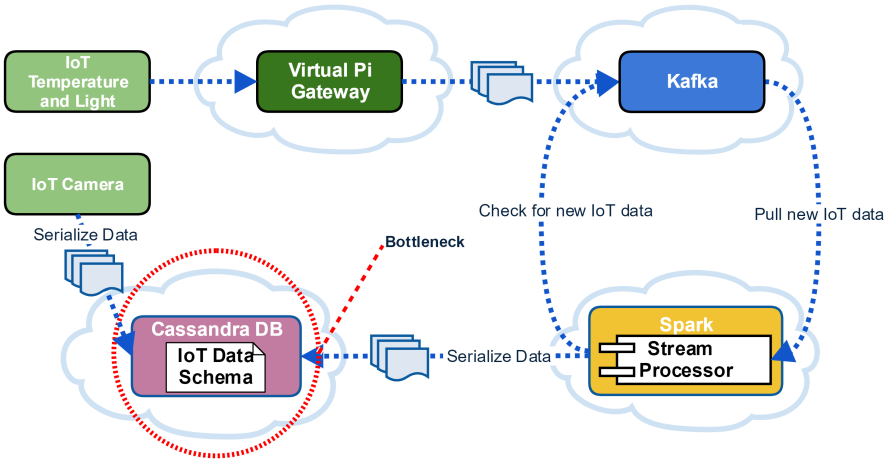

Figure 5.13: Bottleneck Point - IoT Camera + Temperature and Light Device

traffic. Of the three experiment types we have done, this is the most important one as it more closely models a real world scenarios since almost all IoT networks are heterogeneous. In this experiment setup we create IoT devices using a 2:1 ratio where we create two IoT Temperature and Light Devices for every one IoT camera. This experiment also allows us to model bottlenecks that use different data paths. As shown in figure 5.13 we have all data arriving at the Cassandra database but the data may take a different path depending on device type. Since all data arrives at the Cassandra database this is the only place where we can measure the affect of having both IoT Device types. Another property of this experiment that is not present in the previous experiments is the simultaneous modeling of a stream write and a batch write to the Cassandra database. As previously mentioned, the IoT Cameras are designed to push a constant stream of images where as the IoT Temperature and Light Devices have their data collected in batches.

IoT Camera + Temperature and Light Discussion: *Exhaustive Search*

Based on executing the experiment 10 times, the results of the IoT Camera + Temper-

