# Container-based IoT Sensor Node on Raspberry Pi and the Kubernetes Cluster Framework

**Asad Javed**

**School of Science**

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 25.7.2016

**Thesis supervisors:**

Assoc. Prof. Keijo Heljanko

**Thesis advisor:**

D.Sc. (Tech.) Vesa Hirvisalo

**Aalto University**
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Asad Javed

Title: Container-based IoT Sensor Node on Raspberry Pi and the Kubernetes Cluster Framework

Date: 25.7.2016 Language: English Number of pages: 8+68

Department of Computer Science
Degree Programme in Computer Science and Engineering
Master's Programme in ICT Innovation

Supervisor: Assoc. Prof. Keijo Heljanko

Advisor: D.Sc. (Tech.) Vesa Hirvisalo

In recent years, Internet of Things is envisioned to become a promising paradigm in the future Internet. It allows physical objects or things to interact with each other and with users, thus providing machine-to-machine communication that is long been promised. As this paradigm continues to grow, it is changing the nature of the devices that are being attached. This opens a path of embedded systems to be the natural means of communication, control, and development. The ability of these systems to connect and share useful information via Internet is becoming ubiquitous.

In many cases, enormous amount of data is generated from embedded devices that need to be processed in an efficient way along with the required computation power. Container-based virtualization has come into existence to accomplish those needs in order to produce an improved system, which has the capability to adapt operational features in terms of security, availability, and isolation.

This thesis project is aimed to design and develop a Kubernetes managed container-based embedded IoT sensor node through the use of a cluster. In this project, the cluster was formed by connecting five Raspberry Pi boards to a network switch. This sensor node operates by collecting data from camera and temperature sensors, processing it in a containerized environment, and then sending it to the cloud platform using the Apache Kafka framework. The main motivation of adopting state-of-the-art technologies is to achieve fault-tolerant behavior and processing location flexibility using edge computing. In the end, the overall cluster is evaluated on the basis of architecture, performance, fault-tolerance, and high availability that depicts the feasibility, scalability, and robustness of this sensor node. The experimental results also conclude that the cluster is fault tolerant and has a flexibility over data processing in terms of cloud and edge computing.

Keywords: Internet of Things, Embedded systems, Linux containers, Cluster computing, Raspberry Pi, Cloud computing, Kubernetes, Docker, Distributed systems, Apache Kafka

# Acknowledgments

I extend the deepest of gratitude towards my supervisor, Assoc. Prof. Keijo Heljanko, for supporting me during the whole process. His valuable guidance and excellent feedback have kept me motivated throughout this project. I would like to thank my colleague, Hussnain Ahmed, for selflessly helping me to finalize this project. In addition, I would also like to thank my instructor, Vesa Hirvisalo, for his comments and valuable insight into the topic. Last but not the least, I would like to thank my parents who supports me in every step of my life. Without their love, support, and prayers, it was not possible for me to excel in my master studies and complete this thesis project.

Espoo, July 25, 2016

Asad Javed

# List of Figures


1.1 Cloud Computing Architecture . . . . . . . . . . . . . . . . . . . . . . 6

2.1 Hypervisor-based virtualization architecture . . . . . . . . . . . . . . . 10
2.2 Container-based virtualization architecture . . . . . . . . . . . . . . . . 11
2.3 Architectural Diagram of Docker . . . . . . . . . . . . . . . . . . . . . 16
2.4 High-level architecture of Borg . . . . . . . . . . . . . . . . . . . . . . 19
2.5 High-level architecture of Kubernetes . . . . . . . . . . . . . . . . . . . 21
2.6 Architectural diagram of Swarm . . . . . . . . . . . . . . . . . . . . . . 25
2.7 Clustered architecture of Kafka . . . . . . . . . . . . . . . . . . . . . . 27
2.8 Anatomy of a Topic . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

3.1 IoT sensor node overview . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.2 Execution order of Kubernetes processes . . . . . . . . . . . . . . . . . 34
3.3 Working scenario of Flannel . . . . . . . . . . . . . . . . . . . . . . . . 36
3.4 Setting of proxy flag on master nodes; (a) In case of a primary master node, (b) When the primary master fails, (c) When both primary and secondary masters fail, and (d) When the primary master comes back online . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.5 Proxy flag settings on worker nodes; (a) In case of a primary master node, (b) When the primary master fails . . . . . . . . . . . . . . . . . 39

4.1 Graphical representation of Zookeeper cluster . . . . . . . . . . . . . . 44
4.2 Graphical representation of Kafka cluster . . . . . . . . . . . . . . . . . 46
4.3 Data pipelining based on Kafka topics . . . . . . . . . . . . . . . . . . 49
4.4 Temperature data pipelining . . . . . . . . . . . . . . . . . . . . . . . . 51

# List of Tables


2.1 Comparison between Virtual Machines and Containers . . . . . . . . 14

3.1 Required Hardware components . . . . . . . . . . . . . . . . . . . . . . 32
3.2 Etcd parameters for running in the cluster . . . . . . . . . . . . . . . . 35
3.3 Kubelet parameters for running apiserver, controller manager, and scheduler on three master nodes . . . . . . . . . . . . . . . . . . . . . 37

4.1 Parameters for configuring camera module . . . . . . . . . . . . . . . . 42
4.2 Parameters for defining motion detection . . . . . . . . . . . . . . . . 43
4.3 Parameters for configuring Zookeeper cluster . . . . . . . . . . . . . . 45
4.4 Parameters for Kafka broker . . . . . . . . . . . . . . . . . . . . . . . 47
4.5 Important fields of YAML file . . . . . . . . . . . . . . . . . . . . . . . 48

5.1 Processing time of the images that are sent to the local topic . . . . . 54
5.2 Processing time of the images that are sent to the remote topic . . . . 55
5.3 Fault-tolerance behavior based on etcd data store . . . . . . . . . . . 56

# Contents

**Abstract** ii

**Acknowledgments** iii

**Contents** vi

**Abbreviations and Acronyms** viii

**1 Introduction** 1
- 1.1 Motivation . . . 2
- 1.2 Research Goals and Terminologies . . . 2
  - 1.2.1 Internet of Things . . . 2
  - 1.2.2 Cluster Computing . . . 4
  - 1.2.3 Cloud Computing . . . 5
  - 1.2.4 Embedded Systems . . . 6
- 1.3 Thesis Structure . . . 7

**2 Background** 8
- 2.1 Linux Virtualization . . . 8
  - 2.1.1 Hypervisor-based Virtualization . . . 9
  - 2.1.2 Container-based Virtualization . . . 10
  - 2.1.3 Docker Containers . . . 13
- 2.2 Container Management Systems . . . 17
  - 2.2.1 Google Kubernetes . . . 18
  - 2.2.2 Docker Swarm . . . 24
- 2.3 Data Computing Platforms . . . 26
  - 2.3.1 Apache Kafka . . . 26
  - 2.3.2 Apache Zookeeper . . . 29

**3 Design and Implementation** 30
- 3.1 System Architecture . . . 30
- 3.2 Raspberry Pi Cluster . . . 32
  - 3.2.1 Basic setup . . . 32
  - 3.2.2 Kubernetes configuration . . . 33

**4 Data Pipelining** 41
4.1 IoT Sensor Node . . . . . . 41
4.1.1 Configuring motion detection . . . . . . 42
4.1.2 Setting up Kafka cluster . . . . . . 44
4.1.3 Containers deployment . . . . . . 47
4.2 Data Processing and Simulation . . . . . . 48
4.3 Containers for temperature data . . . . . . 51
4.4 Design Decisions . . . . . . 52

**5 Evaluation** 53
5.1 Architecture . . . . . . 53
5.2 Performance . . . . . . 53
5.3 Fault tolerance . . . . . . 55
5.4 Scalability . . . . . . 57

**6 Summary and Conclusion** 58
6.1 Limitations . . . . . . 60
6.2 Future Work . . . . . . 60

**References** 61

**A Configuring Kubernetes cluster** 65
A.1 Service configuration files for Kubernetes cluster . . . . . . 65
A.2 Shell scripts for monitoring master nodes . . . . . . 65

**B Configuring and running Kafka cluster** 66
B.1 Configuration files for creating pods . . . . . . 66
B.2 Code for Kafka producers and consumers . . . . . . 66
B.3 Script inside Zookeeper and Kakfa container for cluster configuration 67

# Abbreviations and Acronyms

| | |
|---|---|
| IoT | Internet of Things |
| RPi | Raspberry Pi |
| RFID | Radio-Frequency Identification |
| NFC | Near Field Communication |
| HPC | High Performance Clusters |
| IIoT | Industrial Internet of Things |
| VM | Virtual Machine |
| VMM | Virtual Machine Monitor |
| EC2 | Elastic Compute Cloud |
| IaaS | Infrastructure-as-a-Service |
| PaaS | Platform-as-a-Service |
| SaaS | Software-as-a-Service |
| DST | Digital Set-Top |
| GB | Gigabyte |
| RAM | Random Access Memory |
| ARM | Advanced RISC Machine |
| RISC | Reduced Instruction Set Computer |
| USB | Universal Serial Bus |
| HDMI | High Definition Multimedia Interface |
| API | Application Programming Interface |
| IPC | Inter-Process Communication |
| rc | Replication Controller |
| DHCP | Dynamic Host Configuration Protocol |
| TCP | Transmission Control Protocol |
| URL | Uniform Resource Locator |
| DNS | Domain Name System |
| YAML | Yet Another Markup Language |
| JSON | JavaScript Object Notation |
| CSI | Camera Serial Interface |
| BIOS | Basic Input/Output System |
| GPU | Graphics Processing Unit |
| JDK | Java Development Kit |

# Chapter 1

# Introduction

As the Internet is expanded around the globe, countless number of networked automated devices are being developed that are gaining dominance in today's modern world. The ability of these devices to connect, communicate with, share information, and remotely manage the resources via Internet is becoming pervasive. In many cases, the devices are built specifically for the purpose to communicate with the cloud platform and handle huge volumes of data. This increases the overall flexibility by providing robust system but can also lead to a challenging scenarios in terms of efficiency and computation power. Since the data produced by devices are growing rapidly, a single server is not enough to perform the required system computations. There is a need to produce an improved system which is hardware independent and has the capability to provide performance efficiency in terms of availability, security, and isolation.

Virtualization technologies has come into existence to accomplish those needs. Server virtualization has become an expected course of action since last decade, ranging from traditional services to the ever-expanding cloud services [27][26]. The ubiquitous nature of virtualization also depicts an increasing amount of resource efficient applications especially in the world of embedded systems. It offers many benefits in several areas of embedded computing, including enhanced CPU utilization, increased security, better migration, and cost reduction. Although embedded devices perform considerably well in real scenarios, it is still necessary to achieve better performance in terms of low power and efficiency. For this purpose, ARM-based chips are utilizing virtualization to provide power efficient and optimized systems, instead of their x86 counterparts.

In this thesis, a container-based virtualization technique has been adopted to implement an embedded IoT sensor node. The main idea is to develop a clustered system of five Raspberry Pi (RPi) embedded boards by utilizing Docker containers and the Kubernetes cluster framework. The main reason of selecting RPi is to produce an efficient system, since it has an ARM-based processor. The sensor node operates by collecting temperature data and motion detected pictures from the camera sensors attached to each RPi board. The pictures in the form of data is then replicated across a cluster and sent to the cloud platform using a well-known messaging system called Apache Kafka. The cloud platform also contains Kafka

cluster which receives data from the sensor node and process it in an efficient way. In addition, the cloud server is also integrated with Hadoop Distributed File System (HDFS) in order to store and process large volumes of data. Following sections and chapters describes the overall concept, technologies, and implementation procedure related to this project.

## 1.1 Motivation

This section laid emphasis on the motivation behind the implementation of IoT sensor node using state-of-the-art technologies and frameworks. The first motivation is related to the fault-tolerance property of the system which can preserve their state in case of a failure. This property ensures that the system continues to execute its intended operation rather than failing completely. It will be judged on the basis of different parameters on the local cluster and the cloud platform as well. Another main motivation is to adapt the features of edge computing [24] alongside cloud computing. Through this way, the services and applications can be handle at the edge of the network, thus minimizing physical distance delay and consuming less amount of bandwidth. In addition, it can also provide the flexibility in choosing processing location.

## 1.2 Research Goals and Terminologies

This section describes some important terminologies that are used in the implementation of IoT sensor node. It starts by explaining the concept of Internet of Things in general along with different perspectives. The section further proceeds by describing cluster computing, cloud computing, and embedded systems as a specific technologies in IoT.

### 1.2.1 Internet of Things

The Internet of Things (IoT) is the emerging network infrastructure that is gaining considerable attention in modern telecommunications [1]. It is a novel paradigm which allows physical objects or things to interact with each other and with users, thus providing communication and exchanging data over the Internet. The rapid growth of wireless technologies and the Internet have made IoT an integral part of the scientific research. The idea behind this concept is to make communication even more pervasive and agile without requiring any human-to-human or human-to-computer interaction. In this phenomenon, the objects of everyday life such as devices, home appliances, buildings, vehicles, sensors, displays, and other items equipped with micro-controllers are integrated with small electronic chips. These chips also contain distinctive identifiers and have the capability to send/receive specialized signals. Moreover, by enabling network access, the objects can be used for the development of society, and hence find applications in many areas such as traffic management, home automation, health-care, industrial automation, and many others. Due to the

vast majority of applications, IoT can also be called as Internet of everything [39] [20].

In a broader sense, the goal of IoT is to connect objects anywhere, anytime with any network in order to provide ease of access and high availability. The term Internet of Things is theoretically composed of two words. The first word focuses more on the network oriented domain, while the latter one gives the idea of any generic object that can be integrated with one or more networks to make network of things. In fact, when put together, the term can be seen as a world-wide network of connected devices that are using communication protocols with proper addressing to exchange data [1]. A thing, in IoT, can be a robotic vehicle which measures outside temperature using a sensor attached to it, a person that has a heart monitor for measuring heart conditions, an animal with a camera attached to it for surveillance, or any other object that can be made accessible to the Internet and transfer data over the network. Hence, by connecting all the things together and providing network capabilities, the overall term can be considered as being smart.

According to the survey presented in [1], the IoT paradigm has comprised of three visions or perspectives: Things-oriented vision, Internet-oriented vision, and Semantic-oriented vision.

**Things-oriented vision:** This category comprises of physical objects ranging from specialized electronic devices to simple non-specialized objects that can be made useful by enabling network access, as mentioned in the above paragraph. The main items include Radio-Frequency Identification (RFID) tags, Near Field Communication (NFC), wireless sensors, actuators, and other smart items.

**Internet-oriented vision:** This domain comprises of different sets containing one or more objects that are connected to each other and exchanging data over the network. This is formulated by accessing the communication protocol stack which is utilizing IP stack to make IoT a reality.

**Semantic-oriented vision:** The idea behind this perspective is to organize the information semantically that is generated by IoT. Since large number of devices with unique identifiers are connected, the issues related to storing, searching, representing, and organizing information become challenging. Therefore, semantic oriented technologies based on data reasoning are used to overcome this problem, and hence the need for separate domain is essential.

Apart from IoT, there is another term called Industrial Internet of Things (IIoT)[1] which includes manufacturing industries and also considered as the industrial subset of IoT [38]. It is a major trend with some implications for the global economy. It spans multiple industries including oil and gas, manufacturing, mining, agriculture, and utilities which represents 62 percent of Gross Domestic Product (GDP). It also encompasses other companies that are based on durable physical goods and

[1] `https://www.accenture.com/us-en/labs-insight-industrial-internet-of-things.aspx`

services for improving business with the outside economy. These companies include organizations which provide services such as hospitals, ports, warehouses, transportation, health-care, and many others. Due to the rapid growth in manufacturing, IIoT would eventually lead to the fourth industrial revolution, called as Industry 4.0[2], by generating a business value. Industry 4.0 has been defined as “a collective term for technologies and concepts of value chain organization which draws together Cyber-Physical Systems, the Internet of Things and the Internet of Services.”

### 1.2.2 Cluster Computing

In recent years, cluster computing has grown rapidly in the field of distributed and parallel processing. It plays an important role in IoT especially for mainstream computing. For many years, supercomputers are being used for processing huge amount of data, but as they grow, the vulnerability to failure also increases because of the complex built-in circuitry[3]. Although the processing power of supercomputers are massively increasing, they are costly if used in production. Cluster computing comes into existence as an alternative to supercomputers. A cluster is a collection of loosely coupled computers that are connected together in a parallel manner using a high-speed local area network. All devices are usually connected using Ethernet in order to make a complete networked cluster. Clusters are specifically designed to process large sets of data because the devices in the cluster work as a single virtual system. The computers work together to process data in parallel and can be viewed as a single system. In a simple computer cluster mechanism, a task is divided into small chunks and then distributed across multiple nodes. This causes the task to complete rapidly as compared to the single computer. Clusters are usually built to provide reliability and efficiency, as they are highly cost-effective and easy to deploy as compared to supercomputers [28][6].

Moreover, clusters are mainly divided into three categories [28]. The first category is the *high availability cluster* that is implemented to provide different services. This cluster is available constantly and are based on redundant nodes. There are at-least two nodes in this configuration meaning that it can tolerate one node failure. Whenever a single node fail, another node will always be available to handle those services. The next in line is the *load-balancing cluster* which provides system configuration based on the front-end and back-end servers. It operates by balancing the load on one or more font-end servers, which then distributed the load to the other back-end servers. This form of clusters is usually used to improve performance, and sometimes they can also be integrated with high availability clusters to make an effective and efficient system. The last category includes *High Performance Cluster (HPC)* which is primarily implemented for scientific computation. This cluster provides better performance by splitting the work across different nodes of the cluster.

[2]https://en.wikipedia.org/wiki/Industry_4.0
[3]http://technews.acm.org/archives.cfm?fo=2012-11-nov/nov-26-2012.html

### 1.2.3 Cloud Computing

Cloud computing is considered as an emerging concept in IoT. It provides resources and services through the use of Internet and these services are distributed in a cloud. The word 'cloud' is a metaphor which in fact describes a web where computation is pre-installed. All the applications, operating systems, storage and many other components reside in the web and are also shared with multiple users [25][4]. According to Buyya in [7], the cloud computing is defined as "a collection of inter-connected distributed and parallel virtualized computers which are dynamically provisioned and presented as one or more unified system based on service level agreements established through negotiations between the service provider and the consumer." Hence, in a broader sense, the cloud is considered as a data center in which computer nodes are virtualized based on Virtual Machines (VMs), also known as hypervisor-based virtualization, which is described in Section 2.1.1.

The main reason for using cloud computing is to solve large-scale problems by properly utilizing the resources and services distributed in the cloud. Users do not know the location of resources as these are transparent and hidden but the resources can be shared with multiple users, and are easy to access at any time. For instance, if there is a need to solve large problems then the focus of cloud computing would be to use power of hundreds of computers in order to speed up the work of engineers. Hence, it can be considered as a distributed system which offers computing services over a communication network. Furthermore, there is a lot of interest in cloud computing and many corporations and institutions have developed several platforms which are being used. As an example, Google Apps is the largest cloud service which is provided by Google and Microsoft [25].

#### Architecture

Since cloud computing is growing rapidly, it is further divided into three service models for utilizing the overall computing power [25][40]. These models are illustrated in the form of cloud computing architecture in Figure 1.1. The architecture includes:

- *Infrastructure-as-a-Service (IaaS):* Also known as hardware-as-a-service. It is the simplest service of cloud computing, which basically allows you to access the computing resources remotely. It provides processing, block storage, and network resources for computing large problems. Amazon Elastic Compute Cloud (EC2) [36] is the most common example of IaaS.

- *Platform-as-a-Service (PaaS):* This service offers a development environment and has some sets of application interfaces, which are used to develop and run software programs. It provides database, web server, operating system, run-time environment, storage, and many other resources as a computing platform. Microsoft Azure[4] and Google App Engine[5] are the two key examples of PaaS.

