Aalto University
School of Science
Master's Programme in ICT Innovation: Cloud Computing and Services

Maryam Tavakkoli

Analyzing the Applicability of Kubernetes for the Deployment of an IoT Publish/Subscribe System

Master's Thesis
Espoo, October 21, 2019

Supervisor: Professor Antti Ylä-Jääski, Aalto University
Advisor: Dr. Kimmo Hätönen, Nokia Bell Labs

Aalto University
School of Science
Master's Programme in ICT Innovation: Cloud Computing and Services

ABSTRACT OF MASTER'S THESIS

Author:	Maryam Tavakkoli		
Title:	Analyzing the Applicability of Kubernetes for the Deployment of an IoT Publish/Subscribe System		
Date:	October 21, 2019	**Pages:**	90
Major:	Master's Programme in ICT Innovation: Cloud Computing and Services	**Code:**	SCI3081
Supervisor:	Professor Antti Ylä-Jääski		
Advisor:	Dr. Kimmo Hätönen		

With the increased availability and affordability of miniature computing devices, such as sensors, the era of Internet of Things (IoT) has arrived. Meanwhile, the developments of the cellular mobile technologies and emerge of 5G accelerated the adoption of IoT scenarios within mobile networks. Rapid growth in the number of IoT devices has resulted in greater volumes of data being generated and exchanged between various entities. This highlights the need for efficient data transmission between data producers and consumers. To this goal, Nokia Bell Labs has developed a distributed data dissemination system based on Publish/Subscribe messaging protocol and according to the micro-service architecture.

Currently, the system is deployed on virtual machines (VM), where the corresponding compiled Java file (Jar file) of each micro-service is running on a separate VM. This approach for deployment might cause a problem, named 'Dependency Hell' for the continuous integration and continuous development (CI/CD) workflow. Moreover, one of the system's limitations is that no service discovery is present. Instead, services are connected through hard-coded IP addresses, defined on their corresponding configuration files. Hence, IPs must be changed manually per deployment according to the infrastructure.

To solve the present limitations, this thesis proposes a deployment based on Kubernetes. Kubernetes is a container orchestration framework that introduces several benefits for the deployment process including automation, management, monitoring and scaling of multi-container packaged applications, such as the current Pub/Sub IoT system. A proof-of-concept solution for the deployment of the system using Kubernetes is presented and its implications on the system's efficiency and scalability are discussed.

Keywords:	Internet of Things (IoT), 5G, Publish/Subscribe, Deployment, Virtual Machine, Container, Docker, Kubernetes
Language:	English

"What you seek is seeking you."
- RUMI

Acknowledgements

This thesis is made as a completion of the EIT Digital Master program in Cloud Computing and Services. This work is the result of hard work during the last semester of my master's studies. As required by EIT Digital Master School, the thesis was carried out in a company and resulted in a collaborated work between academia and industry. More specifically, Nokia Bell Labs, located in Espoo, Finland, provided me this internship opportunity. During this experience, I was able to apply my academic knowledge to a practical project and it has been a stage of intense learning for me. Therefore, I hope that it can provide insightful sources to my colleagues for further research.

I have received a great deal of assistance and support from many people during the writing of this thesis. I would therefore firstly like to thank my supervisor at Aalto University, Professor Antti Ylä-Jääski, for his time, valuable input and support throughout the entire master period.

Secondly, I would particularly like to single out my daily supervisor at Nokia Bell Labs, Dr. Kimmo Hätönen. I would like to thank him for trusting me and providing me the opportunity to work on this research project and for his constructive guidelines and valuable insight into the topic. I am also grateful to all of those with whom I have had the pleasure to work during this project at Nokia Bell Labs.

I would also like to express my special gratitude to my best friend, my loving husband, who has been a constant source of encouragement and support. Last but not least, I would like to thank my parents and my sister who support me in every step of my life. Without their love and support, it was not possible for me to excel in my master studies and complete this thesis project.

Espoo, October 21, 2019

Maryam Tavakkoli

Abbreviations and Acronyms

5G	5th Generation of Telecommunication Networks
BTS	Base Transceiver Station
CI/CD	continuous integration and continuous development
CN	Core Network
DF	Data Fetcher
DH	Data Hub
DNS	Domain Name System
DS	Data Switch
GB	Gigabyte
I/O	Input/Output
IoT	Internet of Things
KPI	Key Performance Indicator
MEC	Multi Access Edge Computing
NESC	Nokia Engineering and Services Cloud
OS	Operating system
PaaS	Platform-as-a-Service
RAM	Random Access Memory
RAN	Radio Access Network
SSH	Secure Socket Shell
TCP	Transmission Control Protocol
UE	User Equipment
VM	Virtual Machine
YAML	Yet Another Markup Language

Contents

Chapter 1

Introduction

We are in the era of the Internet of things (IoT) where billions of devices get connected to the Internet and the generated data is rapidly growing. The number of IoT devices is projected to surpass 75.4 billion by 2025 [1]. Meanwhile, Fifth-Generation (5G) telecommunications technologies will bring faster mobile connectivity, which makes the possibility of IoT scenarios even more feasible. According to Ericsson's mobility report [2], there will be around 1.8 billion IoT devices with cellular connections in 2023.

The rapid surge in the number of IoT devices has resulted in greater volumes of data being generated and exchanged between various entities. This highlights the need for a suitable message delivery solution in an IoT use-case. Among the possible approaches, publish/subscribe is one of the popular communication patterns which allows the distribution of data to the interested entities in an efficient way.

A distributed transmission bus based on the Publish/Subscribe paradigm is being developed in Nokia Bell Labs [3]. The microservice architecture design is used for the development of this IoT data transfer system. This architecture structures an application as a collection of small services, where each is running in its process and is independently deployable. To take the current project one step further to the real use-case implementation, we need to make it easily deployable.

Deploying a software application traditionally meant to have dedicated physical servers that would run the entire application. This approach is already outdated since it is very costly and time-consuming. As an alternative, virtualization technologies have been developed, where several new machines can be easily launched to serve various needs. Resources can be consumed more effectively using virtualization techniques compared to the bare-metal setups. However, still, efficiency could be increased further [4]. Moreover, deploying and running applications directly on the infrastructure might cause

a problem known as 'Dependency Hell' and later will cause troubles for the CI/CD workflow.

The developments in the Linux kernel have led to the evolution of container-based virtualization [5]. A lightweight virtualization, which has gained its momentum recently due to its efficiency and mobility [6]. Container virtualization seems to be a viable alternative to the hypervisor for application infrastructure[4].

Using container-based virtualization, we can deploy each component of the Pub/Sub IoT micro-service system as an independent separate service. Furthermore, containers can be run in any infrastructure without any dependency. However, the availability of these services becomes a concern when running multiple different containers in a micro-service architectural style [7], and we need to make sure that our system is always up and running.

A system is needed to manage the set of these containers, especially when it comes to applications that might comprise hundreds of containers. This is where a container management tool such as Kubernetes comes in, and provides mechanisms for deploying, maintaining and scaling of the containerized micro-services. This way Kubernetes manages the availability of containers while it hides the complexity of their orchestration [7].

Due to its micro-service architecture, each of the modules in the current IoT pub/sub prototype system can be developed and executed separately. However, in the current state of the system, these services are connected through hard-coded IP addresses that are defined by the developers. These pre-defined addresses cause deployment of the system to be dependent on the environment. Hence, the used approach for the deployment should help for removing this dependency. The approach also should support the deployment of the system in the larger scales in the future.