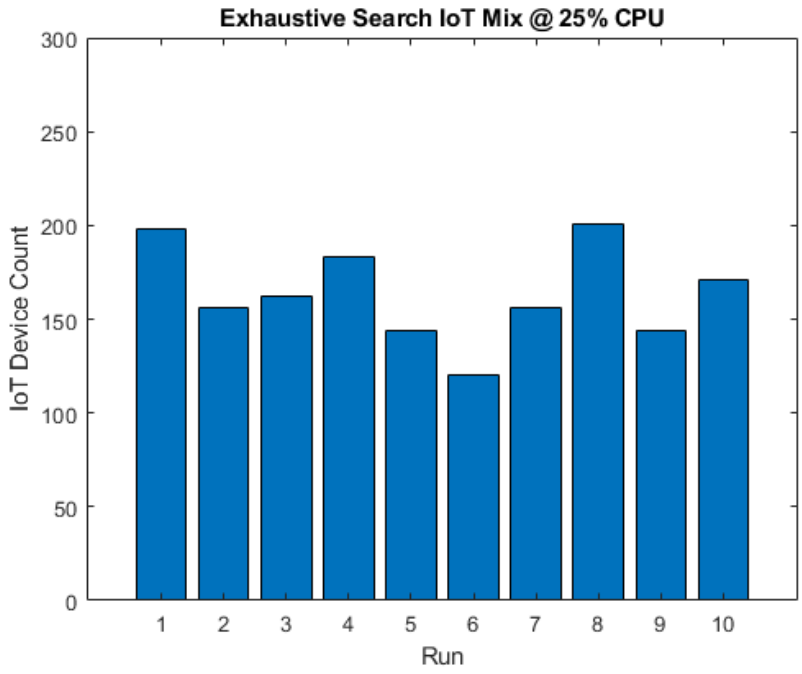

Figure 5.14: Exhaustive Search Results IoT Camera + Temperature and Light

ature and Light experiment show that there is a high amount of variation compared to the previous experiments when looking at the number of IoT devices required to hit the 25% CPU utilization target. As shown in figure 5.14, for example the lowest number of IoT Camera + IoT Temperature and Light devices in the results is 120 and the highest number of devices required to hit the target is 201. The average number of IoT Devices over the 10 runs is 163. We can compute the standard deviation and it is within approximately 15.52% of the mean. This suggests that there is high amount variance in the data. The reason for these large fluctuations it likely a combination of both background processes and the stream processor that is only present in this experiment. The batch pulls required bursts of CPU power to complete the task, making this type of scenario hard to model.

IoT Camera + Temperature and Light Discussion: ***Linear Regression***

After executing the 10 runs from the exhaustive search we can derive the following regression function (Equation 5.3) from the data in an effort to predict values beyond the 25% CPU utilization target. We plot the data from the IoT Camera + Temperature and Light exhaustive search experiments shown in 5.15 and we can see that there is a somewhat positive linear relationship between the IoT Device count and the CPU utilization. However there is even more of a significant number of data points that are far from the trend line as compared to the IoT Camera experiments.

$$(Number\_of\_Cores_i) \times (Utilization_i) = 0.4513 \times (Device\_count_i) + 8.2975 \quad (5.3)$$

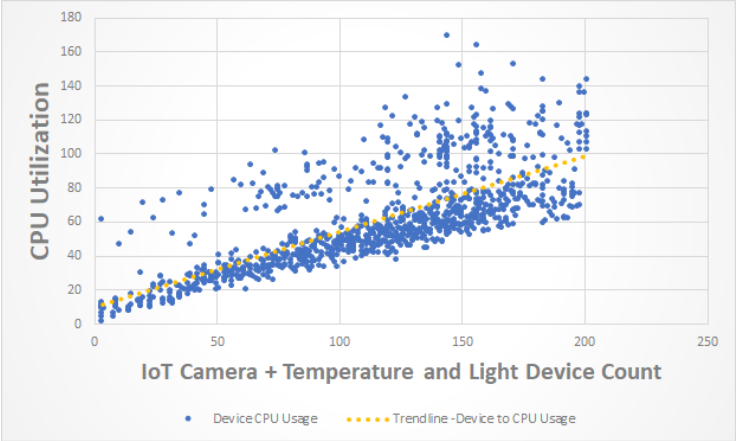

Figure 5.15: Regression IoT Camera + Temperature and Light

Using the regression function we execute three experiments by setting the $Utilization_i$ to 27.5%, 32.5%, and 35.0% and solve for $Device\_count_i$ which is the number of IoT Camera + Temperature and Light devices required to reach the chosen target utilization. We then ran each of the experiments 5 times. We also calculated the standard deviation to determine how close the results of our experiments were to the mean. The results are presented in table 5.5.