[4] https://en.wikipedia.org/wiki/Microsoft_Azure

[5] https://cloud.google.com/appengine/

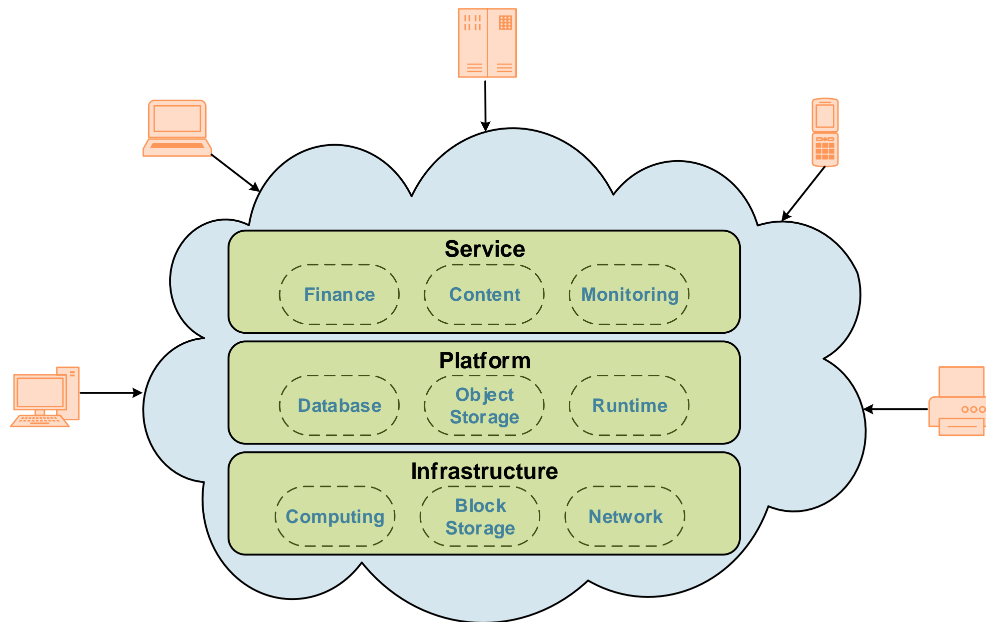


Figure 1.1: Cloud Computing Architecture

- *Software-as-a-Service (SaaS):* This is the last service model which enables users to access databases and application software. End users can easily consume this software, since it is deployed over the Internet. It is considered as a pay-per-use service and also, sometimes, charge a subscription fee. Salesforce.com and Google Docs are the typical examples of SaaS.

### 1.2.4 Embedded Systems

Embedded systems play a vital role in modern-day life. They can be found in almost every electronic device related to you and your daily life. Typically, embedded systems are defined as the computing systems which are hidden from the consumers and perform all computing tasks without requiring any external peripherals or devices such as, keyboard, mouse, and display. They are different from the ordinary personal computers and look like a smart phone or a digital camera. However, the definition is in abstract form as it is described only on the basis of physical appearance of the electronic devices, which can be misleading [18].

In general, embedded system is defined as a computing system that is specifically designed to perform a dedicated task and has hardware and software components which are tightly coupled. The system is embedded inside a large computer system which corresponds to an integral part, as the word 'embedded' speaks for itself. Due to this behavior, it is often sometimes known as embedding system which means systems within systems. Moreover, they can also be categorized into two types; one that can function on their own and the other systems that functions jointly with multiple embedded systems. The good example of standalone embedded system is the

network router containing a number of interfaces such as ports, memory, specialized processor, and some routing algorithm that can transfer packets from one port to another. Similarly, Digital Set-Top (DST) box is a good example of embedded system that do not work on their own. It is a kind of home entertainment system which has a digital audio/video decoder whose functionality is to convert multimedia stream, coming from the satellite, into video frames as output. This decoder works inside DST which might contains many other embedded systems [18].

Essentially, there are various embedded systems available nowadays ranging from consumer applications to industrial use, from commercial devices to educational use, and many others. The focus of this thesis is to use Raspberry Pi 2[6] for doing cluster computing. It is a tiny embedded board integrated with a single computer and shaped like a small credit card. It was developed in England with a purpose to learn programming. It has replaced the original Raspberry Pi 1 and has become the second generation Raspberry Pi. Unlike the original model, it has 1 GB RAM in conjunction with a quad-core ARM Cortex-A7 processor[7] of frequency 900 MHz. Like the original model Pi 1, it also has many different components and peripherals including Ethernet port, HDMI support, 4 USB ports, camera module, as well as display interface. In addition, this board can run full range of ARM Linux distributions because of the ARM processor. As an example, it can run Raspbian, Microsoft Windows 10, Ubuntu Mate, and many other operating systems.

## 1.3 Thesis Structure

This thesis is organized into six subsequent chapters. Chapter 1 introduces the project by describing motivation and research terminologies that are used to develop an IoT sensor node. Chapter 2 explains the concept of Linux virtualization along with container-based distributed systems and data computing platforms which are essential in this project. Chapter 3 takes a closer look into the implementation part which illustrates the complete configuration of Raspberry Pi cluster along with the overall system architecture. The flow of data inside cluster including pipeline configuration and containers deployment are demonstrated in Chapter 4 which also discussed about the key design decisions in the whole system. In Chapter 5, the project is evaluated based on different design parameters along with the experimental testing that is carried out on the overall cluster. In the end, Chapter 6 presents a conclusion with some limitations and future work is also suggested including some new research topics.

[6] https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[7] http://www.arm.com/products/processors/cortex-a/cortex-a7.php

# Chapter 2

# Background

The purpose of this chapter is to present the detailed description of technologies and research terminologies that will be used in the context of IoT and embedded systems data collection and processing. Since this thesis employs a technology-integrated approach to develop an IoT sensor node, the literature review emphasizes those studies that are required in order to build an efficient system. This chapter is composed of three sections in which Section 2.1 discusses the concept of Linux virtualization in terms of hypervisor and container platforms. The comparison between two virtualization techniques is also been made along with an existing framework called Docker, which is built on top of Linux containers. Section 2.2 describes two most important container orchestration systems in which one being a key backbone of this thesis which is termed as Google Kubernetes. Finally, data computing platforms are presented in Section 2.3 in which Apache Kafka is considered as a sound messaging framework for sending large volumes of data from one point to another.

## 2.1 Linux Virtualization

Due to the rapid growth in Internet technology, the demand for IT resources such as computing, software, and storage has also been increased, since the utilization of these resources become evident with huge network infrastructure. Nevertheless, there is a great need to use resources to their full potential in order to execute large-scale Internet-based applications that require substantial amount of data for processing. This causes some issues in the design of network-oriented applications regarding managing, organizing, and distributing autonomous resources, and efficiently utilizing heterogeneous resources. However, various interesting platforms such as ubiquitous computing, grids, desktop computing, and many others have been developed for resolving such issues. One of the possible solutions is to use virtual computing environment that is built on open Internet infrastructure and provides integrated and distributed services for the end-users [12].

Virtual environment or virtualization describes the resource which is separated from the underlying physical hardware in order to provide services for the computing environment. It is considered as a process of creating a virtual version of things

including hardware platforms, operating systems, storage devices, and other computing resources. In terms of Linux, virtualization creates multiple Linux operating systems on a single host computer, thus known as Linux virtualization. It commonly has two approaches; hypervisor-based and container-based virtualization, which are described in what follows.

### 2.1.1 Hypervisor-based Virtualization

This approach has been widely used since last decade for implementing virtual environment inside a large computing system. The word hypervisor, also known as Virtual Machine Monitor (VMM), is a kind of software/firmware that runs inside a virtualization stack and acts as a virtual layer. It is used to create and manage virtual machines on top of a host system which are either different operating systems or multiple instances of the same operating system, thus sharing the hardware resources between virtual platforms. As an example, you can run Windows operating system on top of a Linux host. Moreover, it provides a separate independent environment in which the applications can run in isolated manner without interfering with other applications. This way, the end-users can utilize the computing environment in order to manage and monitor multiple resources centrally [12][30].

Hypervisors are mainly classified into two types[1] which are also illustrated in Figure 2.1.

- **Type 1:** This type contains native or bare-metal hypervisors which directly run on top of a hardware and manage the guest operating systems with the help of hardware resources. The virtual machine is isolated from the hardware because it runs on a separate level. Common examples of this architecture are Oracle VM, Microsoft Hyper-V [34], VMWare ESX [22], and Xen [3].
- **Type 2:** In this type, the hypervisors are termed as hosted hypervisors which run as a software layer inside the traditional host operating system. Unlike Type 1, the virtual machines having guest operating systems are isolated from the hardware and host as well, thus making a third level software layer above hardware server. The popular examples of Type 2 hypervisors are Oracle VM, VirtualBox, VMWare Workstation [31], Microsoft Virtual PC [11], KVM [10], QEMU [15], and Parallels[2].

### Features of Hypervisor

According to [12], hypervisor has four main features:

1. *Transparency:* This feature enables software to execute within the virtual machine environment without any modification and that is independent of the underlying hardware. Hence, it allows other features such as resource sharing,

[1] `https://docs.oracle.com/cd/E20065_01/doc.30/e18549/intro.htm`
[2] `http://www.parallels.com/eu/`

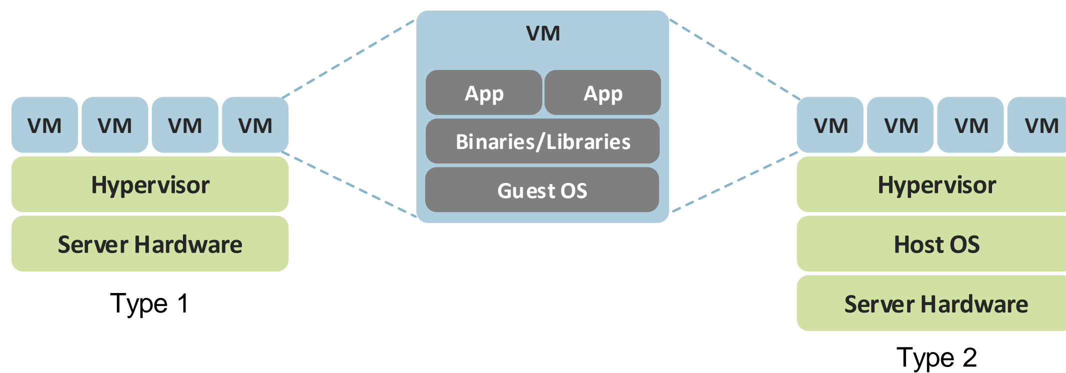


Figure 2.1: Hypervisor-based virtualization architecture

portability, migration, etc., to integrate with the same application which is running inside a virtual machine.

2. *Isolation:* This feature allows hypervisor to create and execute distinct virtual machines, isolated from each other, inside a single physical host system. Each virtual machine shares resources provided by the hardware and has its own version of software running inside it. There is a separate runtime environment for each virtual machine which helps preventing software failure caused by other softwares running in different virtual machine.

3. *Encapsulation:* This feature provides flexibility and security to the software by enclosing the whole system in a virtual hard disk. This way, the installation and backing up of virtual machines are as easy as copying files from one place to another, thus increasing the deployment and migration of virtual machines.

4. *Manageability:* This feature allows hypervisor to manage virtual machines quite easily as compared to physical systems because hypervisor has several operations such as shutdown, boot, sleep, add, or remove for managing those machines. All operations are linked with the programming interface and are controlled by the program completely.

### 2.1.2 Container-based Virtualization

Container-based virtualization is a lightweight virtualization approach that creates virtual environment at a software level inside the host machine, also known as operating system-level virtualization [30]. It removes the overhead of using hypervisors by creating virtual machines in the form of containers (act as guest systems), thereby sharing the resources of the underlying host operating system. It provides different level of abstraction in which a kernel is shared between containers and more than one process can run inside each container. This way, the whole system can be made more resource efficient as there is no additional layer of hypervisor, and thus no full

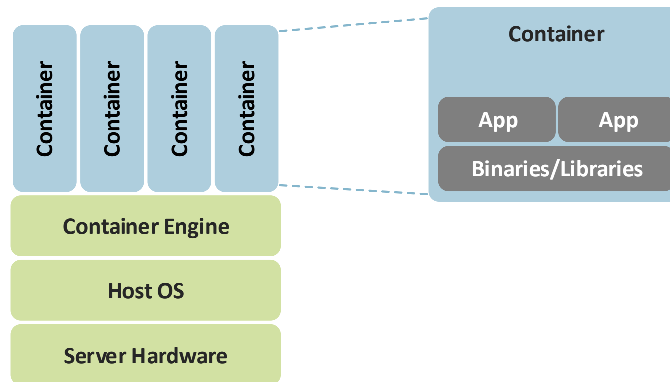


Figure 2.2: Container-based virtualization architecture

operating system which can occupy a lot of storage space for each virtual machine. In particular, it also provides isolation between multiple processes running inside a container as compared to the hypervisor-based approach [21].

As can be seen from Figure 2.2, there are four containers running on top of a host operating system and hardware resources are shared between these four guest systems. The advantage of using a container-based virtualization is to create huge amount of virtual instances on a single host machine, as the kernel and other libraries are shared between those instances. This causes the overall virtual system to have less disk storage as compared to hypervisor-based solutions. However, the approach also has some disadvantages, for instance, Windows operating system cannot run as a container application on top of a Linux host. Another trade-off is that the container cannot provide proper resource isolation as compared to hypervisor because of the shared kernel, and hence multi-tenant security can be jeopardized [21].

In this thesis, the goal is to utilize Linux containers which are considered as an alternative to hypervisors and an operating system-level virtualization. They are implemented using two important kernel features; control groups (cgroups)[3] and namespaces[4], which are basically used to provide isolation within containers and between host system and containers.

## Control groups

Control groups, also known as cgroups, are one of the main features of kernel which allow users to allocate or limit resources among groups of processes. These resources include certain features such as CPU time, system memory, disk bandwidth, network bandwidth, and monitoring. The cgroups features provide efficient resource allocation between containers and the applications running inside them. As an example, if an application requires two processes with different resource usage then that application

[3]https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html

[4]https://lwn.net/Articles/531114/

is divided into two groups. Each group has a separate profile based on cgroups resources and the processes run inside their own dedicated group. This in turn provides isolation between processes and there is no interfering between groups. Moreover, by configuring cgroups, you can also monitor them and deny resources assigned to them. It is even possible to reconfigure cgroups dynamically during run-time. Hence, by using cgroups, system administrators can efficiently control, monitor, deny, and manage system resources, therefore increasing the overall performance.

### Namespaces

Namespaces are the second important kernel feature which provide per process isolation within containers and wraps the global system state of that process on an abstract level. This feature ensures that every process has a dedicated namespace in which it can run freely without interfering with other processes running inside different namespaces. It also ensures container isolation by providing their own separate running environment. There are currently six namespaces inside Linux implementation that are described as follows:

- Mount namespaces (mnt): Deals with the filesystem mount points. This feature allows processes to have their own distinct filesystem layout. Thus, providing isolation on the sets of filesystems.
- UTS namespaces (uts): Isolate two system identifiers 'nodename' and 'domain-name'. This feature allows each container to have their own hostname different from other containers.
- PID namespaces (pid): This feature allows processes to have their own processing identifiers (IDs). These IDs are different for every process within same namespace but can be same in different namespace.
- IPC namespaces (ipc): This feature isolates the inter-process communication resources, such as POSIX message queues[5] and System V IPC objects[6], between multiple namespaces.
- User namespaces (user): Isolates name space based on user and group IDs. This namespace allows processes to have different user IDs.
- Network namespaces (net): This namespace isolates network resources by allowing containers to have their own routing devices. Thus, each network space contain separate routing tables, iptables firewalls, network interface controllers, and other items.

---

[5] `http://linux.die.net/man/7/mq_overview`
[6] `http://man7.org/linux/man-pages/man7/svipc.7.html`

### Features of Linux Containers

According to [14], there are four features of Linux containers, which helps making containers attractive and useful for the users.

1. *Portability:* Linux containers can run in any environment without changing the functionality of the operating system. The applications running inside a container can also be bundled together and then deployed onto various environments.

2. *Fast application delivery:* Containers are fast to build because they are considered as a light-weight virtualization and it takes seconds to build a new container vs minutes for a VM. System developers and administrators can easily deploy applications into the production environment, as the work-flow of containers is easy to interpret. In addition, they provide good visibility to the developers and also reduces the time for development and deployment. For instance, when developer packages an application into container then it can be shared to other team members, which then shared for testing purpose.

3. *Scalability:* It is easy to run and deploy containers in any Linux system, cloud platform, data-centers, desktop computers and many other environments. Moreover, containers can also be scale up and scale down quite rapidly meaning that you can scale up containers from one to one thousand and then scale them down again. Thus, Linux containers are suitable for scale out applications that are deployed in cloud platform.

4. *Higher density workloads:* With Linux containers, you can run huge amount of applications on a single host system. Since containers do not use the full operating system, the resources are efficiently utilized between different applications as compared to hypervisors.

After explaining both virtualization techniques, it is necessary to compare them side-by-side in tabular form. Table 2.1 illustrates the comparison between hypervisor-based (VM) and container-based (containers) virtualization in terms of different performance parameters [9].

### 2.1.3 Docker Containers

Docker containers lie in the category of Linux containers. Docker[7] was first introduced by Solomon Hykes, the founder of dotCloud, on 15th March 2013 at Python Developer's conference in California [19]. At that time, almost 40 people had given a chance to play with Docker. It is considered as a tool that can easily create a distributable application for any environment, scale that application to run independently, and configure it to interact with the outside world. In depth, Docker

[7] `https://docs.docker.com/engine/understanding-docker/`

Table 2.1: Comparison between Virtual Machines and Containers

| Parameters | Virtual Machines | Containers |
|---|---|---|
| **Operating System (OS)** | Each virtual machine runs on top of a server hardware (Type-1 hypervisors) or a host OS (Type-2 hypervisors) and kernel is assigned individually to each VM with full hardware resources | Each container runs on top of a host operating system with kernel and other hardware resources shared between those guest operating systems |
| **Startup time** | It takes few minutes to boot virtual machine | Containers can boot up in few seconds depending upon the system specifications |
| **Storage** | Hypervisor-based virtualization takes much more storage because whole system components have to be installed and run including kernel | The storage is less as compared to VM because OS is shared and so is the kernel |
| **Communication** | Typically, network devices are used for communication if VMs run on different servers but in case of same server, standard IPC mechanisms such as signals, sockets, pipes, etc. are used | Same is the case for containers but standard IPC mechanisms are more generally used |
| **Performance** | There is an additional layer of hypervisor software which degrades performance as it takes time to translate machine instructions from guest OS to host OS | There is no additional layer as container provides near native performance since it is running on top of a host |

is an open source platform that can build, deploy, and run applications faster as compared to other traditional running solutions. With Docker, the applications can be separated from the infrastructure and then you can treat that infrastructure as a managed application. By this way, it is easy for Docker to ship, run, and test your code and also shorten the life-cycle of writing a code. Docker also utilizes two most important kernel features *cgroups* and *namespaces*, as described in Section 2.1.2, to deploy and run containerize applications [19].

Moreover, Docker can run almost any application inside a container securely, thus providing isolation and security on a system level. Due to this isolation, you can

run multiple containers simultaneously on a single host without the risk of them interfering with each other. The light-weight nature of Docker helps utilizing the hardware resources efficiently, since the overload of using hypervisors are removed. Docker is simple to run in any host system. Indeed, you just need a minimal host running an updated compatible version of a kernel and a Docker binary. Since applications are running in an isolated manner inside a container, their surrounding provide a platform which can help utilizing containers in the following ways. Firstly, the applications are encapsulated inside one or more Docker containers. Then the containers are distributed and shipped to other teams for further modification. Finally, they are deployed in a production environment (local data center or cloud) [33].

Apart from local Docker instance, there are several IaaS providers that are supporting running version of Docker containers. Some of the examples are OpenStack[8], Google Cloud Platform[9], Microsoft Azure[10], Amazon EC2[11], and Carina[12]. There is also a platform named resin.io[13] that provides a Docker-based management for IoT devices. They are also incorporating security between the connected devices that helps managing a system securely.