This thesis analyses the suitability of containerizing the current Pub/Sub IoT microservice-based system and using Kubernetes for its deployment, while the aforementioned demands of the system are met.

1.1 Motivation and Scenarios

This section emphasizes the motivation behind this thesis work. The main motivation comes from an innovation ecosystem, named *'Smart Otaniemi'* [8], which is being evolved in Finland. The ecosystem brings researchers, experts, companies, and technologies together to pilot novel projects on renewable and smart energy with the aim of a sustainable future. The project is funded by Business Finland [9] and Nokia Bell Labs is one of the piloting partners in the ecosystem.

Currently, there are several active pilot projects, while *'Platforms, connectivity and enabling technologies'* is one of them. The main goal of this project is to enable reliable and real-time connectivity and data sharing [10]. Intelligent data services enable exchanging data, that comes from different data sources, across different industry sectors. Moreover, 5G has introduced new communication opportunities and supports for scalable and cost-efficient data platforms that can be used in energy, building, and communication market sectors.

As one of the vertical use cases within this project, Nokia's Pub/Sub IoT system serves such connectivity and data sharing. For instance, it can be used for remote monitoring of heating, cooling and air conditioning (HVAC) [10]. Currently, the development of Nokia's system is finalized. However, it is still having limitations to be deployed in a real-world scenario. Moreover, there are specifications that should be considered towards an efficient deployment.

One of the current limitations is the absence of a service discovery mechanism to connect the micro-services within the system. Hence, as a temporary solution, they are connected with hard-coded IP addresses. Furthermore, adapting the features of edge computing is a specification that must be met for the deployment of the system. More specifically, one of the micro-services of the system is developed to perform some data pre-processing. Hence, it would be fully utilized when it resides at the edge of the network, where is closer to the data sources.

To be more precise on the motivation for this thesis work, we aim to study whether Kubernetes can be used for the deployment of such a system and if it removes the limitations and meets the specifications.

1.2 Research Questions and Goals

This thesis develops a proof-of-concept solution for the deployment of the current IoT pub/sub system using state-of-the-art technologies and frameworks. A suitable framework for the deployment of a micro-service architecture in a production level must have an easy setup with minimum manual configurations and be scalable and highly available. Moreover, it should not reduce the efficiency and performance of the system. This thesis studies the feasibility of Kubernetes as a suitable framework to satisfy the aforementioned requirements. The research topic is broken down into the detailed research questions below.

- **RQ1. How can we ease deployment of the Pub/Sub IoT sys-**

tem and eliminate system's limitations for a large-scale deployment?

Currently, the Pub/Sub IoT system is deployed on different VMs. However, running an application directly on the infrastructure might cause the dependency hell problem.

Moreover, Nokia's IoT pub/sub system has a micro-service architectural design. This architecture is beneficial for the deployment, maintenance, and further development of the system. However, these independent services are connected via hard-coded IP addresses in the current state of the system. This study aims to automate the deployment of the system and to minimize manual configuration and installation work. This is especially important when we aim to deploy the system on a large-scale. In this thesis, we study the applicability of Kubernetes and its service discovery mechanism to achieve this goal.

- **RQ2. Which containerization and Kubernetes mechanisms should be used in the deployment procedure to serve the requirements of the system?**

  In this thesis, we study the best approaches for containerizing the current Pub/Sub IoT system and then, managing containers and deploying the system using Kubernetes, as the applied container's orchestration tool. The deployment architecture is designed according to the requirements of the system. An example of such requirements is to consider which micro-services must be exposed externally and be accessible by whom. A prototype of such deployment on a small scale is presented in this thesis.

- **RQ3. What are the implications of the system's deployment based on Kubernetes for the efficiency of the system and its scalability?**

  Finally, we investigate the consequences and results of the proposed proof-of-concept solution. We aim to answer questions, such as, how the deployment approach reduces the manual work, how it affects the performance of the system and how scalable is the proposed solution.

1.3 Research Approach

To answer the RQ1, we conduct a literature review and study the containerization technologies and the orchestration frameworks to find the best-suited

approach for the deployment of the current system. During this step, we specifically paid attention to the *'service discovery'* to be included as a necessary feature in the used framework.

To answer the RQ2, we implement a proof-of-concept solution for the deployment of the system using Kubernetes. To accomplish this, first, each of the microservices within the system is containerized separately and the corresponding container images are created. Next, a private registry is set up to store these images. Later on, all the required steps from installing Kubernetes to the system's deployment using Kubernetes are discussed.

To answer the RQ3, we design different test scenarios based on various metrics and present the results. Furthermore, we discuss the suitability of the proposed approach using Kubernetes for the current Pub/Sub IoT system according to the presented metrics.

1.4 Contributions and Thesis Structure

This thesis is divided into 5 chapters. After introducing the problem statement and motivation for the research in Chapter 1, the rest of this thesis and the main contributions are structured as follows.

In Chapter 2, we present the core concepts of the research topic in two main parts. First, the essential background information about the Pub/Sub IoT microservice-based system is provided. Then, the relevant technologies and terminologies that are used for the deployment of the system are explained.

In Chapter 3, we present a proof-of-concept prototype solution for the deployment of the system. To this goal, first, we explain the steps taken for containerizing the system. Then, we describe a use-case scenario and explain our prototype's design decisions according to that. Finally, we discuss the procedures for setting up a Kubernetes cluster and deploying the Pub/Sub IoT system on top of it.

In Chapter 4, we design some test scenarios to evaluate the proposed proof-of-concept solution. Furthermore, the results of the test-cases are discussed.

Finally, Chapter 5 is a conclusion on the thesis. Summary of the thesis and the future possible works are discussed in this chapter.

Chapter 2

Background and Concepts

The purpose of this chapter is to provide a description of the main concepts and technologies relevant to this thesis. The first four sections provide essential background concepts for the discussed IoT system. The rest of the sections in this chapter explain the relevant terminologies and technologies that are used in this work.

The chapter starts with a general introduction of 5G and the evolution of mobile communication technologies in Section 2.1.

The Publish/Subscribe paradigm is presented in Section 2.2, in which Apache Kafka and Zookeeper are described as the used Pub/Sub technologies being used in this project. As an approach for developing distributed applications, Micro-service architecture is used for the current system. This architectural design is described in Section 2.3. Then, Nokia Bell Labs' proof of concept, which this thesis is based on, is discussed in Section 2.4.

Section 2.5 describes the concept of system virtualization, and introduces its two different methods including hypervisor and container-based virtualization. Section 2.6 described features of docker container engine, a virtualization framework that is built on top of Linux containers. Section 2.7 explains the concept of container orchestration and the available orchestration frameworks. Finally, Section 2.8 discusses the most trending container orchestration framework, Google Kubernetes, which is the backbone of this thesis.

2.1 5G and Evolution of Mobile Technologies

Mobile telecommunication networks have experienced a tremendous change during the last few decades. Each generation of cellular networks has introduced new standards, capacities and features and the evolution continue.

5G, the last advancement in mobile technologies, has introduced an increased bandwidth, higher data rates, lower latency and better QoS for the end-users. Hence, 5G's evolution envisions better support for IoT scenarios.

The background concepts of the mobile technologies and their evolution, 5G and its specific features, such as Multi Access Edge Computing (MEC), and the concept of IoT in the Telecom industry are discussed in this section.

2.1.1 Cellular Network Architecture

The mobile wireless communication network is distributed over areas called cells, which typically are represented as a hexagonal. Each cell includes at least one, but more normally, three fixed-location transceivers known as a cell site or base transceiver station (BTS) [11]. These base stations provide network coverage for cells. By joining them together, these cells ultimately provide radio coverage over a wide geographic area for end-users [12].