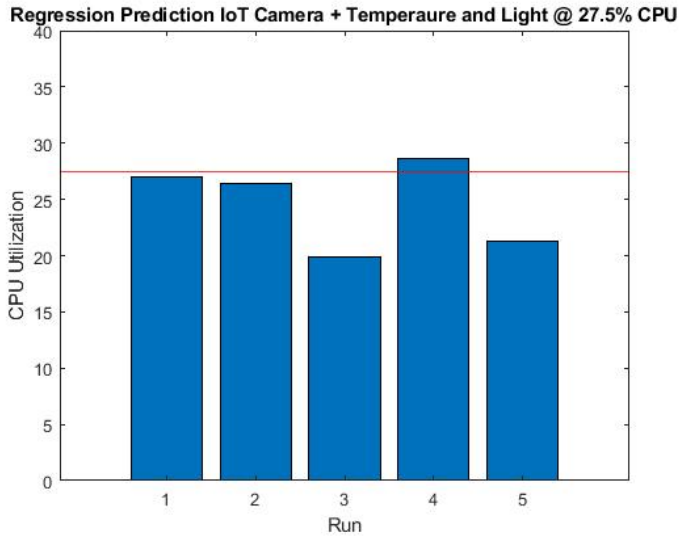

Figure 5.16: IoT Camera + Temperature and Light Prediction @ 27.5% CPU

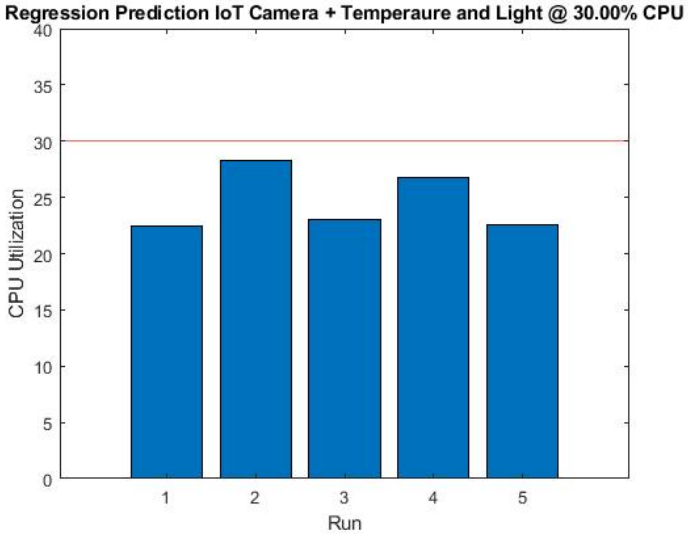

Figure 5.17: IoT Camera + Temperature and Light Prediction @ 30% CPU

In the first experiment we attempt to predict based on a target utilization of 27.5%, with 225 (150 IoT Temperature and Light + 75 IoT Camera) devices required. As shown in figure 5.16 the results from runs 3 and 5 show significant divergence from the mean. In this experiment we were able to reach an average target of 24.61% which is well below the intended target. Only in one case did we trigger a service level violation

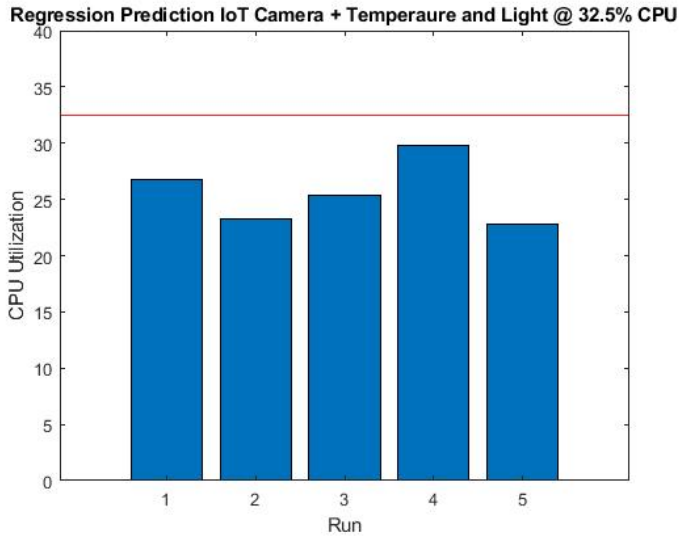

Figure 5.18: IoT Camera + Temperature and Light Prediction @ 32.5% CPU

Predicted number of Devices	CPU Target	Actual Average CPU	Std Dev
225	**27.50%**	**24.61%**	3.8055
247	**30.00%**	**24.63%**	2.6975
269	**32.50%**	**25.62%**	2.8492