The main objective of Docker is to provide efficient services including [33]:

- An easy way to model real world scenarios because building a container is quite easy. Docker follows copy-on-write mechanism which is also incredibly fast and you can also change some parameters according to your needs. Apart from creating containers, launching takes less than a second as there is no need to run separate hypervisor, and hence efficient use of the resources.
- Fast and efficient development life cycle which aims to reduce the time between writing the code and then testing it and being used in the real world. Hence, making the application portable, easy to use, and then shipped properly.
- Consistency of the applications as it ensures that the environment on which code is written matched with the environment on which it is being tested.

## Docker architecture and components

Docker is based on client server model. Docker client talks to Docker daemon which will then create, launch, or distribute containers. Both components (client and daemon) run on a same host, and it can also be possible to run daemon on some remote host and then connect it with the client. Docker daemon and client use sockets or RESTful APIs to communicate with each other. The overview of Docker architecture is shown in Figure 2.3. It is composed of the following core components [33]:

[8] https://wiki.openstack.org/wiki/Docker
[9] https://cloud.google.com/container-engine/docs/quickstart
[10] https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-dockerextension/
[11] http://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html
[12] https://github.com/getcarina/carina
[13] https://resin.io/

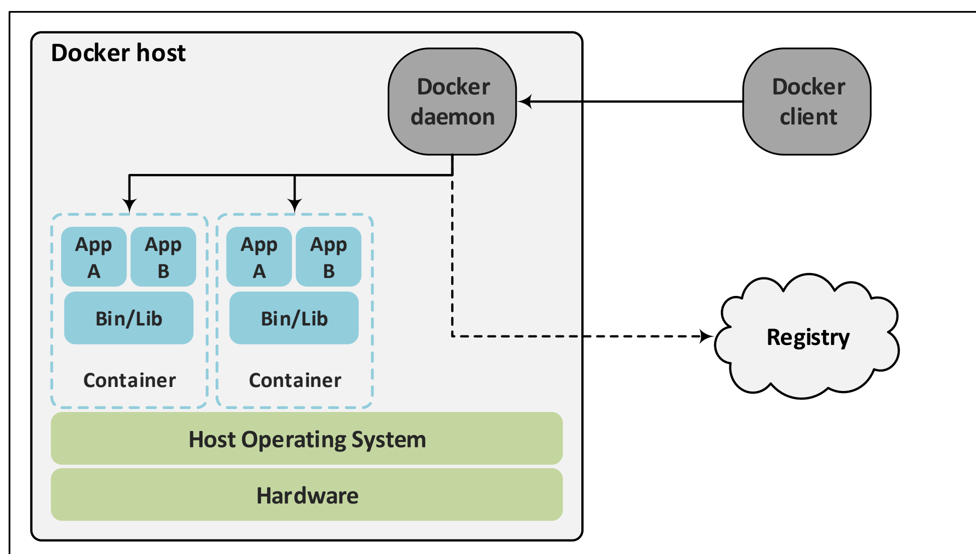


Figure 2.3: Architectural Diagram of Docker

**Docker daemon:** The main functionality of Docker daemon is to manage containers running on a host system. As can be seen from the architecture in Figure 2.3, daemon is running on the host machine and client is used to communicate with that daemon. Users cannot directly interact with the daemon. Both client and daemon use the same Docker binary, as they are on the same host.

**Docker client:** Docker client is a command line interface for users to communicate with the Docker daemon. It takes commands from the users and then interact back and forth with the daemon.

**Docker images:** Images are the main building blocks of Docker. They are also considered as a source code for running containers as Docker launches containers from these images. You can build your own image from scratch or update the existing image and it is also possible to download images which are created by some other people. Hence, they are portable and easy to create, share, and store. Moreover, images are created using multiple layers of instructions. For example, the instructions could be to run some specific command, add a file into a directory, setting the environment variable, and exposing a port for communication. These layers are combined together with the help of union file system to make a single usable image layer. Union file system[14] helps combining the files and directories of separate file system into single coherent file system. Due to this layered architecture of images, Docker is termed as a light-weight virtualization as compared to hypervisor. As an example, if you update an image or if there is a single parameter change than the additional layer is

[14] https://en.wikipedia.org/wiki/UnionFS

built which is then combine to the existing layers of image. Therefore, there is no need to replace/rebuild the image entirely as done in the traditional virtual machines. Thus, making a process simpler just by updating and distributing the existing images with the changes being added.

**Docker registries:** Registries are used to store images. They have two types; public and private. Public registries are often known as Docker Hub[15]. You need to create an account in order to access hub and store your images. It is considered as an extremely large storage system for uploading new images and downloading existing images which other people have created and uploaded. As an example, you can easily find a web-server database image or an operating system image. It is also possible to make your own private image and then store it into Docker hub. Through this private registry, you can provide security to your images so that it can only be shared with your organization.

**Containers:** Finally, container is the last in the series of Docker components. As we have already learned that containers are created and deployed using images which also contain some applications and services, and one or more processes are also running inside a single container. In other words, we can say that images are considered as the building feature of Docker and containers are considered as the execution feature of Docker. Docker adopts the concept of simple containers which are used to ship goods and items from one place to another but instead of shipping goods, it ships software. Each container has a single software image running inside it and has the ability to perform sets of operations such as to create, start, stop, delete, and move containers.

## 2.2 Container Management Systems

One of the nicest things about containers is that they can be managed specifically for cluster of encapsulated applications, especially when used in a PaaS environment [5]. As an example, suppose you have a single server machine and you literally started hundreds of container on a single server, each performing a separate task. These containers are also connected to network and run separately without interfering with each other. If you want to interact with different containers for sharing data, it would be difficult for a user to access an entire bunch of containers one at a time in order to achieve user's requirements. There is a possibility to use container management system to manage sets of containers which has a user friendly APIs. Many different systems are being developed but the two popular systems are Kubernetes (developed by Google) and Docker Swarm, which are described as follows.

[15] `https://hub.docker.com/`

### 2.2.1 Google Kubernetes

Kubernetes[16] is an open source cluster manager for containerized applications across a number of physical or virtual hosts, which provides automatic deployment, scaling, and maintenance of applications. It was developed by Google in 2014 with novel ideas and experiences from the community. With Kubernetes, you can effectively deploy your applications, scale those applications during runtime, roll out new features, and use application specific resources by optimizing hardware. This gives the benefit to quickly respond to user's demands and maintain the desired state requested by the user. Moreover, Kubernetes is (1) portable: easily run as a public, private, or hybrid system, and as well as on cloud system, (2) extensible: built as a collection of modular, compose-able, and hook-able components with the ability to use alternative distributed mechanisms, schedulers, and controllers, and (3) self-healing: automatic mechanisms such as restarting, replicating, placement, and scheduling containers for achieving robustness.

Moreover, Kubernetes is also considered as a fault-tolerant system[17], since it has the ability to preserve the correct execution of the tasks. Fault-tolerance property ensures that the system continues to execute its intended operation rather than failing completely. According to Algirdas in [2], a fault-tolerant computing system is defined as "a system which has the built-in capability (without external assistance) to preserve the continued correct execution of its programs and input/output (I/O) functions in the presence of a certain set of operational faults." This definition completely indicates the behavior of Kubernetes in real-case scenarios.

Kubernetes executes user specific containers by telling a cluster to run a set of containers. These are executed in the form of pods on particular hosts that are automatically selected by the system. The system then takes into consideration the individual and collective resource requirements, hardware/software constraints, deadlines, workload-specific requirements and other constraints in order to complete the desired work effectively. Furthermore, Kubernetes follows the concept of persistent storage to store large volumes of data that is produced by the discrete applications deployments as containers. There are different methods to store long lived data that is required by the applications but Torus[18] is a new approach that is recently developed for persistent storage. It is an open source distributed storage system that is designed to provide scalable and reliable container storage managed by Kubernetes cluster framework.

#### Kubernetes origin: Google Borg

Kubernetes has adopted many ideas from the Google Borg system [35], which is the predecessor to Kubernetes. Borg is considered as a large-scale cluster manager that runs tens of thousands of jobs across a number of clusters made up of hundreds of machines. The basic functionality of the Borg system is to schedule, start, monitor,

[16] http://kubernetes.io/docs/whatisk8s/
[17] https://en.wikipedia.org/wiki/Fault_tolerance
[18] https://coreos.com/blog/torus-distributed-storage-by-coreos.html

and restart applications which are provided by Google. It provides three main features: At first, it allows users to focus on system development instead of managing the details of failure handling and resource management. Secondly, it ensures high reliability and availability of the system so that applications having same characteristics can run efficiently. Finally, distributes all the workload on tens of thousands of machines in order to make system as effective as possible [35].

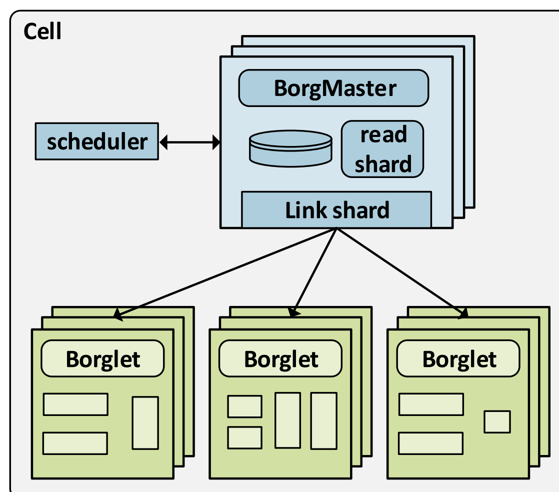


Figure 2.4: High-level architecture of Borg

The high-level architecture of Borg is shown in Figure 2.4. The architecture consists of a Borg cell which has a set of machines logically connected to each other. Each cell is composed of a centralized controller called *Borgmaster* and an agent process called *Borglet* which runs on each machine inside the cell. All these components are written in C++. In order to utilize the power of cluster, the users work is submitted to Borg in the form of jobs which are further composed of a number of tasks representing the same job in binary format. Each job is executed in one dedicated Borg cell, thus increasing the performance of the system. The detailed description of each Borg component is given below [35]:

**Borgmaster:** This is the key component of every Borg cell that further consists of a main Borgmaster process and a separate scheduler. The main process listens and handles client requests and then based on these requests, it either create a new job or provide read-only access to data. It also manages state of the machine, provides communication with Borglet, and maintains backup on a web interface. Although Borgmaster is a single logical process, it is replicated five times (by default) in a cell. Each replica has a local disk storage running a Paxos[19] based highly available and distributed store which is used to maintain and store the in-memory copy of the cell state. Moreover, a single master per cell is also elected (using Paxos) from

[19]https://en.wikipedia.org/wiki/Paxos_(computer_science)

these five replicas which acts as a Paxos leader as well as a state mutator, and all communications such as creating or terminating jobs are handled through single master. Whenever the master fails, it acquires a chubby lock which enables other machines to change the state of the system and elect a new master replica which typically takes about 10 seconds. The failed replica dynamically re-synchronizes its state whenever it recovers from an outage. As can be seen from Figure 2.4, there is a *link shard* interface in Borgmaster machine. This is basically used to handle communication with the Borglets in order to achieve performance scalability and resiliency. Since the Borglet reports its full state to the master, the link shard aggregate and compress this information in order to reduce the load at the master side.

**Scheduling:** This component handles the scheduling of a number of jobs submitted to the Borgmaster. Each job is first recorded into the Paxos store and then the tasks of each job are stored in pending queue. This queue is asynchronously scanned and the tasks are assigned to machines based on the available resources. The scan is done on the basis of round robin algorithm by selecting high to low priority tasks in order to ensure fairness across users. There are two parts in the scheduling algorithm: *feasibility checking*, which is used to find suitable machines for executing the task, and *scoring*, which is used to pick one of the feasible machines. In feasibility checking, the scheduler finds all the machines on which the task can be executed based on the task limitations and it also checks whether the machine has enough resources to execute this particular task. In scoring part, the scheduler assigns a 'goodness' score to each feasible machine and this score is mostly driven by built-in criteria scheme. Based on the goodness score, the task is assigned to the machine and that is how the whole job is being executed.

**Borglet:** The Borglet is considered as a local Borg agent which runs on every machine in a Borg cell. It manages and monitors the set of tasks running inside the machine and performs various number of operations such as to start, stop, and restart tasks. It uses OS kernel settings to manage local resources required by the task and also provides communication with the Borgmaster to report the state of the machine. The Borgmaster retrieves the current state of the machine by sending poll requests to each Borglet after every few seconds. If there is no reply from Borglet after sending certain polls, then the machine is marked 'down' and all the tasks which were running on that machine are rescheduled on some other machine. When the communication is restored, the machine kills all those rescheduled tasks, on the request of Borgmaster, to avoid duplicate execution. This in turn means that the Borglet will stay active if there is no contact between the machine and Borgmaster.

## Kubernetes architecture[20]

In order to configure Kubernetes for managing containers, there are few components/services which need to be installed on top of a physical or cloud-based host. As

[20] `https://github.com/kubernetes/kubernetes/blob/release-1.1/docs/design/architecture.md`

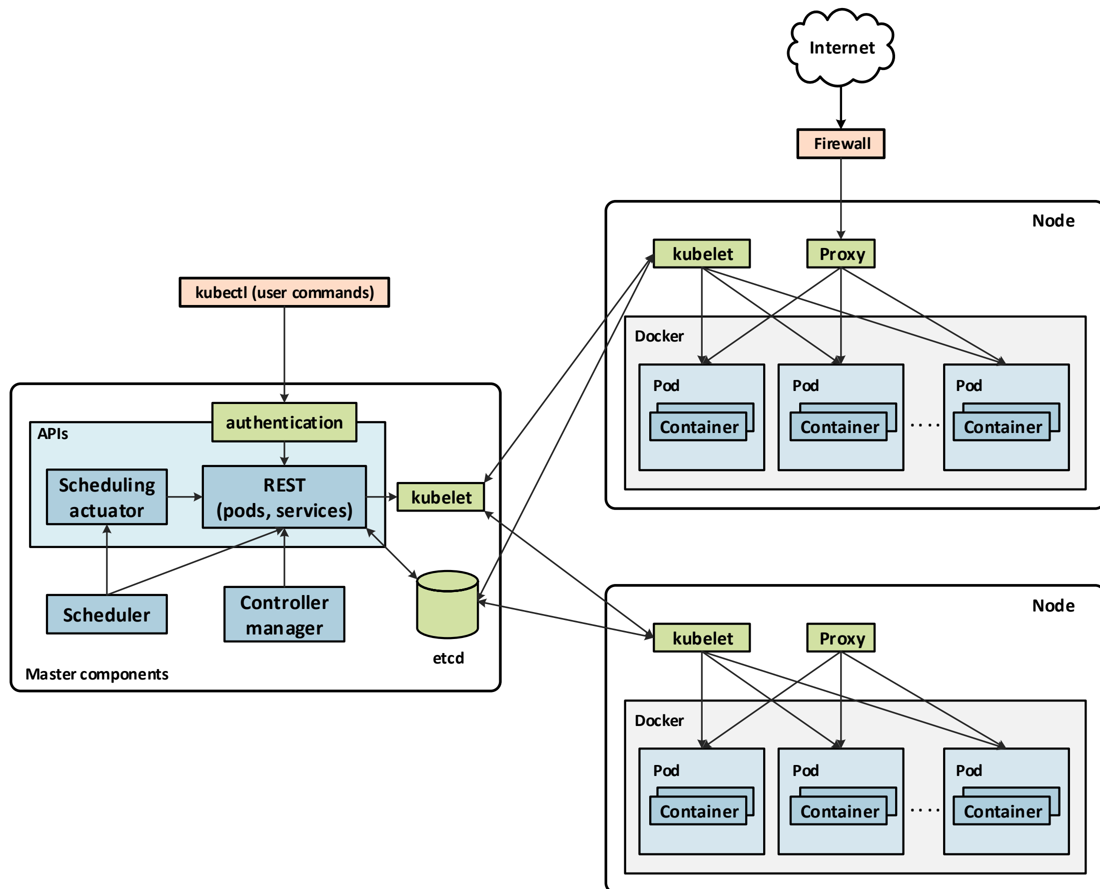


Figure 2.5: High-level architecture of Kubernetes

can be seen from the system architecture in Figure 2.5, the components are divided on the basis of master components such as APIs, scheduler, controller manager, and kubelet on master node, and node agents such as proxy and kubelet on worker nodes. This gives the opportunity to fully utilize the power of cluster computing by running containers across a number of worker nodes which are distributed and managed by master node. In addition, you can also configure one node cluster by installing all Kubernetes components on a single host which, later, can act as both master and a worker node[21].

**Master node components:** Master node server is the controlling unit of Kubernetes cluster which serves as the main contact point for system administrators and users. It deploys, manages, and monitors containerized applications on the worker nodes. This is achieved with the help of few dedicated components that are described as follows:

- *API server:* Kubernetes API server is the key component of the entire cluster.

[21] `https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes`

It allows users to configure organizational units and workloads for enabling communication to the other nodes in the cluster. This server also validates and configures data by implementing RESTful interface which includes API objects such as pods, replication controller, services, and many others. A separate client called kubectl[22] is used along with other tools on the master side, to connect users with the cluster. This gives users a control over the entire cluster.

- *Controller manager:* The controller manager service is a daemon which is used to handle replication tasks defined by the API server. This service is built using a control loop whose purpose is to watch changes in the state of the operations that are also stored in etcd side-by-side. Whenever a change occurs, the control manager reads the new information and moves the current state to the desired state. Some examples of controllers are replication controller, namespace controller, endpoints controller, and service-accounts controller.

- *Scheduler:* The scheduler actually assigns workload to specific worker nodes in the cluster. This is primarily used to analyze the working environment, read the operating requirements, and then place workload on acceptable nodes. This is also responsible for tracking resource utilization on different nodes and keep track of the total hardware resources that are either assigned to processes or free to occupy.

- *etcd:* Etcd is a distributed, reliable, and consistent key-value data-store for shared configuration and service discovery, and considered as one of the core components of Kubernetes cluster. Etcd provides simple, secure, fast, and reliable storage space for storing configuration data that can be used by the cluster. It follows the principle of raft consensus algorithm to operate as a data-store.

Raft is a consensus algorithm[23] that is designed to manage a replicated log in a cluster [23]. Before explaining Raft, the question arises that What is consensus? In a fault-tolerant distributed system, consensus is considered as a fundamental problem that is achieved when multiple servers decide to agree on values, and the decision is final. Typically, the consensus algorithm has to have network access to the majority of servers in order to make progress. As an example, if there is a cluster of five servers and two of them fail, then it can still continue to operate until more servers go down. In addition, the consensus problem usually arises in the context of building a fault-tolerant system, as each server in the system has a replica of the distributed log maintained by Raft. Such a log can be used to build a replicated database.

A Raft cluster usually contains numerous servers but five is a typical number, meaning that system can tolerate two failures. Each server in the cluster is in one of the following three states: *leader*, *follower*, and *candidate*. In a normal scenario, there is only one leader and the rest of the servers are followers. Leader server is a point of interaction which handles all client requests. Follower servers are passive

[22] `http://kubernetes.io/docs/user-guide/kubectl-overview/`
[23] `https://raft.github.io/`

entities that only respond to requests from leaders but do not serve any requests. If, in some case, the client contacted the follower server then the request is redirected to leader in order to respond quickly and properly. There is also a third state named candidate which is used to elect the new leader if the current leader fails to respond. In this operation, Raft implements consensus by first electing a distinct leader from the set of followers. Then, the complete responsibility is given to that newly selected leader for managing the replicated log. The leader receives log entries from different clients and then replicates them on other servers. It also tells servers to apply these entries to their state machines. Through this way, the cluster replicates up-to-date data which also allows followers to maintain data flows [23].

**Worker node components:** Each worker node requires a few components that are necessary to communicate with the master node and runs actual workload assigned to them. For networking, each worker node is also assigned a dedicated subnet which is used by the containers. The components are described below:

- *Docker service:* The first requirement is to run Docker service on each worker node that is used to create containerized applications and perform dedicated work. Each unit of work is deployed as a series of pods and different IP address is assigned to each pod from the same subnet.

- *Kubelet service:* Kubelet is the central component of Kubernetes which is primarily used to manage pods and their containers on the local system. Kubelet service functions as a node agent and enables worker nodes to interact with the etcd store for configuration update, as well as to receive commands from the master node. It is also responsible for registering a node with the Kubernetes cluster and reporting resource utilization.