Base stations are connected to the core network (CN) through a physical or radio link, while the last link connectivity between the network and user equipment (UE) is wireless through the base stations. This wireless connection is done based on the radio access network (RAN). A RAN resides between user equipment, such as a mobile phone, and provides its network connection through its core network [13]. A high-level architecture of the cellular network is shown in Figure 2.1.

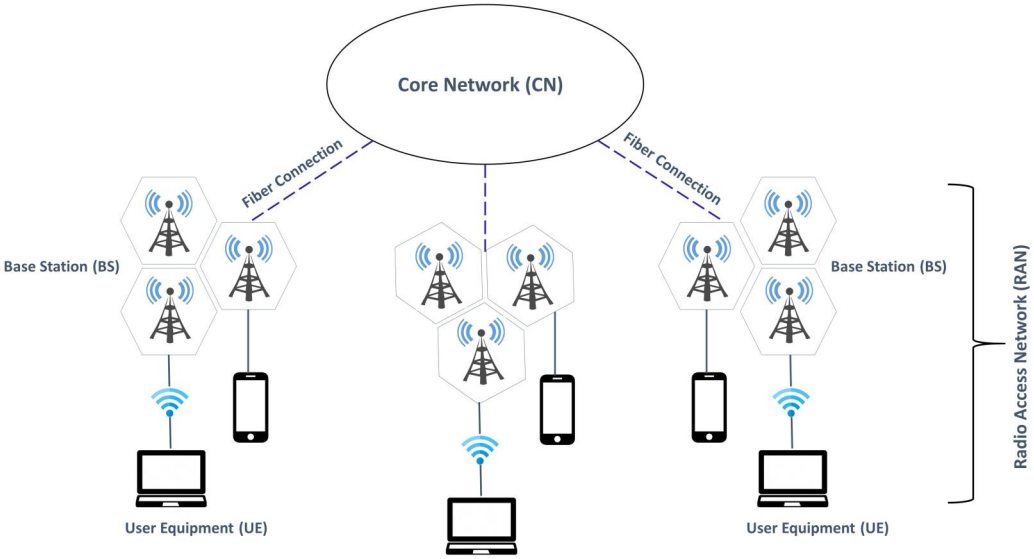

Figure 2.1: High-level architecture of cellular networks.

2.1.2 Evolution of Mobile Wireless Technologies

Telecommunication networks have gone through several evolution phases and generations over the last few decades. The main motivation behind this rapid development is achieving more bandwidth and lower latency for transmitting data [14]. Each generation of mobile communications introduces specific standards or features in terms of data capacity, speed, frequency, latency, etc., which differentiates it from the previous ones [15].

The first generation (1G) of mobile networks was deployed in the 1980s [15]. 1G was analog and made only voice calls possible. The second-generation (2G) were digital systems and supported text messaging. Higher data transmission rate and more data capacity were results of mobile technology advancement to 3G, which supported multimedia and allowed real-time video calls [14][15][16].

The fourth-generation (4G) of mobile networks presented an enhanced version of 3G networks. 4G is an IP-based network system mainly aimed at providing higher speed, quality of service and security to handle more advanced multimedia services over IP [16]. LTE, Short for *'Long Term Evolution'*, and A-LTE (Advanced LTE), which are considered as 4G technologies, offered enhanced network capacity, higher bit rates, and lower delay, to make wireless access possible, anytime and anywhere [14]. As the result, the data rate has improved from 64 kbps in 2G to 2 Mbps in 3G and 50 –100 Mbps in 4G [14].

However, at the time of developing 4G, the efficiency of the traditional point-to-point link between the user equipment and base station was approaching its theoretical limits [17]. Hence, in addition to improvements in the data transmission capacity, there was a demand for more network coverage offered by the available infrastructure. To this goal, an increased node density was needed. However, expanding the number of high-power macrocells was not a viable approach due to the economical reasons [17]. To meet these requirements, small cell network (SCN) was introduced as a potential solution to offer a better network coverage and higher data availability in both fixed and mobile environments, while considering resource optimization and cost-efficiency. Small cell is served by a low-power cellular base station and provides a shorter range of network connectivity compared to the traditional macro-cell [18].

Small cells are categorized as femtocells (coverage up to 30 m range), picocells (coverage up to 200 m range), and microcells (coverage up to 2 km range) [19]. While traditional macro-cells are usually used in rural areas, a microcell can be used in a populated urban area. Picocell is suitable for a large office or a mall, and finally, a femtocell covers the smallest area such as

a home or a small office.

2.1.3 The Internet of Things in the Telecom Industry

The explosive growth in mobile data traffic and the number of mobile-connected devices were results of evolution in mobile technologies. As it is reported by Cisco Visual Networking (Cisco VNI) [20], mobile data traffic has risen 18 times over the 5 recent years. Still, it is expected to keep growing, and by 2021, it will reach nearly seven times as much as it was in 2016.

Furthermore, the number of mobile-connected devices is predicted to become 11.6 billion by 2021 [20]. Following this, the novel concept of IoT emerged, where physical devices are enabled to talk and exchange data among each other. An example of such devices is the collection of sensors, which produce different types of data, to be used for various use-case scenarios including smart home, smart industries, self-driving assistance for vehicles, etc. [21].

Popularity and growth of the IoT concept demand higher data rates, lower latency, and mobility support. To meet IoT application's demands, mobile operators have been trying to find new solutions to boost the capacity of services and provide a higher quality of service in such scenarios. The development of small-cells such as femtocells is a beneficial deployment in this regard. Using femtocells, considerable amount of data traffic would be offloaded from macro-cells. This will reduce the cost significantly from the operator's side and results in lower power consumption in user's devices [20].

Therefore, the 4G has undoubtedly improved the capabilities of cellular networks to make IoT scenarios possible, and cellular networks are already being used in many IoT scenarios. However, 4G is not fully optimized for IoT applications [22]. We are witnessing an exponential rise in the amount of multi-media data being generated by new IoT scenarios, and today's usage is a burden on the current cellular networks [23]. Hence, there is a demand for higher data rates and lower latency to transmit such data faster and cheaper. The next mobile generation, 5G, will be able to support the transmission of 1000 times more mobile traffic than 3G and 4G [20]. Hence, it is expected to act as the backbone of IoT applications in the future.

2.1.4 5G

4G provides the foundation for the next generation of cellular mobile networks, 5G. 5G is initially deployed alongside 4G LTE [24]. The fifth generation of wireless mobile technology envisions an extended network coverage,

higher data rates, lower latency, increased bandwidth and considerable improvement in the quality of service (QoS) which is received by user [25][23].

With a focus on IoT and low latency systems, 5G aims to connect everything. To meet this goal, data rates up to 20 Gbps, 1000 times increase in the capacity, latency lower than one millisecond, 10,000 times more data traffic and support for ultra-low-cost and reliable IoT applications are promised [24]. To achieve all 5G's targets, new technologies including new network architecture, antenna technologies, spectrum, and network slicing are needed [24]. MEC, as an important feature of 5G, which further enables the possibility of IoT scenarios, is discussed in the following section.

2.1.5 Multi-access edge computing (MEC)

To deliver the promised 5G, mobile network providers adapt and utilize Multi-access edge computing (MEC) architecture along with the development of 5G network [26]. MEC offers computation and storage capabilities at the edge of the network, within the RAN, and very close to the end-users.