Table 5.5: Prediction Summary for IoT Camera + Temperature and Light

in run 4. In the second experiment we attempt to predict based on a target utilization of 30%, with 247 (165 IoT Temperature and Light + 82 IoT Camera) devices are required. As shown in figure 5.17 the results were similarly inconsistent compared to the first experiment and we were able to reach an average CPU utilization target of 24.63%. This CPU usage amount is substantially below the intended target and only an increase of 0.02% above the CPU usage from the previous experiment. These results show that there were unknown container processes that affected the results in the first experiment. In the third experiment we observed similarly uneven results when we attempted to predict based on a target utilization of 32.50%, with 269 (179 IoT Temperature and Light + 90 IoT Camera) devices required. As shown in figure 5.18 the results were

all bellow the intended target which means we avoided a service violation, but would have triggered a scaling adaptation to early. Given the results of these 3 experiments, the combination of batch processing plus stream writing, and also using two different types of IoT devices, makes linear regression a less optimal method of prediction. Other possible prediction models that could be a better fit for this type of data would be stochastic hill climbing. This was the final experiment with EMU-IoT. In the next section we summarize our findings and attempt to draw some conclusions from the results of our experiments.

5.5 Chapter Summary

In this chapter we described the experiments that we ran on EMU-IoT to be able to show how we can arrive at a trigger point for an adaption on an IoT Network. We executed two types of experiments. First, using exhaustive search we gathered the training data. We then used a linear model to predict the CPU utilization for an unknown target. After executing several experiments we have observed that when the IoT network is homogeneous and the data values are small, the prediction models are more accurate. Once we introduced IoT device types such as the IoT Camera where we have larger data transmission volumes, the IoT applications began to exhibit more unpredictable CPU utilization spikes. When the IoT network is heterogeneous in nature, this makes our prediction method less precise as expected. In the next Chapter we conclude and discuss future work to extend EMU-IoT.

Chapter 6

Conclusion

IoT networks are increasingly becoming more complex. A better understanding of the behavior of applications and devices on these networks can lead to improved Quality of Service levels. By providing an end to end solution to evaluate IoT networks, we can more accurately measure the metrics that will allow us to determine if QoS goals are being met. By using software defined versions of IoT devices, this allows us to simulate large scale IoT networks without the cost and complexity of deploying a physical system. In this work, we implemented an adaptable network architecture, a platform to emulate IoT devices, a smart testing framework to detect bottlenecks and predict the demand for computing resources to maintain QoS targets. This solution can allow others to evaluate their particular IoT network configuration using their hardware and it provides a process to develop custom IoT devices.

For our first contribution, we demonstrated that we can build a customizable lab that can be used to model, monitor and evaluate heterogeneous IoT networks. The most challenging aspects of building EMU-IoT was making the architecture adaptable. One of our design goals was to make the lab infinitely scalable. We successfully im-

plemented a node management system to allows us to execute experiments of any size. This was important for the usefulness of extending this work in the future. Others can now use our lab to execute experiments on larger systems with more computing resources to emulate real world IoT networks.

For our second contribution, we developed a methodology for designing emulated IoT devices. We introduced the concept of virtualized IoT and provided a definition for such a device as having the properties of being connectible, configurable and deployable. This allowed us to use a structured approach to making virtualized representations of a physical IoT device. We successfully created a software defined temperature + light sensor, and also a network enabled video camera.

For our third contribution, we designed a smart testing framework that evaluates resource usage as a trigger for infrastructure adaptation (adding/removing computing resources). We implemented a prediction engine that can predict resource utilization using a linear regression model. The framework was also designed to be extendible so that others can build new prediction models to detect bottlenecks in a more precise manner.

Lastly for our final contribution, we evaluated the overall platform to show that we can model the performance of IoT networks and execute experiments using a smart testing framework to detect bottlenecks. We successfully executed experiments for both homogeneous and heterogeneous IoT networks and we were able to measure the results. Based on the results of three experiment types we were able to demonstrate the ability to reach the trigger point so that an adaption in the architecture can occur to maintain QoS goals.

6.1 Future Work

The variety of IoT devices in the future is expected to grow. Our platform was designed to be extendible. To model these future scenarios, we hope to support new IoT device types by using the generic framework we defined in EMU-IoT for creating new devices. An example of such devices could be wearables that generate movement data. This type of data can be emitted at high volumes but small size. Our architecture is currently capable of processing this type of data. Another device type we hope to support are IoT devices that are configurable at runtime. This would give us the ability to change the behavior of the device according to an operation plan. An example of this type of device would be a smart home thermostat.