- *Proxy:* Proxy service, also termed as back-end service, is running on each worker node which is used to make applications available to the external world. This service forwards requests to the correct container by providing primitive load balancing and allows host to manage an isolated, predictable, and accessible networking environment.

**Kubernetes work units:** Kubernetes deploy containers for performing user specific tasks in terms of workload. The deployment is carried in the form of work entities that are easily managed by Kubernetes. These work entities are:

- *Pod:* A pod[24] is the basic unit of work that is created and managed by Kubernetes. It allows users to bundle closely related containers into one pod which then acts as a single application. Instead of running a single individual container, you can put together one or more containers and then deploy them as a pod. This in turn schedules all related containers into one host which shares the same environment and manages as a single unit. This means that they can also share IP space and volume directories. In other words, you can

[24] http://kubernetes.io/docs/user-guide/pods/

conceptualize pods as a single virtual host including all the resources needed to carry their operation. From a design perspective, a pod usually contains main container (also called as front-end container) which satisfies basic needs and one or more helper containers (called as back-end container) that facilitate related tasks by providing useful data.

- *Replication controller:* A replication controller[25] (often abbreviated as 'rc') is considered as a complex version of replicated pod. It ensures that a specified number of pod replicas are running which are continuously monitored by Kubernetes controller. It allows replicated pods to be up and running all the time. For instance, if there are too many replicas running then it will try to reduce the number. Similarly, it will increase pod replicas if few pods are running. Unlike manually created pods, the replication controller pods are automatically started if they fail, get deleted, or are terminated in case of system malfunction. You can consider rc pod as a process supervisor that can easily manage multiple pods across multiple hosts instead of just managing a single pod.

- *Services:* A service[26] is an abstraction that provides a policy to access multiple sets of pods containing distinct IP addresses. It usually uses label selector field to target specific pods and combine them in a set. While pods are mortal, they can easily die and they are not restored. When a pod is created, a unique IP address is automatically assigned to it but this address is not stable over time. For instance, consider a pod which is created by the replication controller and multiple replicas are running. If, for some reason, one of the replicas get terminated then the rc will start another pod replica in order to balance the specified number of replicas. This causes a newly created pod to have a new IP address. This in turn leads to a problem in which some sets of pods (front-ends) are unable to find other pods (back-ends) in the same set. Hence, Kubernetes service is created which keeps track of the pods that belong to the same set. In addition, Kubernetes also offers simple endpoints API which is used to access that service externally. The endpoints are updated whenever the set of pods in the service changes. This causes pods to provide communication between front-end and back-end pods quite easily.

### 2.2.2 Docker Swarm

Docker Swarm[27] is a container orchestration tool for providing production scale native clustering and scheduling capabilities for Docker. It is also considered as a container management system which is used to manage and monitor containerized applications across a number of distributed hosts. It works by turning a pool of Docker engines, running on multiple Docker hosts, into a single virtual host system which acts as a Swarm manager. This in turn acts as a cluster in which there is one master node

[25] http://kubernetes.io/docs/user-guide/replication-controller/
[26] http://kubernetes.io/docs/user-guide/services/
[27] https://docs.docker.com/swarm/overview/

and all the communications are done through this node. Through this way, there is no need to communicate directly to each Docker engine, thus removing the overhead of accessing multiple hosts. The cluster mechanism is shown in Figure 2.6. It can be seen from the architectural diagram that there are three swarm nodes and one manager node. The client communicates with the manager to execute commands and run containers on different nodes attached to that manager node. In order to interact with the Docker daemon, various number of tools are available such as Docker Compose[28], Krane, Jenkins, and many others.

Moreover, Docker Swarm follows the principle of "swap, plug, and play". As an initial development, it serves the standard Docker API which enables pluggable back-ends that can easily be swappable with the back-end you prefer. Discovery service of manager node is used to provide those back-ends. The service maintains a list of IP addresses in your cluster which are assigned to each back-end service.

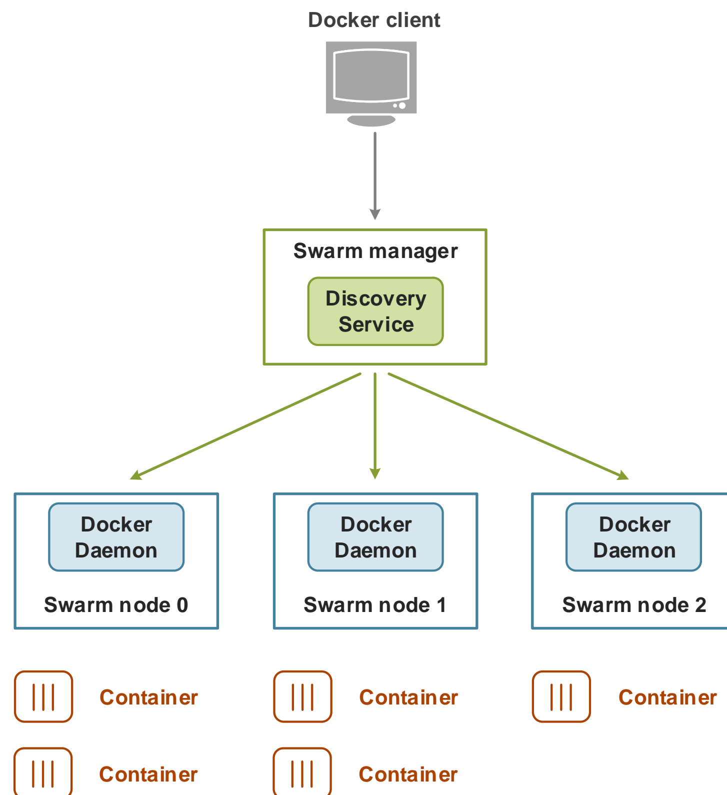


Figure 2.6: Architectural diagram of Swarm

[28] https://docs.docker.com/compose/

## 2.3 Data Computing Platforms

With the rapid advancement of technology, organizations are producing vast amounts of data which is commonly referred to as Big Data [8][17]. According to SAS[29], Big Data is defined as "a paradigm containing large volumes of data that inundates a business on a day-to-day basis, and it is generated by everything around us which also categorizes into structured and unstructured forms". Typically, this data comes from business processes, sensors, mobile devices, website tracking, accounting, and many other sources. Another main source of data production is the emergence of social networking websites which are used by users to create and store records of their daily activities. Similarly, Internet-based companies generate huge amount of 'log' data which typically includes two sources: (1) user activity events regarding social networking such as likes, clicks, sharing, reviews, logins, and search queries; (2) utilization of system and operational metrics such as CPU, memory, disk, network, errors, call stack, and call latency. This shows log data as a component of analytics that is used to track system utilization, user behavior, and other metrics [16]. Since the data has an enormous volume, it poses two main challenges. The first challenge is to collect data and the second one is to process the collected data.

In recent years, several specialized platforms have been developed for collecting and processing log data which relies on physically scrapping log files. Some of the examples are Hadoop [29] which is basically designed for offline consumption, and Cloudera's Flume[30], Google Cloud[31], and Facebook's Scribe [32] which primarily designed for collecting the data into a data warehouse. At LinkedIn, another log processing system has been built, called Kafka [37], that combines the benefits of messaging systems and traditional log processing systems. This platform is described in the following section.

### 2.3.1 Apache Kafka

Apache Kafka[32] is a distributed, partitioned, and replicated publish-subscribe messaging system that is used to send high volumes of data, in the form of messages, from one point to another. Kafka replicates these messages across a cluster of servers in order to prevent data loss and allows both online and offline message consumption. This in turn shows the fault-tolerant behavior of Kafka in the presence of machine failures that also supports low latency message delivery. In a broader sense, Kafka is considered as a unified platform which guarantees zero data loss and handles real-time data feeds. In order to understand the functionality of Kafka, there are few methodologies that need to be discussed before moving deep into the architecture. A stream of messages is divided into particular categories called *topics*. These messages are published to specific topics using dedicated processes which we call *producers*. The published messages are then stored in the set of servers called *brokers*. A separate set

[29] `http://www.sas.com/en_us/insights/big-data/what-is-big-data.html`
[30] `http://archive.cloudera.com/cdh/3/flume/UserGuide/`
[31] `https://github.com/GoogleCloudPlatform/processing-logs-using-dataflow`
[32] `http://www.tutorialspoint.com/apache_kafka/apache_kafka_quick_guide.htm`

of processes called *consumers* are used to pull data from the brokers by subscribing to one or more topics. At a high level, we can say that the producers send messages to the Kafka cluster which then consumed by the consumers [16]. Kafka has the following benefits.

- **Durability:** Kafka allows messages to persist on the disk in order to prevent data loss. It uses distributed commit log for replicating messages across the cluster, and thus making it a durable system.
- **Scalability:** It can easily be expanded without any downtime. Since a single Kafka cluster is acting as a central backbone for handling the large organization, we can elastically spread it to multiple clusters.
- **Reliability:** It is reliable over time, as it is considered as a distributed, replicated, and fault tolerant messaging system.
- **Efficiency:** Kafka publishes and subscribes messages efficiently which shows high system throughput. It can store terabytes of messages without any performance impact.

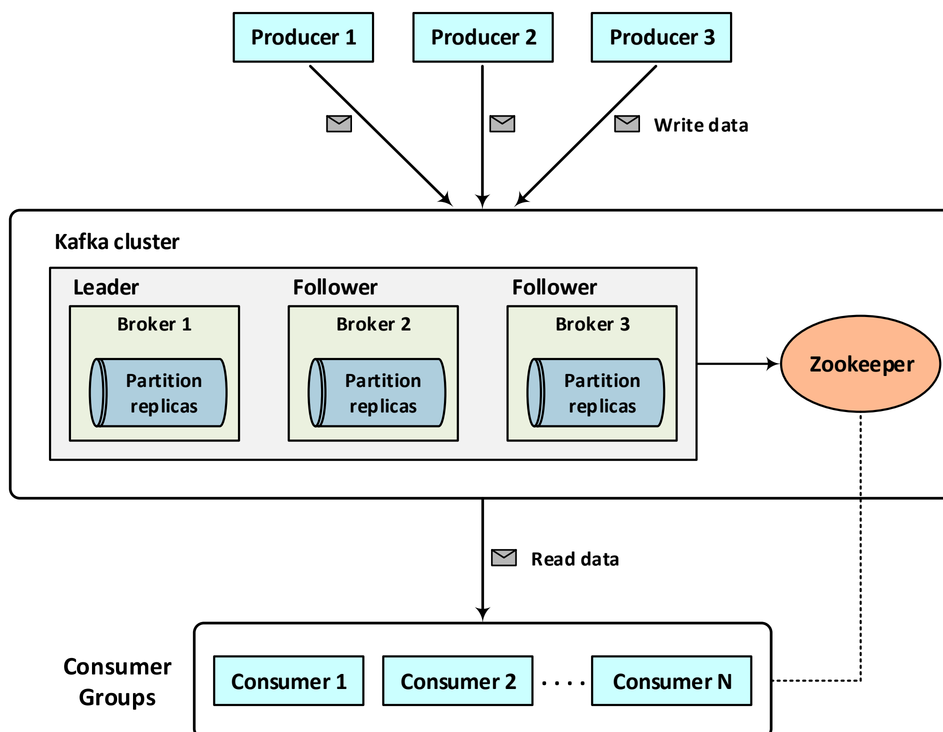


Figure 2.7: Clustered architecture of Kafka

The overall architecture of Kafka is shown in Figure 2.7. It is composed of three server machines which together act as a cluster computing platform. In a typical

Kafka cluster, each server is configured to behave as a single broker system that shows the persistence and replication of message data. In other words, we can say that there is more than one broker in a typical Kafka cluster. Essentially, broker is the key component of Kafka cluster which is basically responsible for maintaining published data. Each broker instance can easily handle thousands of reads and writes per topic, as they have a stateless behavior. At a basic level, Kafka broker uses topics to handle message data. The topic is first created and then divided into multiple partitions in order to balance load. Figure 2.8 illustrates the basic concept of topic which is divided into three partitions. Each partition has multiple offsets in which messages are stored. As an example, suppose that the topic has a replication factor of value '3', then Kafka will create three identical replicas of each partition regarding the topic and distribute them across the cluster. In order to balance load and maintaining data replication, each broker stores one or more partition replicas. Suppose that there are N brokers and N number of partitions then each broker will store one partition.

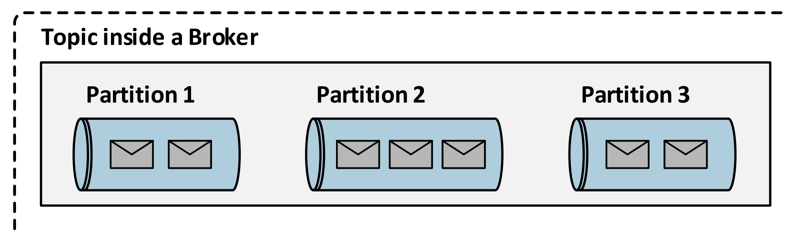


Figure 2.8: Anatomy of a Topic

Moreover, Kafka uses Zookeeper to maintain cluster state. Zookeeper is a synchronization and coordination service for managing Kafka brokers and its main functionality is to perform leader election across multiple broker instances. It will be described in detail in Section 2.3.2. As can be seen from Figure 2.7 that one server acts as a leader and the other two servers act as followers. *Leader* node handles all reads and writes per partition. *Follower* node just follows the instructions given by the leader node. If the leader fails, then the follower node will be automatically appointed as a new leader.

Furthermore, Kafka uses producer applications to send data to the topics of their choice. Producers send data to the brokers which then append messages at the end of the topic queue. Whenever a new broker started, the producers automatically send messages to this new broker. Each producer has two choices to publish data to the partition; they can either select the partitions randomly or they can semantically use the partitioning key and function to determine specific partitions. In order to pull data from brokers, Kafka uses consumer applications. The consumers maintain offset value which shows the number of messages being consumed. Each broker is listening for an asynchronous pull request from the consumers and then update the buffer bytes to be consumed next. Kafka also offers a single abstraction called *consumer group* which consists of one or more consumers having the same group id and name

as well. All groups are used to jointly consume messages by first subscribing to a particular topic and each message is delivered to only one consumer instance within a subscribed group. This shows the load balancing nature among the consumers and the coordination overhead is also neglected.

### 2.3.2 Apache Zookeeper

Apache Zookeeper[33] is considered as a critical dependency of Kafka which provides coordination and synchronization service within a cluster. It is impossible to run Kafka without running Zookeeper first. This framework was originally built at Yahoo for managing services in a distributed environment [13]. Zookeeper has a simple architecture and multiple client APIs through which it can easily manage more complex distributed services. In Kafka system, it provides coordination interface between the brokers and the consumers. The leader election between brokers is also done with the help of Zookeeper. Kafka servers uses Zookeeper cluster to share information regarding topics, brokers, and buffer offsets which are also stored in Zookeeper for safe keeping.

In a distributed environment, Zookeeper uses shared hierarchal namespace to provide coordination between distributed processes. The namespace has data registers called *znodes* and is similar to standard file system containing files and directories. A name in namespace corresponds to a sequence of path elements that are separated by a slash (/). Each *znode* in Zookeeper namespace is identified by a unique path. Unlike a typical file system which is designed for storage, these znodes are kept in memory which shows the efficiency of the system in terms of low latency and high throughput. In order to understand the functionality of Zookeeper, the architecture has some sets of nodes called servers and clients. Zookeeper can be replicated as a server node onto a number of machines attached within the cluster. These sets of nodes called ensemble make up a Zookeeper service in which every server must know each other. The service maintains the state of the system along with the transaction logs and snapshots. This in turn shows the high availability of Zookeeper service which corresponds to the majority of servers being available. In other words, it shows that the Zookeeper will be available, if majority of servers are up and running. Apart from server nodes, clients are used to connect to a single server node. Each client uses TCP connection to send specific requests and then wait for the response. If the TCP connection breaks, the client will make contact to another server.

[33] `https://zookeeper.apache.org/doc/trunk/zookeeperOver.html`

# Chapter 3

# Design and Implementation

Various concepts and technologies have been described so far along with an overview of the key terminologies which are used in our project. This chapter starts by explaining the implementation phase of IoT sensor node. The overall implementation is divided into two phases. The first phase discusses Kubernetes configuration along with the installation of its processes inside a cluster of five nodes. The overall procedure is described in this chapter while Chapter 4 focuses more towards data pipelining and processing which is the second phase of our sensor node implementation. This chapter is composed of two sections in which the first Section 3.1 shows the graphical representation of RPi and Kubernetes cluster in the form of a systematic diagram in conjunction with the hardware components that are used to build a cluster at first place. Section 3.2 describes the step-by-step procedure to configure Kubernetes on top of a RPi cluster. This section further proceeds by explaining kubernetes components in terms of master and worker nodes.

## 3.1 System Architecture

The architectural overview of the proposed IoT sensor node is demonstrated in Figure 3.1. According to the architecture, there are five Raspberry Pi boards connected to a central hub called network switch. These boards use Ethernet to interact with the switch, and thus producing a cluster of five system nodes. The other end of this clustered system is attached to a computer server in order to provide Internet and communicate over the network. There are also five camera sensors attached to each Raspberry Pi that are playing a vital role for making IoT sensor node. Table 3.1 illustrates the required hardware components along with the quantity to assemble a required sensor node.

In this thesis, Raspberry Pi 2 has been chosen for making sensor node because this was the latest model when the project started. Nowadays, the third model of Raspberry Pi has also been released which is quite similar to the older model with few minor changes. Raspberry Pi 3[1] is compatible with the older models as it has an identical

[1] `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`

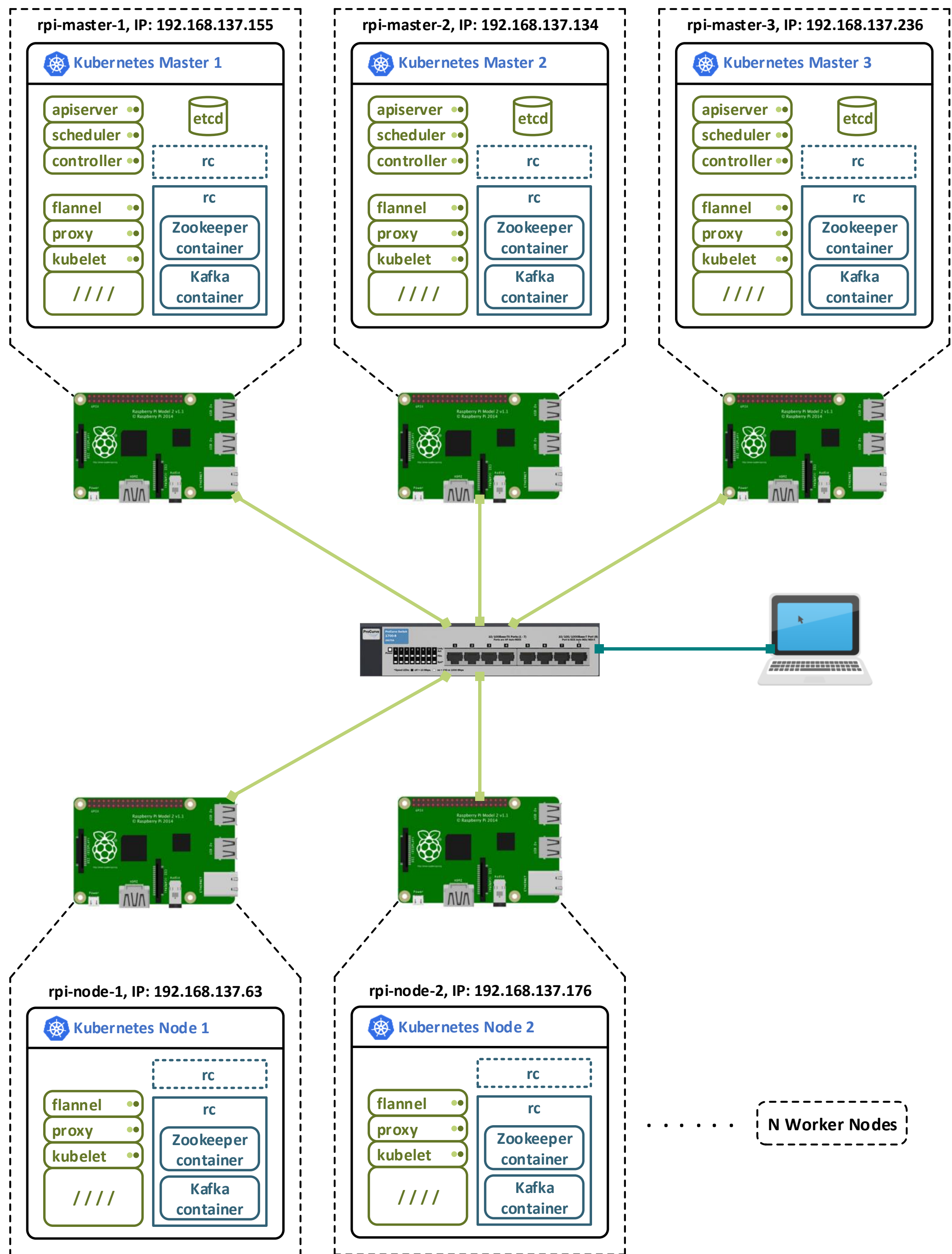


Figure 3.1: IoT sensor node overview

Table 3.1: Required Hardware components

| Hardware components | Quantity |
|---|---|
| Raspberry Pi 2 Model B | 5 |
| ProCurve 1400-8G Network Switch | 1 |
| Pi Camera module version 2 | 5 |
| Adata Micro-SD card 16GB | 5 |
| Ethernet Category 5 cable | 6 |
| USB power supply charger | 5 |
| USB cable | 5 |

form factor. This means that it can be easily swappable with version 2, specifically for this project, without any performance impact. For the switch, ProCurve 1400-8G is selected, since it is a network switch which provides plug-and-play simplicity for high-bandwidth connectivity. As can be seen from Figure 3.1, each RPi node runs some specific processes that are necessary for the sensor node implementation. The green boxes illustrate those processes that are required in the configuration of Kubernetes cluster and these are described in Section 3.2. Similarly, the processes in blue correspond to a data communication and processing, which is the second phase of implementation and will be described in Chapter 4.