The motivation behind the development of this technology is to provide low-latency, high-bandwidth, scalable and reliable access to radio network resources for applications that demand highly real-time and efficient service delivery [27][28]. Moreover, being geographically distributed, MEC provides high-quality of service and broad support for mobility [29]. Considering the benefits MEC introduces, there is tremendous potential for third-parties to make use of mobile operator's services and develop their new and innovative applications utilizing caching capabilities at the edge [28]. IoT use-cases and scenarios are among these new business opportunities being enabled through emerge of MEC and 5G. Facilitating computing and storage resources at the edge of the network, very close to where data generates, ensures a fast response to user's request and supports critical IoT applications, which require data transmissions with low-latency.

The current project can be considered as an example of where MEC architecture assists in developing an IoT scenario: A local IoT gateway, which acts as a MEC platform, receives data from a set of IoT devices, performs some data aggregation or big data analytics on it, and finally sends the acquired information to the associated cloud servers for further functionalities.

Therefore, the new decentralized architecture provided by MEC, enables many IoT applications that need to be served with geo-distribution, location-awareness, scalability, and low-latency features [29]. A high-level architecture of MEC setting can be seen in Figure 2.2.

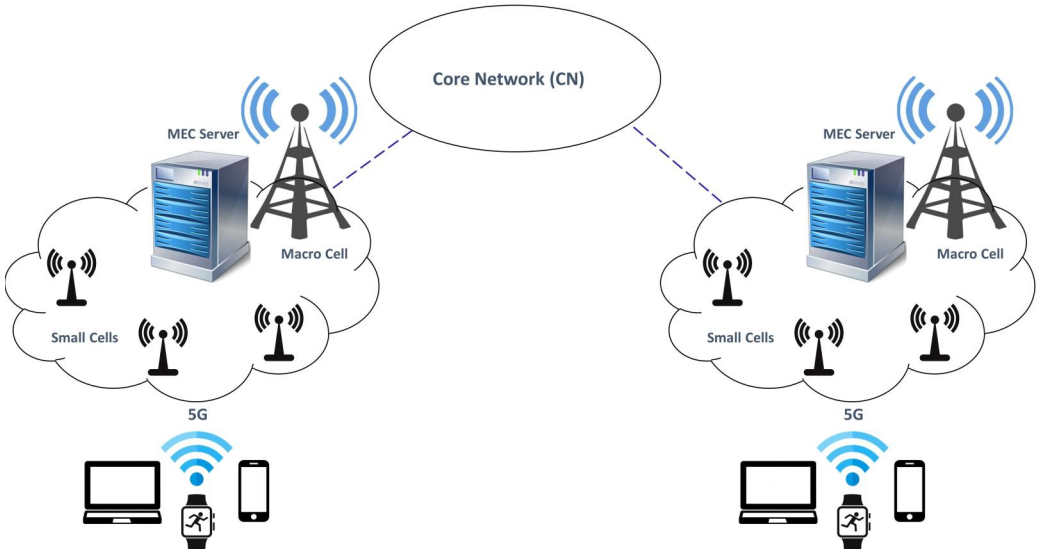

Figure 2.2: High-level architecture of MEC settings.

2.2 Publish/Subscribe Paradigm

Publish/Subscribe model is a message-oriented middleware (MoM) [30], which acts as an alternative to the traditional client-server architecture. In the traditional model, client communicates directly with an endpoint.

Pub/Sub model provides distributed and loosely coupled communications between its two main entities: Client that sends a message is called producer (publisher) and the client or clients that receive the messages is called consumer (subscriber). There is also a third component, the broker, which routes messages between publishers and subscribers. Role of the broker is to distribute incoming messages to the correct subscribers [31]. A more precise architecture overview of the Pub/Sub messaging model is provided in the next sub-section. Moreover, Kafka and Zookeeper, as the Pub/Sun technologies that have been used in this work, are discussed in the following sub-sections.

2.2.1 Publish/Subscribe Architecture

Figure 2.3 depicts the general scheme of the Pub/Sub model. Publishers send messages (publish data) to the event bus which is including a broker. The publisher does not specify any address of the receiver. Instead, the event bus decides, which subscriber should receive the data, and to let the event bus know this, subscribers subscribe for those messages which they are interested

in. Publishers deliver data to the broker only once, and then it is the broker, who distributes the same data to different subscribers, considering the subscriptions of each user. The strength of publish/subscribe model commu-

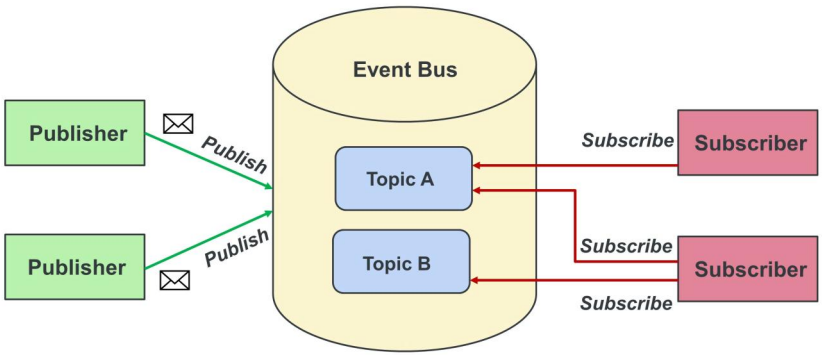

Figure 2.3: General scheme of Publish/Subscribe communication.

nication style lies in the three types of full decoupling that it provides [32], namely decoupling in space, time and synchronization between publishers and subscribers. Space decoupling means messages are not specifically sent for a particular receiver, but any entity who is willing to receive a particular message can subscribe to it. Hence, the sender is required to know neither the exact receiver nor the number of them. Similarly, the receiver does not need to know who was the source of the received message. This feature is called space decoupling.

Time decoupling is the second type of decoupling, which states that message producers and consumers do not need to be connected at the same time to be able to send or receive data. In other words, message delivery is not dependent on the presence of both entities. Messages can be sent even when receivers are not connected at that moment, and consumers will be able to receive messages even if senders are not available.

Lastly, Pub/Sub offers decoupling in synchronization, which means events of sending and receiving messages are asynchronous and non-blocking in this model. Hence, there is no need for senders to wait for acknowledgments from receivers after they sent a message. Providing decoupling features that support the independence of the sender and receiver, Pub/Sub model brings more scalability, and hence makes it a well-suited option for large-scale IoT deployments [33]. Moreover, Supporting asynchronous communications better addresses requirements in distributed systems, which are asynchronous by nature [32].

Matching between publishers and subscribers can be done based on different design types of filtering in a Pub/Sub system, mostly topic-based or

content-based [33]. The way that subscribers describe which messages they are interested in depends on the chosen design. Topic-based or subject-based is the earliest Pub/Sub design type which has been used in many industrial solutions [32]. Also, most of current popular Pub/Sub systems such as Apache kafka [34], IBM MQ [35] and Message Queuing Telemetry Transport (MQTT) [36] work based on this filtering design scheme. Topics can be considered as individual key-words, and subscribers are allowed to subscribe for one or more topics. Publishers should assign each message a topic or topics. So, when the event bus receives a message, it first checks its assigned topic/topics. If there were subscribers who subscribed to that topic/topics, the message will be routed to them.

This very basic performance of the topic-based solution introduces some limitations in the implementation. A topic may be too general and a subscriber may not be interested to receive all data being covered under that topic but is only interested in a specific part. So, in this way subscriber has to filter the part of the received data to find its specific part, and also disregard the other additional parts. This is also not beneficial in network usage since some data is delivered, which is not going to be used. To solve the described drawback, the topic-based scheme is developed by various systems.