Another one of our goals is to offer expanded support for more container platforms. Currently our architecture only supports Docker as a container provider. There are other container provider platforms such as LXC[1] and Rkt [2]. LXC is similar to Docker but has fewer features. Rkt is a security focused container technology provider.

We also hope to improve the intelligence of our smart testing framework. We experimented with linear regression which showed good results with homogeneous IoT network experiments using smaller data size. The experiments with heterogeneous IoT networks yielded less precise predictions. Other algorithms such as logistic regression may provide more accurate predictions of CPU utilization. Also, we hope to implement machine learning libraries that can continuously learn from past data to improve prediction accuracy.

At the moment the application is script based requiring text-based configuration parameters. Likely a graphic interface to model an IoT network, execute experiments

[1] `https://linuxcontainers.org/`
[2] `https://coreos.com/rkt/`

and visualized data would make the platform easier for others to use.

Bibliography

[1] A. Javed, K. Heljanko, A. Buda, and K. Främling, "Cefiot: A fault-tolerant iot architecture for edge and cloud," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, Feb 2018, pp. 813–818.

[2] H. F. Atlam, A. Alenezi, A. Alharthi, R. J. Walters, and G. B. Wills, "Integration of cloud computing with internet of things: Challenges and open issues," in *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, June 2017, pp. 670–675.

[3] A. Botta, W. de Donato, V. Persico, and A. Pescapé, "On the integration of cloud computing and internet of things," in *2014 International Conference on Future Internet of Things and Cloud*, Aug 2014, pp. 23–30.

[4] G. White, A. Palade, C. Cabrera, and S. Clarke, "Quantitative evaluation of qos prediction in iot," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, June 2017, pp. 61–66.

[5] A. Bounceur, O. Marc, M. Lounis, J. Soler, L. Clavier, P. Combeau, R. Vauzelle, L. Lagadec, R. Euler, M. Bezoui, and P. Manzoni, "Cupcarbon-lab: An iot emulator," in *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*, Jan 2018, pp. 1–2.

[6] S. N. Han, G. M. Lee, N. Crespi, K. Heo, N. V. Luong, M. Brut, and P. Gatellier, "Dpwsim: A simulation toolkit for iot applications using devices profile for web services," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, March 2014, pp. 544–547.

[7] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with cooja," in *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, Nov 2006, pp. 641–648.

[8] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, and R. Ranjan, "Iotsim," *J. Syst. Archit.*, vol. 72, no. C, pp. 93–107, Jan. 2017. [Online]. Available: https://doi.org/10.1016/j.sysarc.2016.06.008

[9] G. Kecskemeti, G. Casale, D. N. Jha, J. Lyon, and R. Ranjan, "Modelling and simulation challenges in internet of things," *IEEE Cloud Computing*, vol. 4, no. 1, pp. 62–69, Jan 2017.

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[11] A. , M. , A. Fox, A. , G. , R. , J. , A. D, R. Katz, R. H, A. Konwinski, A. , G. Lee, G. , P. , D. A, R. , A. , S. , and M. , "Above the clouds: A berkeley view of cloud computing," 01 2009.

[12] R. Buyya, S. K. Garg, and R. N. Calheiros, "Sla-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions," in *2011 International Conference on Cloud and Service Computing*, Dec 2011, pp. 1–10.

[13] J. Kang, H. Bannazadeh, and A. Leon-Garcia, "Savi testbed: Control and management of converged virtual ict resources," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, May 2013, pp. 664–667.

[14] D. S. Linthicum, "Connecting fog and cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 18–20, March 2017.

[15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, Fourthquarter 2015.

[16] L. D. Xu, W. He, and S. Li, "Internet of things in industries: A survey," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2233–2243, Nov 2014.

[17] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, Feb 2014.

[18] K. Rajaram and G. Susanth, "Emulation of iot gateway for connecting sensor nodes in heterogenous networks," in *2017 International Conference on Computer, Communication and Signal Processing (ICCCSP)*, Jan 2017, pp. 1–5.