## 3.2 Raspberry Pi Cluster

This section describes the development of IoT sensor node by first creating a Raspberry Pi cluster using the hardware components shown in Table 3.1. The cluster implementation is further divided into two phases. Section 3.2.1 describes the first phase as a basic setup of cluster computing in which all five RPi boards are connected to the switch and then proper software is installed to access those boards. In the second phase (in Section 3.2.2), Kubernetes is configured across a cluster of five nodes in order to operate as a Kubernetes master-worker mechanism. The detailed procedure is described in what follows.

### 3.2.1 Basic setup

The implementation starts by connecting the hardware components in a correct manner as described in the system architecture diagram in Figure 3.1. After connecting all those components, it is time to install the software on each Raspberry Pi. Hypriot operating system version 0.7.0 has been selected since it is a Raspbian-based Linux distribution for the ARM processors. The main reason of using Hypriot is that it has

the bundled support for Docker. The instructions given on the webpage[2] are used to install and configure Hypriot on each Raspberry Pi. Once the installation has been done, next step is to connect the other port of a switch with a computer server so that Internet is available and every RPi gets an IP address dynamically via DHCP protocol. Since there are multiple RPi nodes in the cluster, it is practical and useful to give a unique hostname to each Pi. As can be seen from Figure 3.1, there are three master nodes and two worker nodes which helps making a Kubernetes cluster, therefore the hostnames are; *rpi-master-1, rpi-master-2, rpi-master-3, rpi-node-1,* and *rpi-node-2*. In Hypriot, these hostnames can be changed by editing `/boot/occidentalis.txt` file, instead of editing `/etc/hostname`, on every RPi system.

Although there are two worker nodes in our implementation, more workers can be added to the cluster in order to provide greater scalability (as shown in Figure 3.1 with a letter N). Similarly, more master nodes can also be integrated with the cluster as well. The next step is to configure Kubernetes master and worker components on each corresponding RPi board, which is described in the following section.

### 3.2.2 Kubernetes configuration

This section explains the overall configuration of Kubernetes by installing different components/processes across a cluster of five nodes. There are two methods to configure Kubernetes; either by installing directly on a bare machine or using Docker containers. In this implementation, the latter approach has been selected because containers are easy to handle and executed. In order to behave like a Kubernetes cluster, several processes (as shown in Figure 3.1) are required to be installed on different Raspberry Pi systems in terms of master node and worker node components. These processes should be executed in a specific order in order to ease Kubernetes startup. This is done by utilizing Linux systemd[3] services which helps starting processes in a right order. In addition, these services ensure that the Kubernetes will start automatically whenever a RPi is rebooted. The installation is further divided into following two parts.

#### Kubernetes master node installation

For configuring Kubernetes master node, seven processes in separate containers should be executed with specific order. It is also demonstrated in Figure 3.2. Different container images are used for separate processes which are already stored in Docker registry. The processes are etcd, flannel, kubelet, apiserver, controller, scheduler, and proxy. Four of them (etcd, apiserver, controller, and scheduler) are only specific to master node and are not installed on worker nodes. Appendix A.1 shows the service configuration files of the above mentioned processes. Three variables `K8S_MASTER1_IP`, `K8S_MASTER2_IP`, and `K8S_MASTER3_IP` are important in those files which correspond to the IP addresses of the three master nodes. In our cluster, the IP addresses are `192.168.137.155`, `192.168.137.134`, and `192.168.137.236`

[2] `http://blog.hypriot.com/getting-started-with-docker-on-your-arm-device/`
[3] `https://www.linux.com/learn/understanding-and-using-systemd`

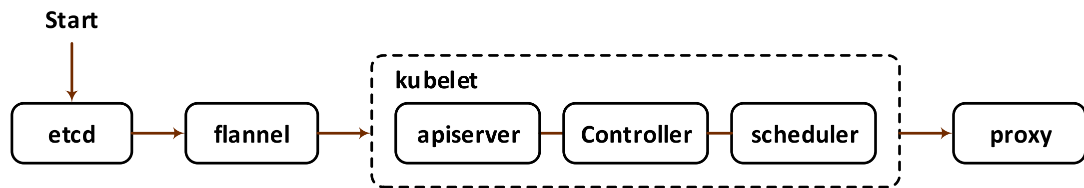


Figure 3.2: Execution order of Kubernetes processes

for *rpi-master-1, rpi-master-2,* and *rpi-master-3* respectively. The official Kubernetes tutorial on the blog page[4] is used for Kubernetes installation along with the few modifications in the configuration files.

**Etcd:** In order to provide high-availability for Kubernetes cluster, etcd is used which is installed on master nodes as a cluster of three etcd servers. These nodes are termed as primary, secondary, and tertiary nodes which depict the availability of Kubernetes cluster in case of a node failure. This behavior is achieved by starting etcd which is also the first process to be started for master node setup. Etcd is a distributed data-store for storing and sharing cluster configuration inside distributed environment. The detailed description of etcd is already described in Section 2.2.1. Here, its configuration is explained in order to achieve highly available cluster in terms of three master nodes. Table 3.2 shows the key parameters for configuring etcd. These parameters are the flags of etcd which are used while running etcd as a containerized application. Moreover, it runs as a systemd service meaning that it will automatically start whenever a RPi boots up.

According to Table 3.2, the *-name* flag assigns any user-defined name to different etcd member nodes (all three RPi on which etcd is running). The flag *-initial-cluster* defines an offline bootstrap configuration since the IP addresses of the master nodes are known. Thus, it is important to mention name and the complete peer URLs of each etcd member in the cluster. These URLs are the advertised peer URLs and it should match with the value of *-initial-advertise-peer-urls* flag on the respective nodes. In order to accept traffic from the clients, *-listen-client-urls* flag is used in which generic IP is mentioned. This IP shows that all addresses on the node is now in listening mode. Etcd uses *-advertise-client-urls* flag for remote clients to reach etcd cluster, that's why it is necessary to mention the IP address instead of writing the keyword localhost. Through this way the node is reachable from the intended clients. Since, there is a single cluster with same configuration file for etcd, it is important to mention the unique id for that cluster and this is handle using *-initial-cluster-token* flag. The last flag *-initial-cluster-state* defines the state of the cluster by setting its value to 'new', as this is the initial static cluster instead of any other existing cluster.

[4] `http://blog.kubernetes.io/2015/12/creating-raspberry-pi-cluster-running.html`

Table 3.2: Etcd parameters for running in the cluster

| Flag | On rpi-master-1 | On rpi-master-2 | On rpi-master-3 |
|---|---|---|---|
| -name | etcd1 | etcd2 | etcd3 |
| -advertise-client-urls | http://192.168.137.155:4001, http://192.168.137.155:2379 | http://192.168.137.134:4001, http://192.168.137.134:2379 | http://192.168.137.236:4001, http://192.168.137.236:2379 |
| -listen-client-urls | http://0.0.0.0:4001 | http://0.0.0.0:4001 | http://0.0.0.0:4001 |
| -initial-advertise-peer-urls | http://192.168.137.155:2380 | http://192.168.137.134:2380 | http://192.168.137.236:2380 |
| -listen-peer-urls | http://0.0.0.0:2380 | http://0.0.0.0:2380 | http://0.0.0.0:2380 |
| -initial-cluster-token | etcd-cluster-1 | etcd-cluster-1 | etcd-cluster-1 |
| -initial-cluster | etcd1= http://192.168.137.155:2380, etcd2= http://192.168.137.134:2380, etcd3= http://192.168.137.236:2380 | etcd1= http://192.168.137.155:2380, etcd2= http://192.168.137.134:2380, etcd3= http://192.168.137.236:2380 | etcd1= http://192.168.137.155:2380, etcd2= http://192.168.137.134:2380, etcd3= http://192.168.137.236:2380 |
| -initial-cluster-state | new | new | new |

After starting etcd cluster on the master nodes, the next step is to configure clients with a list of etcd members. This can be done by executing a command using command line tool called `etcdctl`. This command takes the advertised client URLs of all three master nodes and then register each of them inside member list. The command is also shown in the etcd configuration file in Appendix A.1.

**Flannel:** Once the etcd has been started, the next step is to provide container networking within a cluster. This is achieved by using flannel[5] which is a virtual network for assigning unique IP addresses to containers. It is basically used with Kubernetes for providing container-to-container communication over the network. Flannel works by allocating a unique IP subnet (`/24` by default) to each host machine (each Raspberry Pi) so that the containers can use that subnet to get their own IP addresses. It has an agent called 'flanneld' which runs on each node and allocates subnet lease from a pre-configured network space. This in turn creates a virtual overlay network by using packet encapsulation and that network spans the whole Kubernetes cluster. This overlay network can be first configured by giving IP range and subnet size for each RPi. For example, overlay is configured to use IP range `10.0.0.0/16` and each Raspberry Pi is given subnet `/24`, then it might possible to receive `10.0.43.1/24` for first Pi system and `10.0.19.1/24` for the second Pi system. It is also illustrated in Figure 3.3. Initially, flanneld assigns `10.0.43.0/24` and `10.0.19.0/24` subnets on *rpi-master-1* and *rpi-master-2* nodes respectively. This in turn used by docker service to select appropriate addresses for the containers running on a particular nodes.

Additionally, flannel uses etcd to store data configuration and subnet assignments. Upon startup, a flannel agent first checks the configuration in etcd which has been

[5] `https://coreos.com/blog/introducing-rudder/`

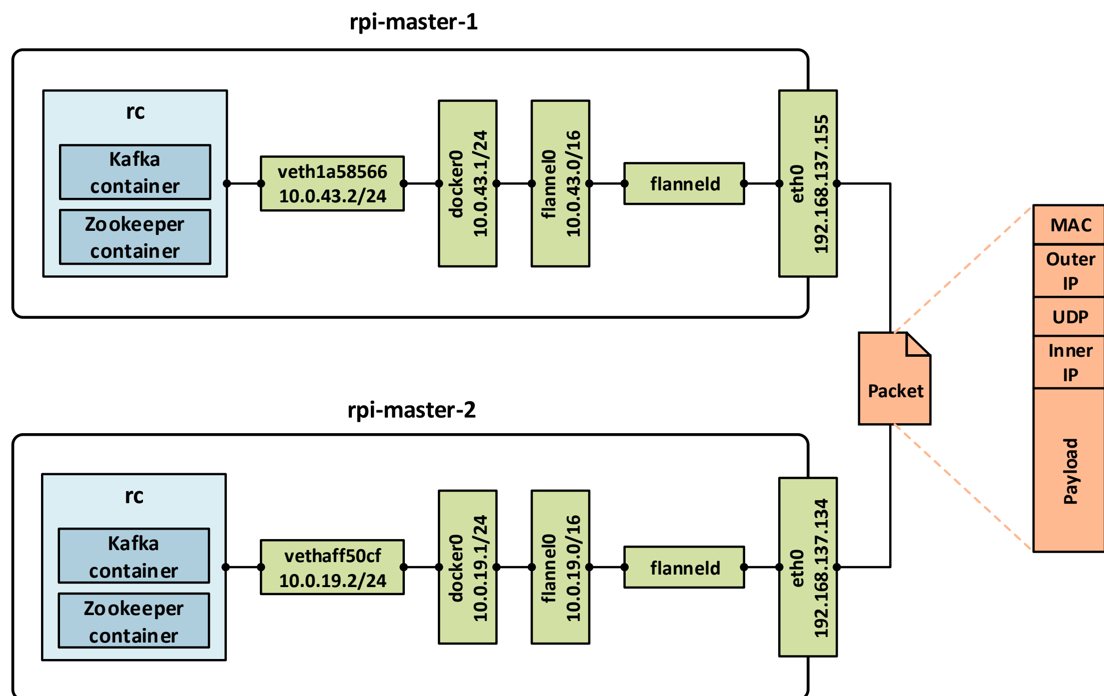


Figure 3.3: Working scenario of Flannel

published while configuring flannel. By default, flannel uses `/coreos.com/network/config` to find network configuration. It then retrieves the same configuration in which a list of IP subnet is stored. The agent then selects an available subnet randomly for a particular host and try to register it by creating a key in the etcd data store. This procedure is done for all RPi nodes present in the cluster.

In this project, flannel also runs as a service, like etcd, which will automatically start whenever a RPi boots up. The configuration file of this service is shown in Appendix A.1. It uses *-etcd-endpoints* flag to tell flannel network regarding the data-store running on a particular nodes. So that it can interact with the endpoints to get the IP subnet information which is stored in `/coreos.com/network/config` file.

**Kubelet:** At this step, two processes are up and running inside the clustered environment. Now, it is time to execute rest of the processes in order to configure master nodes as Kubernetes masters. This is done by running kubelet process which is responsible for maintaining other processes and registering Kubernetes nodes with the cluster. This kubelet component starts the three most important processes called apiserver, controller manager, and scheduler, which are used by the master nodes to carry out container management. These processes are itself started as a separate Docker containers in order to ease cluster operation. This is done using a

framework called hyperkube[6] which is an all-in-one binary for starting Kubernetes master components. This framework allows us to select the specific process and then execute it according to the user requirements. In this implementation, three processes are started as a containerized application using this binary. Table 3.3 shows some important key parameters for starting the kubelet process.

Table 3.3: Kubelet parameters for running apiserver, controller manager, and scheduler on three master nodes

| Flag | On rpi-master-1 | On rpi-master-2 | On rpi-master-3 |
|---|---|---|---|
| -address | `0.0.0.0` | `0.0.0.0` | `0.0.0.0` |
| -enable-server | `true` | `true` | `true` |
| -api-servers | `http://192.168.137.155:8080,`<br>`http://192.168.137.134:8080,`<br>`http://192.168.137.236:8080` | `http://192.168.137.134:8080,`<br>`http://192.168.137.155:8080,`<br>`http://192.168.137.236:8080` | `http://192.168.137.236:8080,`<br>`http://192.168.137.155:8080,`<br>`http://192.168.137.134:8080` |
| -hostname-override | `192.168.137.155` | `192.168.137.134` | `192.168.137.236` |
| -cluster-dns | `10.0.0.10` | `10.0.0.10` | `10.0.0.10` |
| -cluster-domain | `cluster.local` | `cluster.local` | `cluster.local` |
| -config | `/etc/kubernetes/manifests` | `/etc/kubernetes/manifests` | `/etc/kubernetes/manifests` |

According to Table 3.3, the *-config* flag plays a vital role for starting key master node processes. This flag takes directory path as an input and starts monitoring that directory for files. For starting apiserver, controller manager, and scheduler, single file of format YAML or JSON is placed in that directory which then used by kubelet to run those processes as per the instructions written in that file. The IP address mentioned in *-address* flag is used by kubelet to serve requests on all interfaces both locally and externally (exposed to other nodes). The *-enable-server* flag allows master nodes to operate as server nodes by enabling kubelet server as well. This flag is set to 'true' by default. The *-api-servers* flag is the most important flag inside kubelet configuration, since it ensures that the server will be listened on the corresponding IP address and port number. This flag also enables kubelet to register master nodes with the Kubernetes cluster. In order to identify the node, kubelet uses *-hostname-override* flag which shows the correct server node. This flag is set by assigning the IP address of the respective master node. *-cluster-dns* flag assigns a DNS server to the cluster which is used by kubelet to configure all the containers and this DNS is used along with the host DNS server. Each running container search for the local cluster domain alongside DNS and that is configured by adjusting the value of *-cluster-domain* flag. There is another flag called *-pod_infra_container_image* which is specifically used on ARM processors and its basic functionality is to pro-

[6] `https://github.com/luxas/kubernetes-on-arm/tree/master/images/kubernetesonarm/hyperkube`

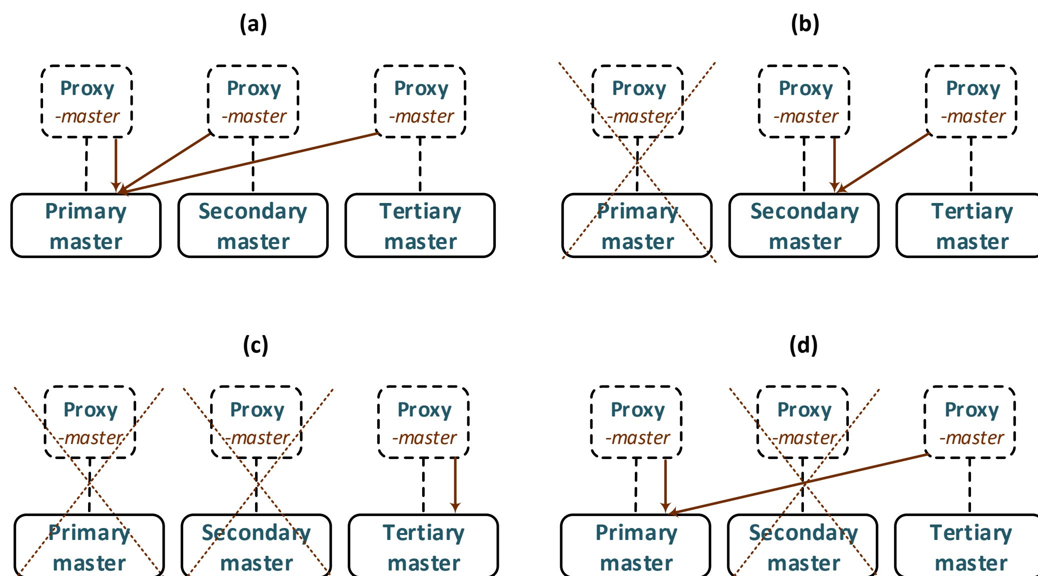


Figure 3.4: Setting of proxy flag on master nodes; (a) In case of a primary master node, (b) When the primary master fails, (c) When both primary and secondary masters fail, and (d) When the primary master comes back online

vide namespaces for containers inside each pod or replication controller. In this project, `gcr.io/google_containers/pause-arm:2.0` image is used for this purpose.