Content-based is the second design type which allows subscribers to subscribe for the actual content of a messages. So, consumers can announce their interest only to a subset of events, and to do that they should define some filters and constraints for part of the event's content that are interested [37]. This filtering is usually described as a pair of attribute and value. Attributes are defined based on content and a logical function will be evaluated as the attribute's value. When a message is received by the event bus, its content will be evaluated against each subscription, and if the result was positive message will be sent to the corresponding subscriber. Considering a topic as an attribute in this approach, we can even conclude that a content-based solution will cover the topic-based design of Pub/Sub systems. Among current industrial Pub/Sub systems IBM MQ provides content-based architecture [35].

2.2.2 Apache Kafka

Apache Kafka [34], an open-source software platform, is a distributed and fault-tolerant message delivery system based on publish/subscribe architecture. Following topic-based design and written in Scala and Java programming languages, it was initially developed by LinkedIn to provide efficient message delivery to multiple users [38]. Currently, Kafka is maintained by Apache foundation and is widely used in Big Data solutions for building

real-time data pipelines [38].

The arbitrary number of messages come from different processes called producers. Then, Kafka groups received data into different topics based on their data stream category. Each topic, later, will be divided to partitions of equal size. Kafka runs on top of a server or a cluster of servers, called brokers. Partitions of all topics are distributed over these cluster nodes. Furthermore, partitions will be replicated to multiple brokers, which will result in a fault-tolerant message delivery system. Brokers are responsible for maintaining published data and sending it to the subscribers later. Other processes, called consumers, who wish to receive data corresponding to a topic, will initiate a subscription request, and it can finally read messages from partitions.

The overall architecture of Kafka and its components are depicted in figure 2.4.

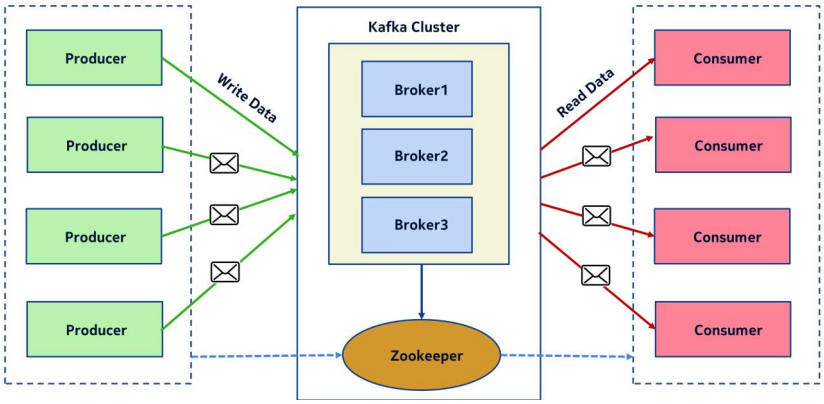

Figure 2.4: General architecture of Apache Kafka.

2.2.3 Apache Zookeeper

Apache Zookeeper [39], an open-source project developed by Apache, provides centralized service for maintaining naming and configuration data as well as flexible and robust synchronization over large clusters in distributed systems [39]. providing coordination and synchronization service, Zookeeper acts as a dependency for running Kafka. Zookeeper brings a coordination interface between the broker and consumers. It also keeps track of Kafka cluster nodes, topics and partitions.

2.3 Micro-Service Applications

The micro-service architecture, which has gained momentum during the last few years, is an architectural style for designing software applications as a set of small, lightweight independent services. Each of the services runs in its independent process, and they all communicate with a lightweight mechanism. [40].

An opposite approach for software development, which is the traditional one, is called monolithic architecture. In a monolithic design, the entire application logic is written as a single unit. Any changes to a small part of such a system demand for building and deploying a new version of the entire application [41]. This feature of the monolithic architecture causes difficulty in the further development of a system, especially when the application is large.

This is while continuous delivery and deployment are much easier in a micro
service-based architecture [42]. Compiling of the whole application is not needed upon every small change in the microservice application. As a result, bugs and security vulnerabilities of the application can be found and fixed easier. Hence, deploying changes and delivering updated versions of the application to the production environment can be performed faster, and this allows the development team to conduct smaller and more frequent releases. Furthermore, scaling of an application developed with monolithic design, requires the whole of the application to be scaled, while in a micro-service approach, every single component or service is isolated and is scalable without any dependency to the other parts.

Moreover, different services in a micro-service architecture can be distributed on multiple physical hosts. To let different micro-services, which are deployed on different hosts, talk to each other, a service discovery protocol is required [43]. In contrast, all modules in a monolithic application must be deployed as a single unit in the same physical environment. Because of this, the same hardware characteristics, such as RAM and CPU, would be followed for all modules of the monolithic application. However, in a micro-service application, these features could be defined separately. For example, more CPU can be assigned to a service, while more RAM is allocated for the other one.

Figure 2.5 illustrates the differences between monolithic and micro-service architectures. Despite all the benefits of the distributed microservice-based architecture, it also introduces new challenges. The internal communications among services, which is accomplished via remote network calls, gener-

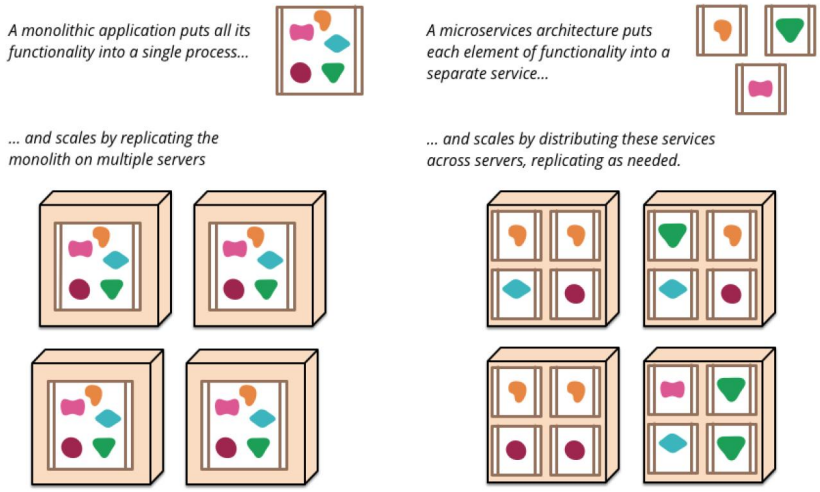

Figure 2.5: Monolithic and Microservice-based architecture [41].

ate network overhead. So, the bandwidth and latency requirements should be considered while developing such an application. Besides, these inter-communications demand for a reliable and secure network [43].

As the microservice-based application grows and turns into a system with a large number of services, it becomes more fault-prone [44] and monitoring of the whole application becomes a difficult task. Hence, an effective automation and monitoring system is required for the deployment of such a system.

2.4 Nokia Bell Lab's Proof of Concept System

A distributed data dissemination system based on the Pub/Sub messaging architecture is being developed in Nokia Bell Labs. The main idea behind the project is to provide effective data transmission between data producers and data consumers. To this goal, content-based design type for Pub/Sub messaging is used, which offers an efficient routing for transferring messages. Furthermore, functionalities such as data compression and aggregation are distributed to the edge of network [45], where data is generated. Motivation for the project and possible use-cases are discussed in the next sub-section. Then, architecture of the system and its components are explained in the following sub-section.