[19] W. Cui, Y. Kim, and T. S. Rosing, "Cross-platform machine learning characterization for task allocation in iot ecosystems," in *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan 2017, pp. 1–7.

[20] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "Diane - dynamic iot application deployment," in *2015 IEEE International Conference on Mobile Services*, June 2015, pp. 298–305.

[21] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, "Exploring container virtualization in iot clouds," in *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, May 2016, pp. 1–6.

[22] Z. Nikdel, B. Gao, and S. W. Neville, "Dockersim: Full-stack simulation of container-based software-as-a-service (saas) cloud deployments and environments," in *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, Aug 2017, pp. 1–6.

[23] N. Naik, "Migrating from virtualization to dockerization in the cloud: Simulation and evaluation of distributed systems," in *2016 IEEE 10th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)*, Oct 2016, pp. 1–8.

[24] F. J. Riggins and S. F. Wamba, "Research directions on the adoption, usage, and impact of the internet of things through the use of big data analytics," in *2015 48th Hawaii International Conference on System Sciences*, Jan 2015, pp. 1531–1540.

[25] S. Rajeswari, K. Suthendran, and K. Rajakumar, "A smart agricultural model by integrating iot, mobile and cloud-based big data analytics," in *2017 International Conference on Intelligent Computing and Control (I2C2)*, June 2017, pp. 1–5.

[26] Y. Sun, H. Song, A. J. Jara, and R. Bie, "Internet of things and big data analytics for smart and connected communities," *IEEE Access*, vol. 4, pp. 766–773, 2016.

[27] H. Khojasteh and J. Mišić, "Task admission control policy in cloud server pools based on task arrival dynamics," *Wireless Communications and Mobile Computing*, vol. 16, no. 11, pp. 1363–1376. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcm.2689

[28] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155 – 162, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X11001129

[29] M. Chernyshev, Z. Baig, O. Bello, and S. Zeadally, "Internet of things (iot): Research, simulators, and testbeds," *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 1637–1647, June 2018.

[30] H. Gupta, A. VahidDastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509

[31] S. Sotiriadis, N. Bessis, E. Asimakopoulou, and N. Mustafee, "Towards simulating the internet of things," in *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, May 2014, pp. 444–448.

[32] G. D'Angelo, S. Ferretti, and V. Ghini, "Simulation of the internet of things," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 1–8.

Appendices

Appendix A

User Guide

Setting up Docker

*after you create your VM's you may need to manually add the server hostnames and ips to each of the /etc/hosts file in each VM to make them discoverable if you don't have a DNS server in your network.

On All VMs

1. Install Docker

2. Enable the Docker remote API on each vm, you need to modify the docker config file to allow this(http://www.littlebigextra.com/how-to-enable-remote-rest-api-on-docker-host/)

Cloning and building the docker images from Github

Gateway Setup:

This is the node-red image, that connects the gateway to the Kafka broker. You need to change the IP address to point to the correct Kafka broker in the following file before

you build. https://github.com/brianr82/node-red-docker/blob/master/sensor_flows.json

If you are using a Physical PI:(Figure) A.1

```
docker build --no-cache="true" -f "rpi/Dockerfile
https://github.com/brianr82/node-red-docker.git" -t
    "brianr82/multinodered:latest"
```

Figure A.1: Docker Physical PI

If you are using SAVI,AWS, Google Cloud, etc:(Figure) A.2

```
docker build --no-cache="true" -f "latest/Dockerfile
    https://github.com/brianr82/node-red-docker.git" -t
    "brianr82/multinodered:latest"
```

Figure A.2: Virtual Cloud PI

Virtual Sensor Setup:(Figure) A.3

On the VM where you want to run the virtual sensors, execute the following command:

```
docker build --no-cache="true" -f "Dockerfile
    https://github.com/brianr82/sensorsim.git" -t
    "brianr82/sensorsim:latest"
```