**Kubernetes Proxy:** The final step is to start proxy process on all three master nodes. Although proxy is the key component of Kubernetes worker node (as described in Section 2.2.1), it can also be executed on a master node to consider this node as a worker node. In our implementation, master node also acts as a worker node alongside other separate worker nodes. This proxy process is executed in a containerized environment, like other processes, which basically forwards service requests to the correct container. Same hyperkube binary is used to start this process and during startup, it requires IP address of the master node by setting *-master* flag option, which takes URL as a combination of both IP and port number. As it is already described that there are three master nodes in our cluster mechanism. One is the primary node which is also the first master node represented by hostname *rpi-master-1*, and the other two are the secondary and tertiary nodes which are represented as *rpi-master-2* and *rpi-master-3* respectively. Different criteria has been followed to set this flag value on all three master nodes and that is also illustrated in Figure 3.4.

For the primary master node, the flag is simply set by assigning IP address of the same node which is `http://192.168.137.155:8080`. In case of a secondary master node, two shell scripts are used to adjust that flag and these script files are shown in Appendix A.2. Each script uses Netcat (often abbreviated as nc) Unix utility tool

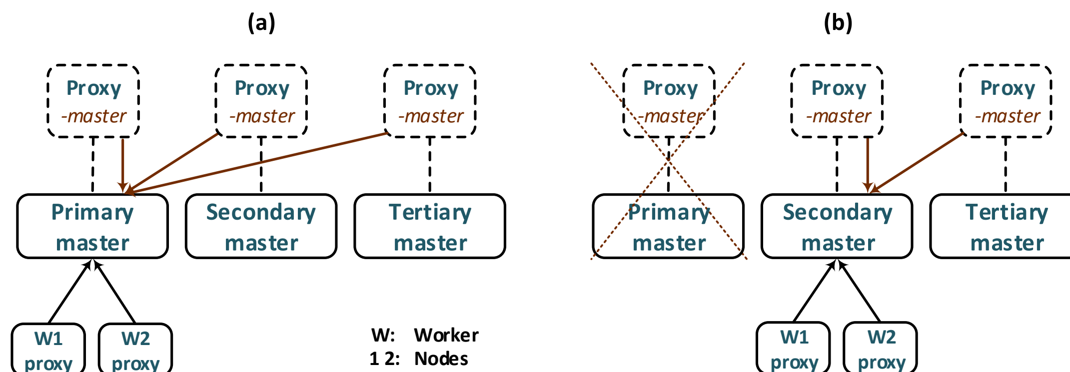


Figure 3.5: Proxy flag settings on worker nodes; (a) In case of a primary master node, (b) When the primary master fails

to continuously monitor both primary and secondary master nodes. It is a simple utility which reads and writes data by using TCP or UDP protocol across network connections. In our scripts, it is used by writing a combination "nc host port" which creates a TCP connection and checks the target host and port on which API server is running. If the network is disconnected or the master node shuts down, then it will change master node according to the availability of the API server. At first place, the flag value on secondary master is set to point to the primary master (as shown in Figure 3.4(a)). Whenever first master node is down, the second master becomes the primary master by updating the *-master* flag field and set its own IP address at that place. This in turn causes secondary master node to behave as a primary master and continue the execution of Kubernetes cluster. Similarly, for the tertiary master, four scripts are provided that are continuously monitors all three nodes. These scripts also utilize the same nc utility for checking API servers. Initially, its flag value is set to point to the first master node. That value is updated based on the next available master node in line. For example, if first two nodes are dead then the third node will become the primary master, as described in Figure 3.4(c). Once the first master comes back up again, the script changes the value of flag on the third master node to make it a tertiary master again.

## Kubernetes worker node installation

So far, three master nodes are up and running that can also operate as a worker nodes. In order to configure two more independent workers, three processes should be executed in specific order. The processes are flannel, kubelet, and proxy which are also created to run as a separate Docker containers. Appendix A.1 shows the service configuration files required for installing worker node components. Flannel and proxy processes are created and executed the same way as for the master nodes. Kubelet process also follows the same procedure and uses flag values given in Table 3.3 except *-config* and *-hostname-override* flags. For the *-config* flag, there is no need to install

other processes present in the 'manifests' directory, and thus no need to set that flag on worker nodes. The *-hostname-override* flag value is changed for each worker node because it is set based on the IP addresses of the two respective worker nodes.

Moreover, for the proxy process, the value of *-master* flag on both worker nodes is set to point to the primary master as done for secondary and tertiary master node. In case of a breakdown of the first master, all worker node proxies update their flag value by setting the IP address of the secondary master (as shown in Figure 3.5(b)). If, after a while, the secondary node also goes down than the worker proxies is set to point to the tertiary master. Once the primary master boots up again, all proxies are again point to that master node. This scenario is also illustrated in Figure 3.5 in which W stands for worker node. Four shell scripts inside each worker node are also used in this case to continuously monitor the three master nodes. This is also done using the same netcat utility tool to check the available API server host and port. These scripts are shown in Appendix A.2. In addition, one of the flags of kubelet named *-api-servers* is also updated in order to utilize the cluster correctly without any performance impact in terms of running containers.

# Chapter 4

# Data Pipelining

In the preceding chapter, a cluster of five RPi nodes are formed along with the configuration of Kubernetes framework. This shows the completion of our first implementation phase. This chapter describes the second phase by explaining sensor node configuration in terms of installing and utilizing different sensors. In this chapter, there are four sections in which Section 4.1 starts by discussing the type of sensors that are selected for this project. The section further explains about the motion detection feature that is associated with camera sensors. This section further proceeds by explaining the step-by-step procedure to configure Kafka cluster inside Docker containers and the deployment of those containers through the use of Kubernetes framework. Section 4.2 discusses the overall concept of data pipelining in the form of communication pipeline diagram. The producers and consumers applications are also demonstrated which utilizes the concept of Kafka topics to provide fault-tolerant system based on Kafka replication feature. Section 4.3 describes the overall procedure to send temperature data of all five RPi nodes. In the end, Section 4.4 puts some emphasis on important decisions that are made throughout the implementation process in order to design an effective system.

## 4.1 IoT Sensor Node

For designing an IoT sensor node, countless number of sensors are available that can sense the environment and generate data for computing. The focus of this thesis is to select two sensing methods that can sense real-time data and then process it in order to operate as a sensor node for IoT. The first method is to use camera sensors for each RPi which can record the data based on external conditions. In the second method, the temperature condition of each RPi processor is recorded and then sent to the cloud platform. Sections 4.1.1, 4.1.2, and 4.1.3 explains the first method which uses camera sensors. The temperature sensing method is described in Section 4.3. In our implementation, camera module version 2 has been selected as a first sensor that is attached to the CSI port of each RPi board. The main reason of using this camera is to provide an efficient system as this module is fully compatible with Raspberry Pi 2. The camera has 5M pixels resolution with the capability to capture 1080p video and still images, which is sufficient in our implementation.

Table 4.1: Parameters for configuring camera module

| Configuration parameter | Value |
|---|---|
| start_x | 1 |
| gpu_mem | 128 |
| hdmi_force_hotplug | 1 |

At this point, a cluster of five RPi nodes is completed which is running Kubernetes framework with cameras attached to each node. The next step is to install and configure the camera module so that it can be used to capture images/videos. This is done by editing a file named `config.txt` which is located in the `/boot` partition of the Hypriot OS. As there is no conventional BIOS in a RPi, various system configuration parameters are stored in that file instead of setting them in BIOS for a typical computer system. This text file is itself stored in SD card and whenever a RPi boots up, it reads those parameters from `/boot/config.txt` in order to perform additional changes. The same procedure is followed for each RPi in the cluster. Table 4.1 shows some basic configuration parameters for setting up camera modules.

As can be seen from Table 4.1, first parameter *start_x* enables the camera module by setting its value to ‘1’. *gpu_mem* sets the memory split between ARM and GPU and uses minimum memory for the camera to operate properly. Its value is in megabytes. The last flag *hdmi_force_hotplug* enables HDMI display so that the camera output can be seen on a monitor display. After setting these parameters, it is time to restart all RPi boards in order to use the camera sensor. As an optional measure, it is also suggested to update the firmware software, as it will install all the necessary drivers that are useful to run camera module.

### 4.1.1 Configuring motion detection

The next step is to configure motion detection alongside camera module, as the idea of using cameras is to detect motion from the video stream. This is done by configuring a ‘motion’ feature with all five cameras that are already attached to each RPi. The instructions from wiki webpage[1] are used to install that feature package. This package is a special binary compiled by the RPi community which also requires various dependencies to be installed first in order to execute motion. After installing all the required dependencies, it is time to install that motion package. The file named `motion-mmal-lowflyerUK` is used which is provided by lowflyer. In addition, it also requires some configuration parameters that can be adjusted and used with the motion binary. These parameters are stored in `motion-mmalcam.conf` file which dictates the overall operation of camera module with the corresponding

[1] `http://wiki.raspberrytorte.com/index.php?title=Motion_MMAL`

Table 4.2: Parameters for defining motion detection

| Configuration parameter | Value |
|---|---|
| daemon | on |
| width | 640 |
| height | 480 |
| framerate | 10 |
| mmalcam_use_still | on |
| threshold | 2000 |
| pre_capture | 2 |
| post_capture | 2 |
| output_pictures | best |
| target_dir | /mnt/cameraN, where N is 1,2,3,4,5 |

output. Table 4.2 shows the key configuration parameters that are used to detect motion.

According to Table 4.2, the *daemon* parameter is set to 'on' as it allows the motion process to start in the background and release the terminal. The *width* and *height* are the image resolution in pixels that is used by the camera to capture motion detected images. For our project, these parameters are set to integer numbers 640 and 480 for width and height respectively. The *framerate* parameter shows the maximum number of image frames to be taken per second. It is set to value 10 but any number can be assigned as there is no limit but for performance efficiency, it is better to select a number between 2 to 100. Since the idea was to capture still images instead of a video stream, *mmalcam_use_still* is turned on which enables motion to capture and store still images. Another parameter called *threshold* is the most important parameter, since it defines the number of changed pixels in an image that will trigger the motion. It is given a value 2000, just for now, so that it will trigger the smallest possible motion. In order to add consistency for the detected image, *pre_capture* is defined which captures and buffers pictures slightly before the time of motion detection. Similarly, *post_capture* is also defined to capture frames after the motion is no longer detected. The *output_pictures* parameter enables motion to store normal pictures whenever motion is detected. The frequency of those pictures are adjusted based on the value assigned to this parameter which is 'best' in our case. This causes the motion to save those pictures which contains most changed pixels. Finally, the *target_dir* sets the directory in which pictures will be stored. Since there are five cameras, the directories are different for each RPi and these are */mnt/camera1, /mnt/camera2, /mnt/camera3, /mnt/camera4,* and */mnt/camera5.*

### 4.1.2 Setting up Kafka cluster

At this point, one stage of IoT sensor node is almost completed which contains camera sensors with motion detection activated on them. The next stage is to send pictures from local cluster to the cloud platform for further processing and storage. For this purpose, Apache Kafka messaging system is selected which is already described in Section 2.3.1. This section explains the step-by-step procedure of configuring Kafka inside our local cluster running Kubernetes framework. As it is known that Kafka has a critical dependency on Apache Zookeeper, it is necessary to run Zookeeper service first before installing Kafka as a clustered system. Zookeeper provides coordination and synchronization within different nodes of the cluster and it is described in detail in Section 2.3.2. Since our implementation is based on container-based virtualization, Docker containers are utilized to build images for both processes (Kafka and Zookeeper) in order to achieve better performance.

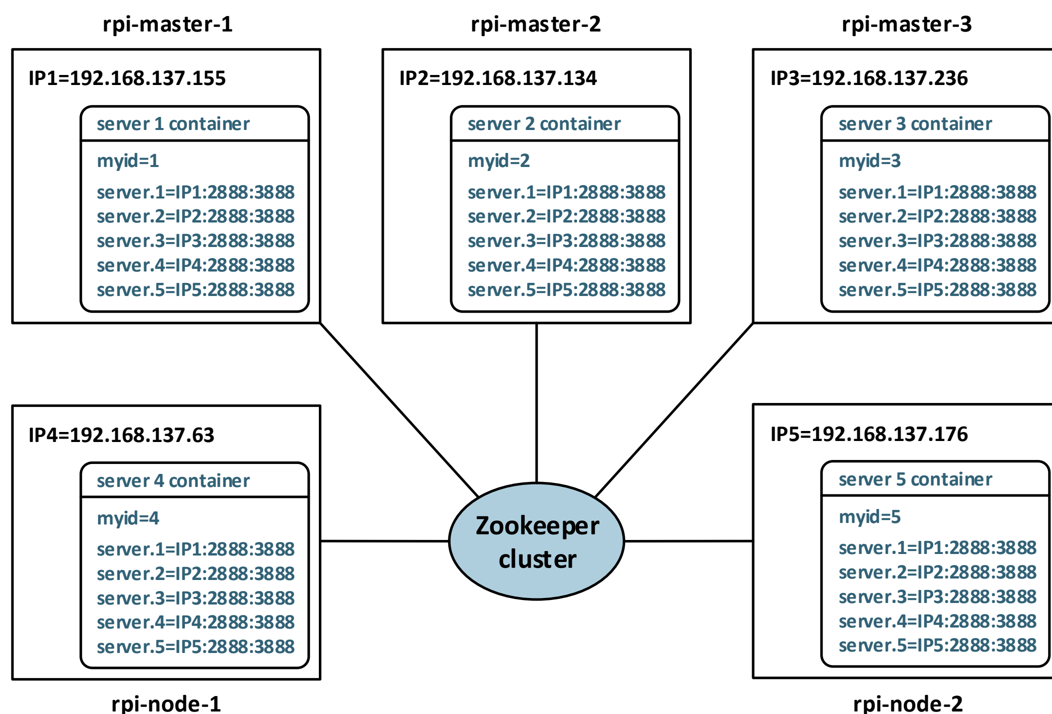


Figure 4.1: Graphical representation of Zookeeper cluster

In order to provide a reliable service, Zookeeper is deployed in a cluster known as an ensemble. For the service to be available continuously, majority of machines in an ensemble should be up and running. This means that the Zookeeper cluster requires majority, so it is suggested to use an odd number of servers. As an example, Zookeeper can handle one machine failure in the cluster of three machines. If two machines are down, the cluster will go down. For instance, there are five nodes in our implementation on which Zookeeper container will run in each node, so it can

handle two machine failures.

Figure 4.1 illustrates the overall mechanism of setting up a Zookeeper server inside the containers. Since there are five RPi in our cluster, a five server Zookeeper ensemble is configured which runs in a containerized environment. The Zookeeper version 3.4.8 is used for this purpose which is considered as a stable release and it can be downloaded from the official website of Apache Zookeeper. Following are some of the steps that are carried out inside the container of each RPi to set a Zookeeper cluster.

Table 4.3: Parameters for configuring Zookeeper cluster

| **Configuration parameter** | **Value** |
|---|---|
| clientPort | 2181 |
| tickTime | 3000 |
| initLimit | 10 |
| syncLimit | 5 |

- The first step is to install Java JDK, since Zookeeper uses its native packaging to run server inside the container.
- Then, a file named `zoo.cfg` is created which stores basic configuration parameters. Some of them are listed in Table 4.3 along with *dataDir =/var/zookeeper/data* and series of five lines of the form *server.x=hostIP:port:port* as shown in Figure 4.1. These *server.x* lines are defined inside each container in order to know the members in an ensemble. In addition, *myid* file is also created in the *dataDir* of each Zookeeper server which only stores a unique integer number and nothing else. That number should match the *x* part of *server.x* parameter which will later use in Kafka to perform leader election. In our sensor node, id's 1,2,3,4, and 5 are selected for *x* and *myid* as well. As an example, the value '1' is stored for the first container running inside rpi-master-1 by using the following command.

```
echo "1" > /var/zookeeper/data/myid
```

- Finally, after configuring all the parameters, it is time to start Zookeeper server. This is done by executing the following command inside zookeeper directory which is already downloaded as a complete package containing Zookeeper scripts.

```
bin/zkServer.sh start
```

At this stage, a Zookeeper ensemble is running based on the parameters shown in Table 4.3. As can be seen, *clientPort* enables client to connect to the Zookeeper

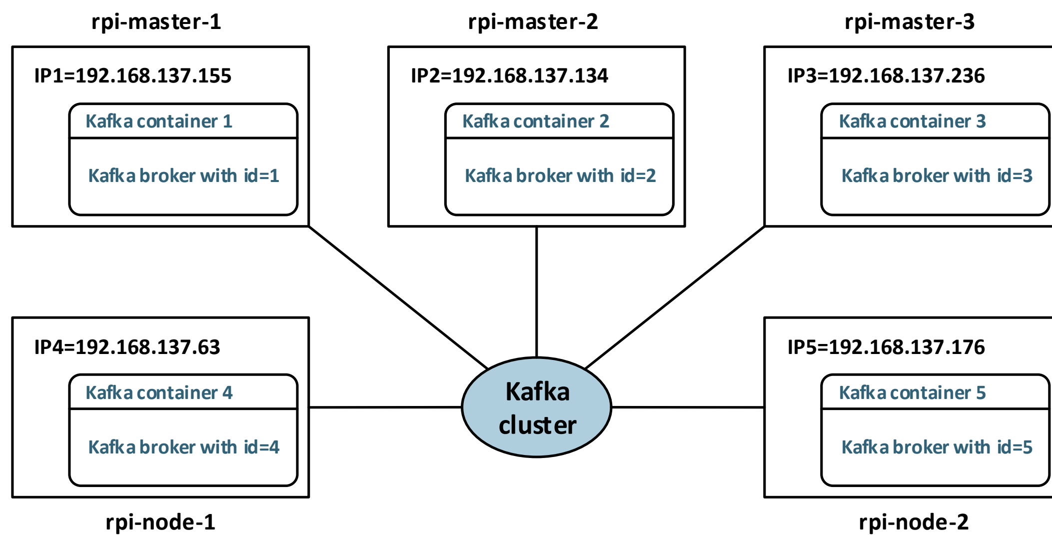


Figure 4.2: Graphical representation of Kafka cluster

server on port 2181. The *tickTime* parameter shows the minimum session timeout (in milliseconds) used by Zookeeper to regulate heart beats and timeouts. Its value is 3000 meaning that the session will expire after three seconds. The *initLimit* allows followers to connect and synchronize with the leader and this time is given in ticks. Finally, *syncLimit* is the amount of time in ticks used by the followers to synchronize with the leader.

Since Zookeeper is configured to run as a cluster of five containers, it is time to configure Kafka cluster inside separate containers as well. This is done by downloading the Kafka package version 0.9.0 from the official website that contains some scripts and configuration files. In order to provide a fault tolerant system, five brokers are configured each runs on different RPi node, which then act as a cluster. Kafka also uses various configuration parameters that are stored in `server.properties` file. Table 4.4 shows some of the key parameters that have to be defined for each broker. The *broker.id* must be set to a unique integer for each broker and it should also match with the Zookeeper service id's for the corresponding servers. The *port* defines a server socket for listening and its default value is 9092. *host.name* parameter defines the IP address that will bind to each broker server. That address corresponds to the IP address of each Kafka container. In order to provide communication, the broker will use *advertised.host.name* to advertise hostname with producers and consumers. Similarly, the port given in *advertised.port* parameter is publish to Zookeeper, so that clients can use it and its value is also 9092. There is a last parameter called *zookeeper.connect* which is a comma separated list of hostIP:port pairs. These pairs define the address of Zookeeper servers. In our cluster, it is assigned a string IP1:2181,IP2:2181,IP3:2181,IP4:2181,IP5:2181, where 2181 is a port used by clients to connect to the Zookeeper server.

Moreover, a graphical representation of Kafka cluster is shown in Figure 4.2. As

Table 4.4: Parameters for Kafka broker

| Configuration parameters |
| --- |
| broker.id |
| port |
| host.name |
| advertised.host.name |
| advertised.port |
| zookeeper.connect |

can be seen, there are five distinct containers that runs on five separate nodes. These containers are also considered as a brokers with unique broker id. Zookeeper is also running inside five separate containers adjacent to each Kafka container as discussed in the previous paragraphs. Through this way, the sensor node is ready to produce some data which later send to the cloud for processing.