2.4.1 Motivation of the Project

The initial motivation of the project was to develop a distributed management plane data communication to be delivered in the cellular networks [45]. However, later, a wider range of applicability for the project was found. Currently, this prototype system can be used as a general distributed transmission bus to serve different IoT scenarios, where several geo-distributed IoT devices generate data, acting as producers, and different data consumers can access this data if they subscribe for it. The system is developed based on the microservice architecture, where TCP/IP protocol is used for handling communications among services. The current system is written in the Java programming language.

As mentioned earlier, Nokia's system is designed based on Pub/Sub messaging pattern. Those who want to consume data will issue subscriptions to describe type of data they need. Then, the system starts to receive the corresponding data and delivers it to the matching subscriber. In the next section, each service is described in more detail.

2.4.2 Introducing Components of Nokiafls System

A general overview of Nokia's proof of concept system is illustrated in Figure 2.6. Dashboard, Coordinator, Data Fetcher and Data Hub are the main services, which are described as follow:

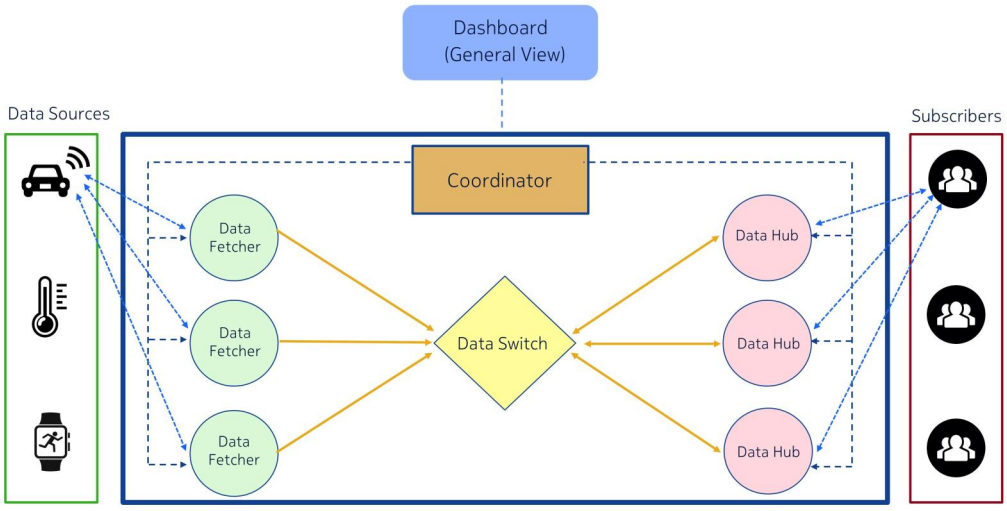

Figure 2.6: General overview of Nokia's prototype system

- **Data Fetcher**

  Data Fetcher is responsible for retrieving data from external data sources into the system. It also performs some pre-processing on data before delivering it to the corresponding consumer. The current data processing includes KPI calculation. However, this module could be developed with further functionalities in the future. Compression and encryption of data are from the other currently available capabilities of data fetcher. Considering the abovementioned features, data fetcher will act more appropriate when it is resided at the edge of the network, where it is closer to the data sources.

- **Data Switch**

  Data switch is not a microservice by itself, instead, it resembles two separate microservices, namely Kafka broker and Zookeeper, which together provide one of the main functionalities of the system: the actual Pub/Sub messaging pattern. Data switch receives the pre-processed data from data fetcher and then routes it to the right consumer or consumers who have subscribed for it.

- **Data Hub**

  Data hub, which has a corresponding UI, is a gateway for subscribers. End users who aim to consume data, subscribe for their specific data type, being shown as a list in the data hub. Upon receiving requests, made by subscribers, data hub forwards them to the coordinator. Later, data hub receives corresponding data for each subscription and forwards it to the end-user.

- **Coordinator**

  Coordinator is considered the main component responsible for managing and orchestrating all other components and services. The whole process of a subscription, creation, and cancellation of a subscription is handled by coordinator. For example, as mentioned earlier, to receive relevant data from data switch, data hub first needs to send its requests to the coordinator. Then, coordinator processes this request and asks data fetcher to publish appropriate data, and finally, data hub will receive the corresponding data.

- **Dashboard**

  Dashboard acts as a user interface for monitoring the state of the system. Currently, it illustrates the state of main microservices, if they are up and running and how much traffic they generate.

2.5 System Virtualization

Virtualization, one of the key concepts in cloud computing, which has been well established for decades [46], refers to creating a virtual version of a resource, such as operating system, hardware, storage or network, in a layer abstracted from the actual one.

These emulated virtualized systems could be configured, maintained and replicated easier and on-demand [47]. Furthermore, by virtualizing, computing infrastructure is assigned to users and applications based on their real needs and hence, resources are much better utilized. This leads to a decrease in the upfront operational costs, as well as a reduction in carbon emissions. Due to these benefits, which not only have economical impacts but also are environmentally friendly, virtualization is seen as one of the green IT technologies [48].

In terms of Linux, virtualization refers to running one or more virtual machines operated by Linux operating system on a single physical computer. The two prevalent Linux virtualization technologies include hypervisor-based and container-based virtualizations, which are described in the following sections.

2.5.1 Hypervisor Virtualization

The first well-know virtualization technology that has been around for decades, is hypervisor, also called Virtual Machine Monitor (VMM). In a hypervisor, computer hardware or software will be the host and provides full abstraction for one or several virtual machines acting as the guests [49]. The host machine splits and allocates its locally available resources to the guest machines. This way each guest VM will have its OS and operate isolated from others. Hence, multiple machines with different operating systems can execute on a single physical host at the same time.

Two different architectures are shown in Figure 2.7 are differentiated for a hypervisor.

Type 1 hypervisors, called native or bare-metal hypervisors, run directly on hardware to control it and manage operating systems for guests [50]. The most well-known examples of this architecture include Oracle VM [51], VMWare ESX [52], Microsoft Hyper-V [53], and Xen [54]. Type 2 hypervisors, called embedded or hosted hypervisors, on the other hand, require a host operating system to run their operations on top of it [50]. Hence, these type of hypervisors are dependent on the host OS for their resource allocation. Some popular hypervisor tools coming from this architecture are

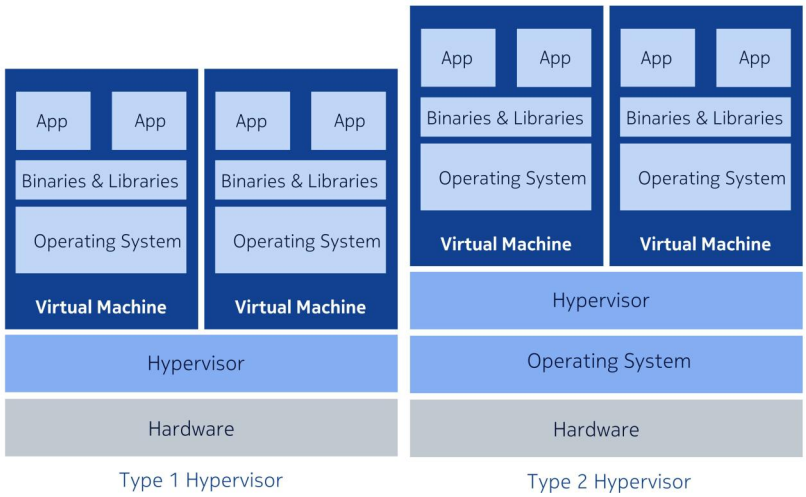

Figure 2.7: Hypervisor-based virtualization architecture.

Oracle VM VirtualBox [51], VMWare Workstation [55], Microsoft Virtual PC [56] and QEMU [57].