Figure A.3: Virtual Sensor Temperature and Light

Setting up the Docker Containers and the Overlay Network

On the VM where you want to run Kafka:(Figure) A.4

```
docker run -d -p "2181:2181" -p "9092:9092" --name "kafka-001" --env
    "ADVERTISED_HOST=142.150.208.238" --env "ADVERTISED_PORT=9092"
    "spotify/kafka"
```

Figure A.4: Kafka Docker Setup

On the VM where you want to run Spark + Cassandra

Create the overlay network:(Figure) A.5

```
docker network create --attachable --driver overlay "briannet"
```

Figure A.5: Docker Overlay Network

Start the Cassandra Cluster:(Figure) A.6

```
docker run -d --name "cassandra-001" -h "cassandra-001" -e
    "CASSANDRA_LISTEN_ADDRESS=cassandra-001" --net "briannet"
    "cassandra"
```

Figure A.6: Docker Cassandra

Before we can store tuples in the database we need to create the schema. First use Bash to get into the container to run the CQL statements to create the keyspace and schema.(Figure) A.7

```
docker run -it --rm --net "briannet" "cassandra" "cqlsh" "cassandra-001"
```

Figure A.7: Docker Bash Cassandra

Create the Keyspace:(Figure) A.8

```
CREATE KEYSPACE "sensordata" WITH replication = {'class':
    'SimpleStrategy', 'replication_factor': 1 };
```

Figure A.8: Cassandra Keyspace

Then create the schema:(Figure) A.9

```
create table "sensordata.all_data" ("sensor_id" "text", "sensorType"
    text, "timestamp" int, "daydate" date, "value" double, primary key
    (("sensor_id", "daydate","sensorType"), "timestamp")) with
    clustering order by ("timestamp" desc);
```

Figure A.9: Cassandra Schema

SPARK

Start the Spark Master: (Figure) A.10

```
docker run -d -t -p "8080:8080" --name "spark-master" --net "briannet"
    --hostname "spark-master" "corba/spark" "/start-master.sh"
```

Figure A.10: Spark Master

Start the Spark Workers: (Figure) A.11

```
docker run -d -t -p "8081" -p "4040" --name "spark-worker-001" --net
    "briannet" --hostname "spark-worker-001" -e
    SPARK_MASTER="spark-master" corba/spark
docker run -d -t -p "8081" -p "4040" --name "spark-worker-002" --net
    "briannet" --hostname "spark-worker-002" -e
    SPARK_MASTER="spark-master" corba/spark
```

Figure A.11: Spark Workers

Building the Spark Job

From your local machine:

1. Clone the repo into an IntelliJ Java project (https://github.com/brianr82/SensorExperiment.git)

2. Change the ip address inside the script to the IP of the Kafka broker

3. Assemble the JAR:

    a. In intelliJ open the terminal tab and type ‘sbt assembly’

    b. Wait for it to finish creating the JAR

Starting the Spark Job

Copy the jar file to the Spark+Cassandra Docker VM: (Figure) A.12

```
scp -i "/home/brianr/key/brian-hb.key" "/home/brianr/IdeaProjects
/SensorExperiment/target/scala-2.11/SensorExperiment-assembly-1.0.jar"
    "ubuntu@10.12.7.42:/home/ubuntu"
```

Figure A.12: Spark Job

Copy the jar from the Spark+Cassandra Docker VM into the SPARK master container: (Figure) A.13

```
docker cp "SensorExperiment-assembly-1.0.jar
    spark-master:/SensorExperiment-assembly-1.0.jar"
```

Figure A.13: Copy Streaming JAR to Container

From the virtual machine running the Spark master container, start the Spark Job to pull the data from Kafka: (Figure) A.14

```
docker exec -d -it exec "spark-master /usr/local/spark/bin/spark-submit
    \--class "KafkaSensorStream" --master spark://spark-master:7077[4]
    SensorExperiment-assembly-1.0.jar"
```

Figure A.14: Start Job

End of infrastructure setup

The Spark job should now be running on a 5 second interval pulling the data from Kafka and saving it to Cassandra. Next Step, clone the experiment execution tool, and run some experiments. https://github.com/brianr82/Sensor-Manager.git