### 4.1.3 Containers deployment

So far, the Kafka cluster of five containers is already configured along with the Zookeeper cluster. These containers have to be deployed in order to execute Kafka. This is done by utilizing Kubernetes framework. As Kafka is dependent on Zookeeper, both of them needs to be executed together. For this purpose, a work unit called pod is used which allows Kubernetes to run and manage bulk of containers inside a single pod. It is already described under Kubernetes work units heading of Section 2.2.1. Since there are five RPi nodes in our implementation, five pods are used in which each pod contains two containers that will configure Kafka cluster. These containers are considered as front-end (Kafka) and back-end (Zookeeper) containers. Through pod, the containers are executed side-by-side on a same node and managed as a single application.

In order to execute a pod, a configuration file formatted as YAML is created which is used by Kubernetes `kubectl` command to run those containers. For running Kafka cluster, five files are created with few minor changes. Each file stores user-specified instructions based on REST APIs version 1[2] parameters. Some of the important fields are listed in Table 4.5. Rest of the parameters are shown in each YAML file (for five nodes) which is presented in Appendix B.1.

As can be seen from Table 4.5, a `kind` field defines the type of a pod which is 'ReplicationController' in our case. It ensures that the specified number of pod replicas are running at one single time which is controlled by the `replicas` field. The main reason of using replication controller is to allow Kubernetes to continuously

[2] `http://kubernetes.io/docs/user-guide/replication-controller/operations/`

Table 4.5: Important fields of YAML file

| **Fields** |
| --- |
| kind |
| nodeSelector |
| volumes |
| volumeMounts |
| replicas |

monitor the active pods and start their replica again whenever the current pod dies. A `nodeSelector` field allows pods to execute on a dedicated node and it is set by assigning the names of each RPi which are *rpi-master-1, rpi-master-2, rpi-master-3, rpi-node-1,* and *rpi-node-2*. This flag is necessary because pods get data from the camera sensors and those cameras are fixed for each RPi node, thats why they have to be executed on a specified node. The `volumes` and `volumeMounts` are used to mount data between host machine and the container. As it is described in Section 4.1.1 that a target directory of the host machine is associated with each camera in which motion pictures are stored. The `volumes` field uses that directory which is mounted with a directory mentioned in `volumeMounts` field inside each container. For our implementation, these are */mnt/camera1, /mnt/camera2, /mnt/camera3, /mnt/camera4, /mnt/camera5* for RPi nodes and */camera1, /camera2, /camera3, /camera4, /camera5* for Kafka containers. All these directories are also mentioned in the YAML configuration files.

## 4.2 Data Processing and Simulation

At this point, the IoT sensor node is configured and ready to operate. Now is the time to use our sensor node for gathering data and transferring it to the cloud for processing and storage. This section describes the step-by-step procedure to provide communication and send data from the local Kafka cluster to the remote Kafka cluster inside the cloud platform. Figure 4.3 demonstrates the overall concept of data communication by utilizing Kafka messaging system. Separate Kafka producers and consumers based on Java programming are used for this purpose to produce and consume data from sensors. For our implementation, there are five producers, each running inside separate Kafka container and the code can be found in Appendix B.2. Each producer code is in the form of a jar file which is placed inside the same script file that is used to configure Kafka container. Following are some of the steps that corresponds to a data pipelining inside our cluster. These steps are carried out for each Kafka container.

- At first, the producer creates a local topic named `node1` for camera1 called

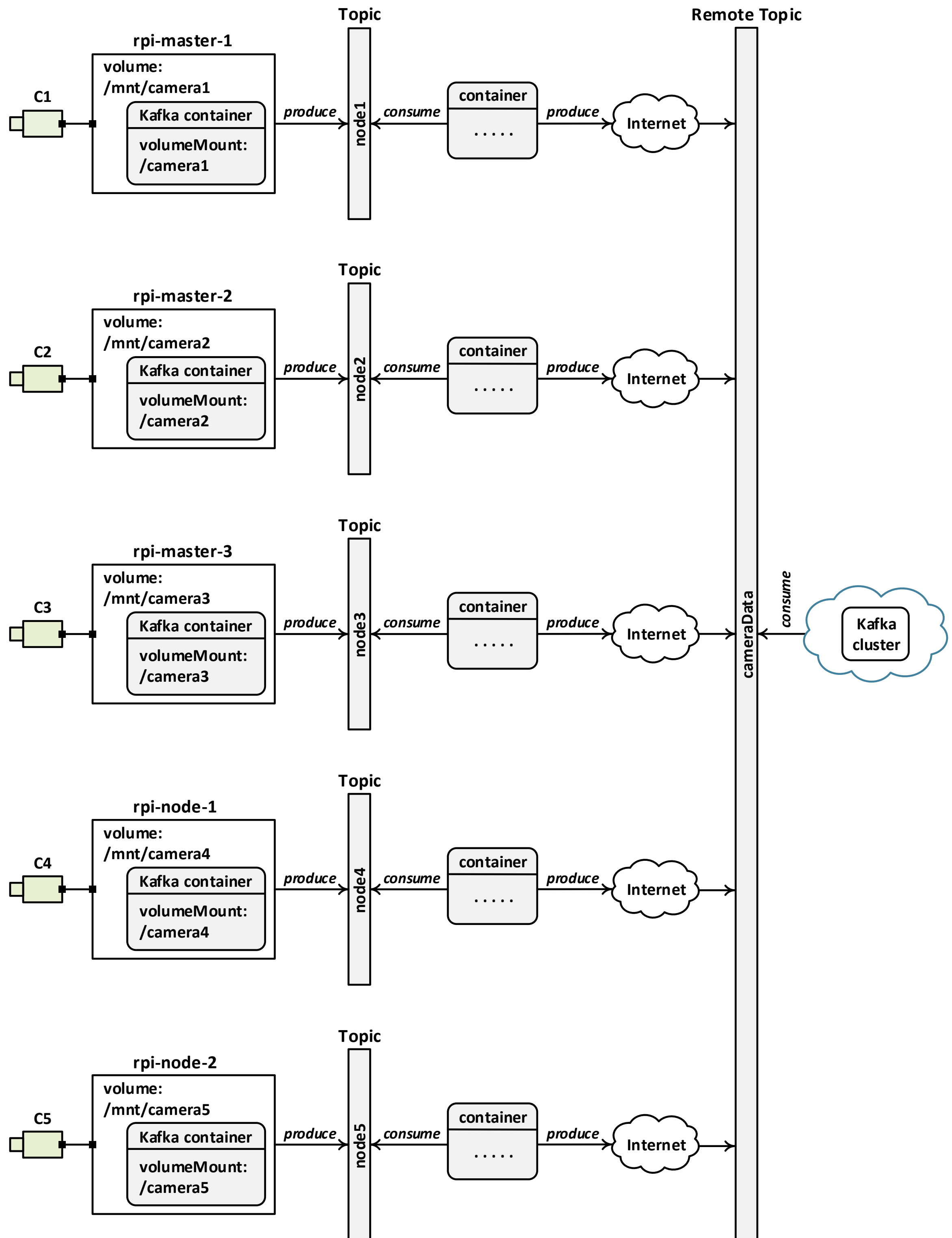


Figure 4.3: Data pipelining based on Kafka topics

C1, as shown in Figure 4.3. Similarly, for other cameras C2, C3, C4, and C5, the topics are `node2, node3, node4,` and `node5` respectively which are created by other producers on a dedicated node. In addition, a replication factor of value '3' is also selected during topic creation. This value provides fault-tolerance behavior by replicating data on multiple nodes of the cluster. This causes the consumer application to consume data from some other node in case of a node failure.

- Secondly, the producer continuously monitors container directory for motion pictures. Whenever there is an image, it is converted into Base64[3] string so that Kafka can easily send that string to the topic. Base64 format provides basic encoding and decoding for an image by utilizing Java APIs.
- After conversion, the image string is then stored in JSON format along with the camera id, date, and time of that particular image. It looks like a following string after JSON formatting.

```
{id:<camera id>,date:YYYYMMDD,time:HHMMSS,value:<Base64 string>}
```

- Finally, each producer then produces the above mentioned string to the respective Kafka topics so that consumer application can consume from those topics. In addition, that particular image is deleted from the local directory after it is successfully sent to the topic.

After producing images of all five cameras, they are then consumed on any of the five containers because of the replication factor of Kafka. The data is consumed inside a different container which runs on any of the nodes. This container is created using the replication controller of Kubernetes without setting a `nodeSelector` field value. Through this way, the Kubernetes schedules that container (as a pod) on any of the active RPi nodes. The container between local topic and remote topic in Figure 4.3 depicts this behavior. It shows that the container runs on any arbitrary node for consuming purpose. In addition, it continuously monitors this pod and automatically starts it again whenever the current pod dies. A consumer application written in Java is used which automatically starts after the pod has been deployed. The application contains both consumer and producer. At first, the consumer subscribes to the local topics and then starts consuming from them. It then uses a remote topic named `cameraData` to produce the same data (as shown in Figure 4.3). This topic is already created inside the cloud. The consumer code is also shown in Appendix B.2. Moreover, there is a streaming application that runs on the cloud side which continuously checks the Kafka brokers for new string. If the data is available, it is consumed from the same topic `cameraData` and then stored in HDFS permanently for further processing.

[3] `https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/binary/Base64.html`

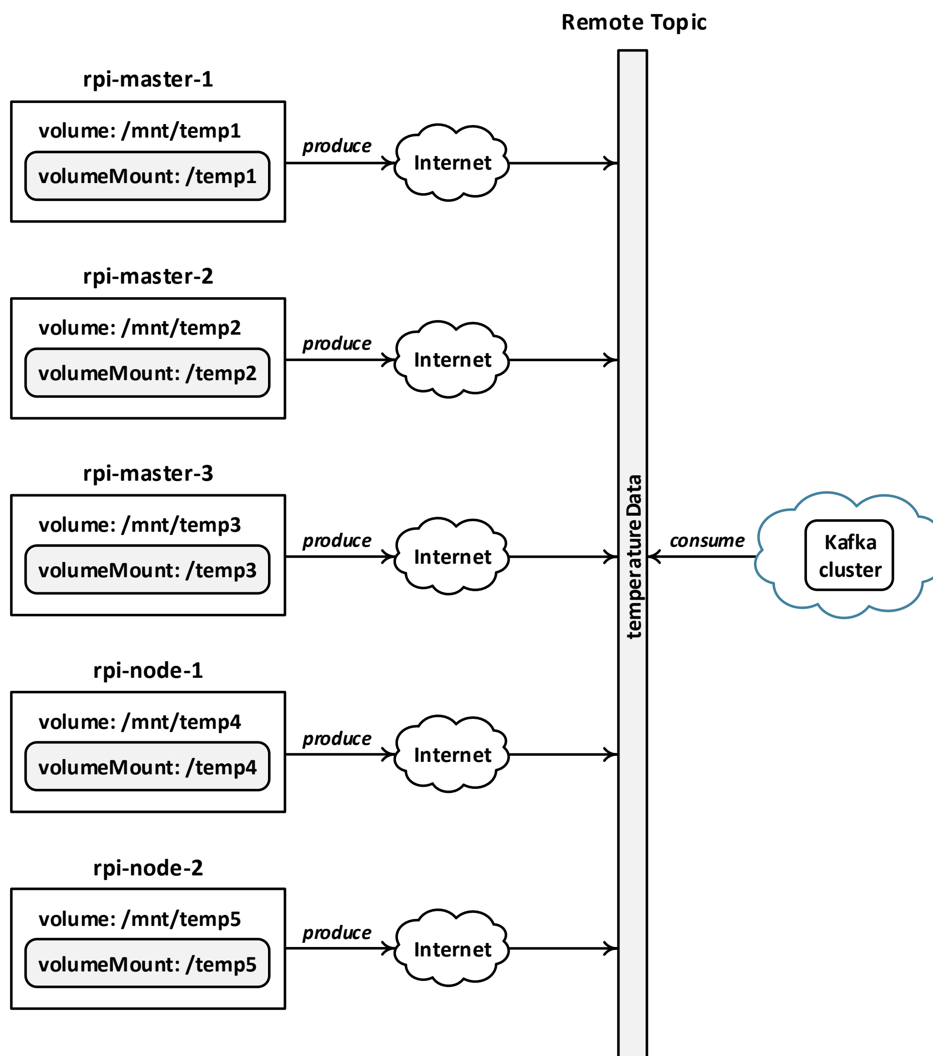


Figure 4.4: Temperature data pipelining

## 4.3 Containers for temperature data

As it is already mentioned that two sensing methods had been selected for implementing IoT sensor node. The camera sensor method is explained in the previous section. This section focuses towards the temperature sensor method by using the internal temperature of each RPi processor. This is done by utilizing a userspace tool called `"vcgencmd"` that can access various system information specific to the RPi board including the amount of memory, clock frequencies, CPU temperature, hardware codecs, and component voltages. In order to read CPU temperature, the keyword `measure_temp` is used with the tool and the overall command is `/opt/vc/bin/vcgencmd measure_temp`. A small script in each RPi is used to continuously read temperature and the values are stored in a directory which

is mounted with the directory inside containers. The directories are */mnt/temp1, /mnt/temp2, /mnt/temp3, /mnt/temp4, /mnt/temp5* for five RPi nodes and */temp1, /temp2, /temp3, /temp4, /temp5* for the containers, as shown in Figure 4.4. Kubernetes replication controller is also used here to execute containers and then read the temperature data for each RPi. Once the data has been stored, it is then sent to the Kafka topic named `temperatureData` which is already present inside Kafka cluster on the cloud platform. On the cloud side, same streaming application is used to consume data from the topic and then store it in HDFS permanently.

## 4.4 Design Decisions

This section laid emphasis on some important decisions that are made during the implementation phase in order to achieve an efficient IoT sensor node. The first design parameter is related to Kubernetes framework. At first, there was just one master node in our Kubernetes cluster and the whole sensor node was dependent on this single master. Later on, it is decided to provide highly available cluster by configuring three RPi nodes as master nodes. This causes the clustered system to tolerate two nodes failure. This high availability feature is achieved by utilizing an etcd data-store which is one of the processes of Kubernetes master node. The key functionality of etcd is to store and share cluster configuration across different nodes and it is configured as a cluster of three member nodes. Through this way, the cluster is considered as a highly available Kubernetes cluster.

The second decision corresponds to the Kafka server which uses Java Runtime environment to use heap memory. As it is known that each Raspberry Pi has only 1 GB RAM, the heap memory is adjusted in such a way that Kafka process uses less than 1 GB memory. It is updated by editing `kafka-server-start.sh` script file. This file contains a command in which two options `-Xmx` and `-Xms` are used to adjust the heap size. These parameters are adjusted to use half of the memory (512 MB) for Kafka. For instance, it is updated according to the values `-Xmx512M` and `-Xms512M`. Through this way, Kafka runs efficiently across a cluster without any hindrance.

Finally, the last decision is related to the replication factor of Kafka topic. It is decided to replicate each sensor data three times inside a cluster. This is done at the time of creating a topic. This allows Kafka consumer to consume data from any of the five nodes. Thus, providing fault-tolerant system by replicating data across a cluster.

# Chapter 5

# Evaluation

In this chapter, the proposed implementation is evaluated based on multiple parameters. The focus will be more towards the development and deployment process in terms of some operational characteristics including architecture, functionality, performance, high availability, fault tolerance, and scalability. The testing is carried out in an advanced way in which every small step is taken into consideration ranging from the RPi board to the camera module, from Kubernetes pod to the process running inside Docker containers, and from the local cluster to the cloud platform. This shows the overall functionality of the development process for end-users and system administrators.

## 5.1 Architecture

The sensor node was first evaluated on the basis of its overall architecture. It was seen that all the requirements were met especially related to the hardware components. According to the architecture, there are five RPi nodes in which three of them operate as a Kubernetes master nodes and two nodes are the worker nodes. Each node has a 1 GB RAM that is sufficient to perform the required tasks. Each node is connected to a camera module version 2 which is highly compatible with RPi 2 boards. The camera functionality was also tested based on the image sizes and the frequency of the motion detected pictures and the results were quite impressive. Moreover, the Ethernet cable of Category 5 is used to assemble a cluster that showed a great performance, as it can handle 10/100 Mbps speeds at up to 100 MHz bandwidth. This shows that the hardware components are feasible in terms of cost and processing power for this thesis project.

## 5.2 Performance

The cluster performance (considered as software performance) is tested by doing various experiments on the processes running inside each RPi node and the containers on that node. The experiments include creating and deleting processes, connecting and disconnecting network availability, and powering off and on the nodes. This

Table 5.1: Processing time of the images that are sent to the local topic

| | **Image 320x188** | **Image 640x480** | **Image 1280x720** |
|---|---|---|---|
| **Processing Time (ms)** | 23 | 99 | 371 |
| | 15 | 68 | 283 |
| | 14 | 73 | 407 |
| | 15 | 94 | 362 |
| | 24 | 65 | 302 |
| | 19 | 74 | 299 |
| | 16 | 74 | 342 |
| | 21 | 71 | 396 |
| | 22 | 68 | 367 |
| | 19 | 80 | 319 |
| **Average** | 18.8 | 76.6 | 344.8 |

testing is divided in terms of Kubernetes replication controller, Kafka producers, and Kafka consumers. For the replication controller, it was observed that this rc is the good choice to create and deploy containers as one or more pods. It was then tested by deleting a pod that was associated with that replication controller. Once the current pod had been deleted, the rc had again started a new pod on the same dedicated node. Similarly, it was also checked by shutting down one of the cluster nodes and the behavior was still same. In both cases, the pod was started again by the replication controller that was then combined with the rest of the cluster to operate as it was operating initially.

The next experiment was done on Java producers and consumers applications that are running inside all five containers. It was first tested by disconnecting the network cable of one of the node from the cluster and then observed the behavior. It was seen that the application are not sending further data to the topic but after sometime, the cable was plugged again and the Java producer started again without any additional setup. Similarly, same was the case with consumer application. Then the applications are tested by shutting down the node on which that particular producer or consumer was running. It was observed that when the node booted up again, the Java application was automatically started because of the replication controller of Kubernetes.

Finally, the performance was tested on the basis of different image sizes. Table 5.1 shows the processing time of different images in case of a local producer application for just one node. This processing time includes the conversion of image into base64 string, then transforming the string into JSON format, and then send to the local topic. It can be seen that the average time increases with the increase in image size.

Table 5.2: Processing time of the images that are sent to the remote topic

| | Image 320x188 | Image 640x480 | Image 1280x720 |
|---|---|---|---|
| Processing Time (ms) | 7 | 32 | 61 |
| | 4 | 36 | 76 |
| | 4 | 16 | 56 |
| | 5 | 24 | 50 |
| | 5 | 16 | 45 |
| | 4 | 22 | 76 |
| | 4 | 18 | 50 |
| | 4 | 15 | 55 |
| | 4 | 16 | 50 |
| | 5 | 15 | 60 |
| **Average** | 4.6 | 21 | 57.9 |

In this project, the image of size 640x480 was captured from the sensors which shows the average processing time of 76.6 ms. This image size is reasonable for our project, as it is sufficient to capture motion in that resolution along with less processing time. If the size of image is greater than 1280x720 than it will take more processing power and sometimes, Kafka gives the error of the buffer size. On the other hand, Table 5.2 illustrates the processing time of different images for the consumer application that first consume from local topics and then produce to the remote topic. This time also includes the conversion of image into base64 string together with transforming that string into JSON formatted string and then send to the remote topic. The average processing time of 21 ms for 640x480 images shows the reasonable amount of time for these images.

## 5.3 Fault tolerance

Fault-tolerance behavior was tested on the basis of three processes that was running inside the cluster. These are etcd data-store on three master nodes, Zookeeper ensemble, and Kafka brokers. The first testing was done on etcd cluster itself. As etcd is configured to operate as a three node cluster, it can tolerate one RPi node failure. The whole process was tested thoroughly and it was evaluated that if two master nodes are dead, the cluster still operate but it was not possible to deploy more pods. As an example, the primary master was first switched down and then observed the cluster behavior. Everything was working fine at that point, then after sometime it was decided to shut down the secondary master. Although the pods were running on third master and the worker nodes, it was still not possible to execute

any other pods. Once the cluster had been tested with one master node (tertiary node), another master was started again. At that point, the cluster had two master nodes again but it was not operated as expected until the etcd process had been restarted again on both available nodes. All these observations are also recorded in Table 5.3 on the basis of different cases. The 'up' and 'down' words correspond to the active etcd member on a respective master nodes. This showed that the IoT sensor node can tolerate one machine failure in best case and two machines in worst case scenario. Moreover, the testing also depicts the high availability behavior of Kubernetes cluster in terms of master nodes.