Despite differences in types, hypervisors introduce several benefits. Isolation for guest machines is the main one. Isolation guarantees that consequences of any operation within one VM will not affect the other VMs or the host. This prevents any crash, failure or security threat caused by one environment disturb functionality of the other machines. Moreover, running a hypervisor, enables users to have multiple machines with different operating systems on top of a single host machine. Additionally, this introduces various business opportunities for service providers since they will be able to provide a wide range of customer's needs with limited resources.

Despite their benefits, hypervisors have several drawbacks. Firstly, dependency of guest VMs to booting up of the host machine causes a slow startup time. Each VM also needs to be booted up like a normal operating system, which makes it even slower. Upfront costs and possible security vulnerabilities are considered as the other disadvantages.

2.5.2 Container Virtualization

Container-based virtualization, also known as operating system level virtualization, is a lightweight alternative to the hypervisors. This type of virtualization utilizes host's kernel features to creates multiple isolated user-space instances, called containers [49]. Container architecture is shown in Figure 2.8.

A container, from point of view of processes running inside it, looks like a

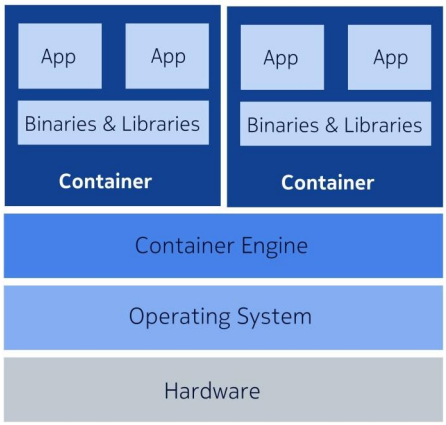

Figure 2.8: Container-based virtualization architecture.

real separate machine, while in reality they are running in an isolated space inside the host OS and share their resources. Hence, despite hypervisors, containers do not have their own virtualized hardware and OS but they use resources of their host. This way, each container acts as an independent OS without any intermediate layer and guest virtualized OS as it is in hypervisors.

Since there is no emulated hardware in containers, they do not need a time for booting an entire OS. This leads to a fast startup, in milliseconds, which is more efficient than the conventional hypervisors [50]. A container encapsulates all packages, which it might need such as libraries, binaries, runtimes and other system-specific configurations [58]. However, it is still more lightweight than a virtual machine, which contains a set of toolchains to run an entire OS, including kernel and drivers. This small resource fingerprint in containers introduces better performance, enhanced security and a good scalability [50].

Container-based virtualization can be implemented on top of any OS, however the current popular techniques such as Docker rely on Linux kernel features. In Linux, resource management for containers is accomplished through Control Groups (Cgroups). Cgroup limits and prioritizes usage of host's hardware resources, such as memory, CPU and I/O, for containers [59]. Moreover, Linux namespaces provide isolation for containers so that each process will have its specific view of the system. An example of this controlled view is to allow a container to see parts of host's file system and not all of it. Also, it controls the list of visible processes in container's process tree.

Due to the all above-mentioned features of containers, they have gained

more popularity during past few years. Realizing advantages of containers compared to the classical hypervisor-based virtualization, developers have just started to use them more widely. To benefit these advantages, in this thesis we use Linux containers for the implementation of our solution.

2.6 Docker Container Engine

Docker, an open-source project in the category of Linux containers, is one of the most popular container virtualization technologies [60]. It is a lightweight platform for developing, deploying and running applications within the containers. Being one of the most powerful technologies at the moment, it is even considered synonymous with containerization in some terminologies. The benefits that Docker brings for deployment and operations are discussed in the next sub-section. Then, in the second sub-section, its architecture and the most important components are explained.

2.6.1 Docker for Development and Operations

Docker provides a fast and automated approach to deploy an application inside portable containers. This way, the application would be suitable for any environment, it can be scaled, and will be configured to interact with the outside world. Moreover, the application, which is built with Docker, will be independent from infrastructure and could be run on any platform. This feature solves the *'dependency hell'* for developers [61] and makes deployment, shipment and test of application easier and faster and shortens the lifecycle for releasing applications. This is specifically beneficial for CI/CD workflows [62].

Additionally, Docker gives the possibility of configuring infrastructure components, such as CPU, memory or networking, to the user through its config files. Hence, developers are able to manage infrastructure the same way they treat to applications. To arrange such characteristics, docker introduces a kernel and an application-level API to the Linux container, which will run processes such as CPU, I/O and memory in isolation [60]. Furthermore, to deploy and run containerized applications, Docker utilizes two most important features of Linux kernel, cgroups and namespaces, as described in Section 2.5.2.

Applications can be run in a secure and isolated space within Docker containers. Hence, using Docker, multiple applications or micro-services could be run simultaneously within different containers on top of the same physical host. According to this possibility, developers can break a huge system to

smaller set of services, each deployable as a separate Docker container. This way debugging, managing and updating of each component would be easier. In addition, the application could be scaled horizontally only with the services which are needed. Also, being more lightweight than the conventional hypervisors, Docker can utilize hardware resources more efficiently and it can manage its workload dynamically.

2.6.2 Docker Architecture and Components

Providing a simple tooling, Docker uses a universal packaging approach to wrap up all application dependencies within a container, which will be run on the Docker Engine [61]. Docker engine is built based on client-server architecture. The Docker Daemon, which runs on the host system, manages the images and containers. Additionally, it is responsible for monitoring state of the system, enforcing policies, executing client's commands and determining namespace environments for running containers. The Docker Client is where

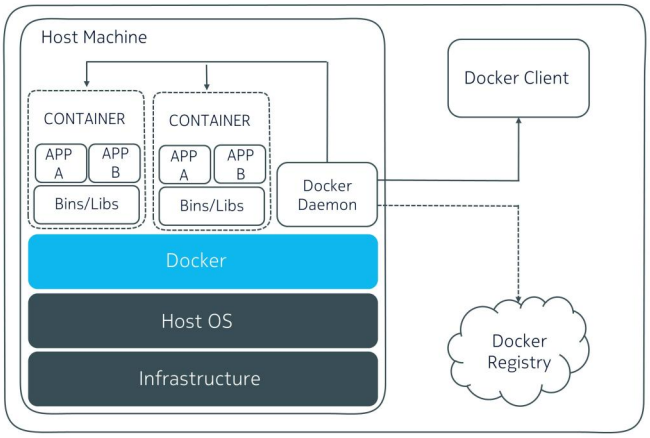

Figure 2.9: Docker Engine architecture.

users interact with Docker. A command-line tool, which connects to the Docker Daemon, is responsible for managing operations related to images, containers, networks, the swarm mode and Docker engine configurations. Using sockets or RESTful APIs, Docker daemon and Docker client communicate with each other. The Docker architecture is illustrated in Figure 2.9. The rest of Docker components are described as following.

- **Docker Image**

  A Docker image is the main building block of Docker. It provides the source-code for creating and running containers. An image contains

all the necessary libraries and binaries to build and run an application. There are pre-built Docker images, which developers can download and use, however they can build their own images as well. To do so, one should write the required instructions in a Docker File. The image template (Docker file) starts with a base image. An image is a series of data layers built on top of the base image. To combine different layers of an image and treating them as a single layer, Docker uses a special file system called Union File System (UnionFS) [63]. Union File System allows combining files and directories of different file systems into a single consistent file system [63]. Upon applying any changes to the image, a new layer will be added on top of the existing ones. Hence, despite the traditional approach based on the hypervisors, rebuilding of the whole image is not needed. This process makes the rebuilding of images quite fast [64].