Table 5.3: Fault-tolerance behavior based on etcd data store

| **Case** | **Primary master** | **Secondary master** | **Tertiary master** | **Observations** |
|---|---|---|---|---|
| **1** | up | up | up | This was the default condition in which all three nodes were active and etcd was running. |
| **2** | down | up | up | The secondary master became primary master and the cluster was operated as before. This showed that the sensor node tolerated one node failure. |
| **3** | down | down | up | The tertiary master became the primary master and the processes was working on the worker nodes as before but it was not possible to execute more pods or managed already executed pods. |
| **4** | down | down | down | This is the worst case scenerio in which the processes still executed on the worker nodes but there was no control over those process from the master nodes. |
| **5** | down | up | up | In that case, the etcd cluster was restarted on both nodes in order to operate as Kubernetes masters again. |

Secondly, the performance was evaluated based on the Zookeeper that was running inside five separate containers. Since Zookeeper cluster needs majority, it means that at least three containers are up and running out of five containers. In that case, two of them are the follower nodes and the other node is the leader node. This was also observed by performing different experiments. At first, the network connectivity was disconnected and then tested the cluster. It was observed that the cluster was

operating as expected. There was an interesting thing as well. After sometime, it was decided to disconnect one more node so that the two containers were running Zookeeper. In this scenario, the cluster was working but the Zookeeper cluster was not serving requests. When the network had been started again for any of the nodes (meaning that there were now three Zookeeper containers up and running again), the Zookeeper was started serving requests as before.

Finally, Kafka brokers were also tested by doing various experiments. These experiments were performed by first disconnecting the Internet and then shutting down the RPi node itself. It was observed that three containers should be up and running in order to operate properly because of the replication factor of value three. In case of one more container failure (means three out of two containers are down), the cluster got malfunctioned and it was needed to start the cluster again in order to operate as a sensor node.

## 5.4 Scalability

The IoT sensor node is scalable in such a way that more nodes can be attached to the current implementation and these nodes are configured to operate either Kubernetes master or worker nodes. The process is really simple, you just have to install some dedicated components as a containerized applications on a RPi and added it to the current cluster. In addition, the Kafka cluster is also scalable since it runs inside Docker containers and it is quite easy to update configuration files for Kafka inside a single container and build a new image to use for the updated cluster. That new container image can be downloaded from Docker Hub and then used for configuring a new cluster with an additional node.

# Chapter 6

# Summary and Conclusion

With the rapid advancement of technology, automated devices are gaining popularity in the field of Internet of Things and embedded systems. The ability of these devices to connect and share resources via Internet is becoming more pervasive. Although the devices generate huge volumes of data, it is necessary to process it in an effective way and extract the useful information for further processing and storage. As the data is growing rapidly, a single server is not sufficient to provide a required computation power. Virtualization is one of the possible way to achieve high processing power, as it has become a promising paradigm in IoT.

There are two approaches in Virtualization paradigm. Hypervisor-based virtualization is a type of virtualization that creates and manages virtual machines on top of a host operating system. It provides an independent virtual environment in which applications can run in an isolated manner. On the other hand, container-based virtualization is an alternative to hypervisor-based virtualization. It is an operating system-level virtualization that creates virtual environment in the form of containers at a software level. This causes a container to share the underlying resources of the host operating system instead of creating their own resources, as in the case of hypervisors. In addition, kernel is also shared between the host system and the container, and thus providing an efficient system without the overhead of hypervisor software. Moreover, in terms of embedded devices, container-based virtualization is efficient to use and the traditional virtualization is difficult as the devices have small RAM. Although there are various techniques to use container-based system, Docker containers is one of the possible way that is also the core concept of this thesis project. It is an open source platform that can build, deploy, and run applications (inside containers) faster as compared to other traditional platforms. In addition, it provides security and isolation to the containers by utilizing the two important kernel features called *control groups* and *namespaces*. Through this way, it is easy to run multiple containers simultaneously on a single host machine without any interference.

Moreover, one of the good thing about containers is that they can be managed as a cluster of encapsulated applications. Various systems have been developed for this purpose but the main focus of this thesis is to utilize Kubernetes framework that is developed by Google. It is an open source cluster manager for containerized applications across a number of hosts that are distributed either virtually or

physically. With Kubernetes, you can effectively deploy multiple containers, scale those containers, and roll out new resources. In other words, it can be said that Kubernetes is portable, extensible, self-healing, and fault-tolerant system. In order to deploy multiple containers, Kubernetes uses its work unit called pod which is created and then executed automatically on some particular host machine. The pod contains at least one container. Apart from this, there is another work unit called replication controller which also creates and monitors pods and whenever there is a system failure, it will automatically start the new pod on same or different host, depending upon the availability of the host machine.

In this thesis, the implementation of IoT sensor node is proposed and tested based on state-of-the-art technologies. The focus was to utilize container-based virtualization called Docker containers and container orchestration framework called Kubernetes on top of a Raspberry Pi cluster mechanism. The cluster is formed using five RPi nodes that are connected to a central hub called network switch. All the devices are connected through Ethernet with the switch and the other end of that switch is connected to the computer server so that the clustered system can communicate with the outside world. In addition, five camera modules containing motion detection feature are also attached to each RPi node in order to operate as a sensor node. Whenever there is a motion in front of a camera sensor, the picture is first captured and then sent to the cloud platform through the use of a well-known messaging system called Apache Kafka. It uses the concept of topics to publish and subscribe messages. In addition, Java APIs are used to write producer and consumer applications that perform the necessary intermediate data pipelining. This Kafka is also running as a cluster of five Docker containers. Kubernetes replication controller is used to create and deploy those containers in the form of a pods. On the other hand, the cloud server is also integrated with Kafka and HDFS for further processing and storage.

There are few observations that conclude our motivation behind this project. First observation is related to the Kafka framework. The idea of using Kafka was to provide fault tolerance by tolerating Internet connection loss. This means that if there is no communication between the local Kafka cluster and the remote cluster then the data will be stored inside the topics of the brokers. In addition, the data was also replicated on different brokers because of the replication factor which was 3. Moreover, this Kafka also showed fault tolerance in which three brokers should be up and running in our case. In case of four dead brokers, the cluster was malfunctioned and it had to be started again. This fault tolerance was also observed in case of three Kubernetes master nodes on which etcd cluster was running. Due to this etcd, the whole sensor node even tolerated two master nodes in worst case. Our second motivation was related to edge computing which was also achieved using Kafka. With this framework, it was easy to choose data processing location in terms of local or remote cluster, regardless of the underlying architecture.

In the end, it is concluded that this sensor node is feasible in terms of fault-tolerance and high availability. The simulation results and the project evaluation illustrate that the overall system is efficient and the processing time is also minimal through the use of a container-based technology. Rest of the chapter is organized

into two sections. Section 6.1 describes some of the limitations that are faced during our project implementation. Section 6.2 discusses some of the future work that can be done using our current sensor node, specially regarding other sensors or increasing the number of RPi nodes.

## 6.1 Limitations

The first limitation is related to Raspberry Pi itself, as it has 1 GB of RAM. Due to this small memory, it was challenging to run multiple processes at first place, specially regarding Kafka cluster. Because of the memory constraint, Kafka was not properly started as the limit of heap memory for Kafka server was initialized in such a way that it will use 1 GB memory. Then it is adjusted so that Kafka can use half of the memory.

Another limitation is related to the version of Apache Kafka, At first, there was a compatibility issue between the versions of Apache Kafka running on our local cluster and on the cloud platform. In our RPi cluster, version 0.9.0 is used and the cloud is configured to use Kafka version 0.8.2. Due to this issue, the Java APIs of Kafka for producer and consumer application were throwing a buffer error and some of the pictures were got corrupted. It is then resolved by using the same version on both sides.

## 6.2 Future Work

Currently, there are five RPi nodes in our IoT sensor node in which Kubernetes framework is configured as a clustered system. Out of five nodes, three are configured as Kubernetes master nodes and two of them are operating as a worker nodes. It is also possible to attach more workers with the cluster in order to provide more processing and computation power. More master nodes can also be attached but they might affect the performance of the overall system, since each master node requires etcd process to operate on the back-end. This process, sometimes, degrades performance if it is setup as a cluster of large number of nodes. On the other hand, our IoT sensor node is just using a single type of external camera sensors but there are countless number of other sensors that can be attached to each RPi node. Some of the examples are ultrasonic, noise, temperature, IR, and photo-cell sensors. As a future work, this node can also be utilize to operate as a surveillance system for public and private sectors. Moreover, this sensor node is managed separately in local and cloud cluster but it may be possible in future to provide a management system for both edge and cloud computing.

# Bibliography

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010, doi:10.1016/j.comnet.2010.05.010.

[2] Algirdas Avizienis. Fault-Tolerant Systems. *IEEE Trans. Computers*, 25(12):1304–1312, 1976, doi:10.1109/TC.1976.1674598.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP 2003, Bolton Landing, NY, USA, October 19-22*, pages 164–177, 2003, doi:10.1145/945445.945462.

[4] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Morgan & Claypool Publishers, 2013. doi:10.2200/S00516ED2V01Y201306CAC024, ISBN: 9781627050098.

[5] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014, doi:10.1109/MCC.2014.51.

[6] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999. ISBN: 0130137847.

[7] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud Computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Syst.*, 25(6):599–616, 2009, doi:10.1016/j.future.2008.12.001.

[8] Marcos Dias de Assunção, Rodrigo N. Calheiros, Silvia Bianchi, Marco A. S. Netto, and Rajkumar Buyya. Big Data computing and clouds: Trends and future directions. *J. Parallel Distrib. Comput.*, 79-80:3–15, 2015, doi:10.1016/j.jpdc.2014.08.003.

[9] Rajdeep Dua, A. Reddy Raja, and Dharmesh Kakadia. Virtualization vs Containerization to Support PaaS. In *2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14*, pages 610–614, 2014, doi:10.1109/IC2E.2014.41.

[10] Irfan Habib. Virtualization with KVM. *Linux J.*, 2008(166), February 2008. url: http://dl.acm.org/citation.cfm?id=1344209.1344217 (accessed 10.6.2016).

[11] Jerry Honeycutt. Microsoft Virtual PC 2004 Technical Overview. *Microsoft, Nov*, 2003. url: http://www-ti.informatik.uni-tuebingen.de/~spruth/Mirror/partit/VirtPC02.pdf (accessed 10.6.2016).

[12] Jinpeng Huai, Qin Li, and Chunming Hu. CIVIC: A Hypervisor based Virtual Computing Environment. In *2007 International Conference on Parallel Processing Workshops (ICPP Workshops), 10-14 September, Xi-An, China*, page 51. IEEE Computer Society, 2007, doi:10.1109/ICPPW.2007.28.

[13] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25*, 2010. url: https://www.usenix.org/legacy/event/usenix10/tech/full_papers/Hunt.pdf (accessed 11.5.2016).

[14] Ann Mary Joy. Performance Comparison Between Linux Containers and Virtual Machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, pages 342–346, March 2015, doi:10.1109/ICACEA.2015.7164727.

[15] Jan Kiszka. Towards Linux as a Real-Time Hypervisor. In *Eleventh Real-Time Linux Workshop*, page 205. Citeseer, 2009. url: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.385.2999&rep=rep1&type=pdf#page=215 (accessed 11.6.2016).

[16] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A Distributed Messaging System for Log Processing. NetDB, 2011. url: http://people.csail.mit.edu/matei/courses/2015/6.S897/readings/kafka.pdf (accessed 11.5.2016).

[17] Alexandros Labrinidis and Hosagrahar V. Jagadish. Challenges and Opportunities with Big Data. *PVLDB*, 5(12):2032–2033, 2012, doi:10.14778/2367502.2367572.

[18] Qing Li and Caroline Yao. *Real-Time Concepts for Embedded Systems*. CRC Press, 2003. ISBN: 978-1578201242.

[19] Karl Matthias and Sean P Kane. *Docker: Up & Running*. O'Reilly Media, Inc., 2015. ISBN: 978-1-491-91757-2.

[20] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of Things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012, doi:10.1016/j.adhoc.2012.02.016.

[21] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. Lightweight Virtualization: A Performance Comparison. In *2015 IEEE International Conference on Cloud Engineering (IC2E), Tempe, AZ, USA, March 9-13*, pages 386–393, 2015, doi:10.1109/IC2E.2015.74.

[22] Al Muller, Seburn Wilson, Don Happe, Gary J. Humphrey, and Ralph Troupe. *Virtualization with VMware ESX Server*. Syngress Publishing, First edition, 2005. ISBN: 978-1-59749-019-1.

[23] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20*, pages 305–320, 2014. ISBN: 978-1-931971-10-2.

[24] Michael Rabinovich, Zhen Xiao, and Amit Aggarwal. Web Content Caching and Distribution. chapter Computing on the Edge: A Platform for Replicating Internet Applications, pages 57–77. Kluwer Academic Publishers, Norwell, MA, USA, 2004, doi:10.1007/1-4020-2258-1_4. ISBN: 1-4020-2257-3.

[25] Matthew N.O. Sadiku, Sarhan M. Musa, and Omonowo D. Momoh. Cloud Computing: Opportunities and Challenges. *Potentials, IEEE*, 33(1):34–36, Jan 2014, doi:10.1109/MPOT.2013.2279684.

[26] Kristian Sandström, Aneta Vulgarakis, Markus Lindgren, and Thomas Nolte. Virtualization Technologies in Embedded Real-Time Systems. In *Proceedings of 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation, ETFA 2013, Cagliari, Italy, September 10-13*, pages 1–8, 2013, doi:10.1109/ETFA.2013.6648012.

[27] Amir Ali Semnanian, Jeffrey Pham, Burkhard Englert, and Xiaolong Wu. Virtualization Technology and its Impact on Computer Hardware Architecture. In *Eighth International Conference on Information Technology: New Generations, ITNG 2011, Las Vegas, Nevada, USA, 11-13 April*, pages 719–724, 2011, doi:10.1109/ITNG.2011.127.

[28] Priti Sharma, Brijesh Kumar, and Pooja Gupta. An Introduction to Cluster Computing using Mobile Nodes. In *ICETET*, pages 670–673. IEEE Computer Society, 2009, doi:10.1109/ICETET.2009.28.

[29] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7*, pages 1–10, 2010, doi:10.1109/MSST.2010.5496972.

[30] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy C. Bavier, and Larry L. Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23*, pages 275–287, 2007, doi:10.1145/1272996.1273025.

[31] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference, June 25-30, Boston, Massachusetts, USA*, pages 1–14, 2001. url: `http://www.vmware.com/pdf/usenix_io_devices.pdf` (accessed 11.6.2016).

[32] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10*, pages 1013–1020, 2010, doi:10.1145/1807167.1807278.

[33] James Turnbull. *The Docker Book: Containerization is the New Virtualization.* James Turnbull, 2014. url: `http://books.linuxfocus.net/files/books/James.Turnbull.The.Docker.Book.Containerization.is.the.new.virtualization.B00LRROTI4.pdf` (accessed 15.4.2016).

[34] Anthony Velte and Toby Velte. *Microsoft Virtualization with Hyper-V.* McGraw-Hill, Inc., First edition, 2010. ISBN: 0071614036, 9780071614030.

[35] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24*, pages 18:1–18:17, 2015, doi:10.1145/2741948.2741964.

[36] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March, San Diego, CA, USA*, pages 1163–1171, 2010, doi:10.1109/INFCOM.2010.5461931.

[37] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a Replicated Logging System with Apache Kafka. *PVLDB*, 8(12):1654–1665, 2015, doi:10.14778/2824032.2824063.

[38] Li Da Xu, Wu He, and Shancang Li. Internet of Things in Industries: A Survey. *IEEE Trans. Industrial Informatics*, 10(4):2233–2243, 2014, doi:10.1109/TII.2014.2300753.

[39] Andrea Zanella, Nicola Bui, Angelo Paolo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of Things for Smart Cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014, doi:10.1109/JIOT.2014.2306328.

[40] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud Computing: State-of-the-art and research challenges. *J. Internet Services and Applications*, 1(1):7–18, 2010, doi:10.1007/s13174-010-0007-6.

# Appendix A

# Configuring Kubernetes cluster

## A.1 Service configuration files for Kubernetes cluster

In this appendix, the kubernetes configuration files are presented which are based on Linux systemd service files. These files are divided in terms of Kubernetes master and worker nodes. All files are uploaded on a GitHub repository that can be accessed through the following link.

`https://github.com/asad26/IoT-sensor-node`

## A.2 Shell scripts for monitoring master nodes

This appendix presents some important shell scripts that are used to monitor the API server of Kubernetes master nodes. Since there are three masters in our cluster, it is necessary for worker nodes to point to the correct master in case of a node failure. This is done by using the netcat tool inside scripts that can check whether the API server is running on a particular port. These script files are also uploaded on a GitHub and accessed through the following link.

`https://github.com/asad26/IoT-sensor-node/tree/master/scripts`

# Appendix B

# Configuring and running Kafka cluster

## B.1 Configuration files for creating pods

This appendix focuses towards the YAML files that are used by Kubernetes to deploy pods. These files start two containers inside each pod that will setup Kafka and Zookeeper cluster. In addition, replication controller is also created along with each pod which will continuously monitor the respective pods. All files can be seen from the following link.

`https://github.com/asad26/Kubernetes-pods`

## B.2 Code for Kafka producers and consumers

This appendix shows Java codes for Kafka producers and consumers that are used to produce and consume to the Kafka topic. Five separate producers are executed in the form of a jar file that continuously monitors the directory and whenever there is a picture, it is sent to the Kafka topic. In addition, five local topics are used based on five camera modules. On the other hand, there is just one consumer, that is consuming from all five topics and send the data to the remote topic on the cloud. All code files can be seen on GitHub from the following link.

`https://github.com/asad26/Kubernetes-pods`

## B.3 Script inside Zookeeper and Kakfa container for cluster configuration

### Zookeeper script

The following shell script is used to configure Zookeeper cluster inside five containers. This script starts by getting the command line argument for id and then write that integer value to the file named `myid`. After that it starts the Zookeeper server and then goes to sleep.

```
#!/bin/bash
#
id=$1
echo "$id" > /var/zookeeper/data/myid
sleep 3
/opt/zookeeper-3.4.8/bin/zkServer.sh start
sleep infinity
```

### Kafka script

The following script configures Kafka cluster inside containers. This script is same for each container except the last line which runs the producer application using Java and that is different with respect to each RPi node. The script first updates the properties file and then wait for the Zookeeper to start. Once the Zookeeper has been started, it then executes Kafka broker and then produces the images.

```
#!/bin/bash
#
id=$1
hostIP=$(/bin/hostname -I | /usr/bin/awk '{print $1}')
sed -i "s/broker.id=.*/broker.id=$id/g" \
          /opt/kafka_2.11-0.9.0.0/config/server.properties
sed -i "s/host.name=.*/host.name=$hostIP/1" \
          /opt/kafka_2.11-0.9.0.0/config/server.properties
sed -i "s/advertised.host.name=.*/advertised.host.name=$hostIP/g" \
          /opt/kafka_2.11-0.9.0.0/config/server.properties

while true
do
        zstat=$(echo stat | nc $hostIP 2181 | grep Mode | cut -f 2 -d ' ')
        if [ $zstat = 'follower' ] || [ $zstat = 'leader' ]; then
                nohup /opt/kafka_2.11-0.9.0.0/bin/kafka-server-start.sh \
                /opt/kafka_2.11-0.9.0.0/config/server.properties &> \
                /opt/kafka_2.11-0.9.0.0/logs/kafka-boot.log &
                sleep 2
                break
        fi
        sleep 2
done
sleep 10
java -jar /opt/producer1-local.jar
```