- **Docker File**

  Docker file is a template, which contains all the required instructions to create a Docker image. The Docker engine acquires the required information for configuring a container or executing the containerized applications from this file. These instructions will be executed in order by conducting a *'Docker build'* command issued by a user. In case any of these instructions result in a change to the current content of the image, changes will be applied to a new layer based on the layering approach, which described before. The base image is specified with the 'FROM' command written in the very first line in the Docker file. This is where the entire image will be built upon.

- **Docker Container**

  Docker containers are the running instances created and deployed from docker images and the allocated system resources, which are manageable through Docker client tool. While several containers can be launched from the same or different images on the same host, they are all isolated from each other and act separately.

  As shown in Figure 2.10, upon starting a container, a new writable layer, namely *'container layer'*, is created on top of its used image [65]. Any changes that occurs within the running container only applies to the container layer. When different containers run on top of the same image, they write their changes to their own writable layers. When the container stops, all of changes made in its container layer will be lost. This way the images will remain immutable and several containers could be made out of a single Docker image.

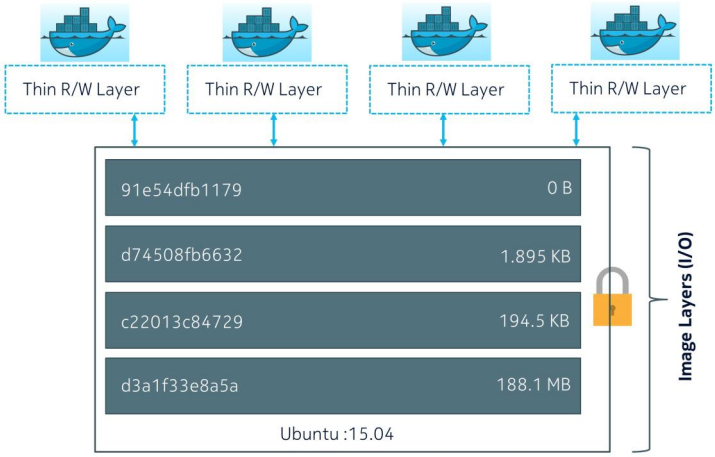

Figure 2.10: Multiple Docker containers on top of one Docker image.

- **Docker Registry**

  Docker registry is where Docker images can be stored. The main purpose of a registry is to simplify distribution of images. Hence, developers can make Docker images and store them in the registries. Then, upon access to the registry using Docker client, other users can download and use the available images. We can consider these registries similar to the source-code repositories [66]. The Docker registry can be either public or private. Docker Hub [67] is the most popular public Docker registry, where you can freely sign up and get access to a huge image storage system for uploading and downloading images. Docker Registry [68], an open-source project lead by Docker, provides private Docker repository for organizations with authenticated access, which lets them control access to their images.

2.7 Container Orchestration Frameworks

With the growth of virtualization technologies, especially launch of Docker, and with the increased interest of PaaS providers, application virtualization became popular [69]. This popularity is because containers, according to their nature, potentially solve many known problems of application development and deployment: Encapsulating an application with all its dependencies as a self-contained software, which can be executed on top of any platform, they solve the *'dependency hell'* problem. This also makes the application

portability much easier. Being more lightweight than VMs, they improve the performance overhead and cut down the startup time [69].

Considering the abovementioned benefits, cloud industry has already adopted the container technologies [69]. However, operating with containers at scale increases the demand for a management and orchestration tool. Multiple containers, which build a distributed architecture, are required to interact with each other smoothly. Hence, as the footprint of the application grows, demand for an automation process increase. Dynamic deployment, automation, management, scaling and monitoring of multi-container packaged applications can be achieved by a container orchestration framework [69]. Such a framework deploys and distributes processes across several physical hosts, monitors them and keeps track of host health. To do so, a container management framework employs a software layer, which abstracts complexity of hosts and displays the entire cluster as a single pool of resources. Furthermore, containers within a cluster are able to communicate with each other regardless of the physical host that they are deployed on. For this purpose, the management framework creates virtual networks between containers. Some of the current framework even offer more and provide load-balancing and service discovery mechanisms. Several orchestration frameworks are being developed, while Kubernetes is one of the most popular ones that we conducted our study based on it is and is discussed more in the next section.

2.8 Kubernetes

Kubernetes is an open-source cluster manger, which initially developed and introduced by Google at the June 2014 Google Developer Forum [60]. Kubernetes's origin has adopted many ideas from Google's first internal container management technology, called Borg [70]. Applications of Google were internally run and managed at scale using Borg. Later, many external developers became interested in Linux containers and Google developed its public cloud infrastructure. These advances motivated Google to develop its open-source container management framework, know as Kubernetes.

Kubernetes can be used for effective deployment, updating, managing, resource sharing, monitoring and scaling of multi-container applications in a highly distributed environment.

2.8.1 Kubernetes Architecture

A high-level architecture of Kubernetes is shown in Figure 2.11. As it is depicted, each Kubernetes cluster consists of a master node and one or more worker nodes. This structure allows Kubernetes to optimize power of cluster computing by distributing containers over different worker nodes, while managing them by the master node. Master node includes components such as API server, controller manager, scheduler and etcd. Worker node has only two components namely kubelet and kubeproxy. Each of these modules are further discussed in detail.

The key components of Kubernetes are described as follows.

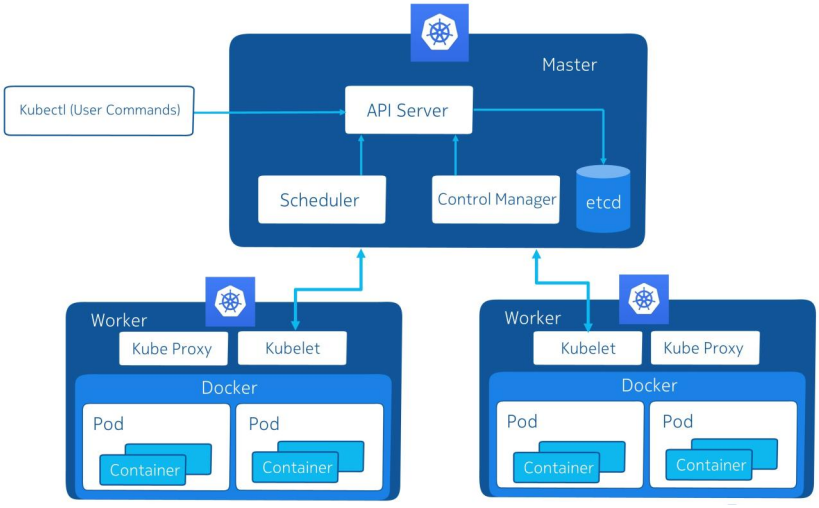

Figure 2.11: High-level architecture of Kubernetes.

- **Master and Worker Nodes**

  Following a master-slave type of architecture [71], Kubernetes consists of master and worker nodes. A node, in general, refers to the host device, which is either a virtual or a physical machine. Worker nodes run the pods and will be managed by the master node. The master node within a cluster is responsible for managing containers and consists of three processes, kube APIserver, kube controller manager and kube scheduler [72].

- **Pods**

  The smallest deployable unit of computing in Kubernetes is called pod [73]. Each pod consists of one or more application containers, which

are deployed on the same physical host, and share the same set of resources such as storage and networking. In other words, a pod models an application-specific *'logical host'* and includes different containers which are tightly coupled [74]. Hence, pod places and operates one level higher than the individual containers [75]. Kubernetes applies its scheduling and orchestrating mechanisms on top of pods instead of containers. This allows developers to deploy several containers out of their closely related micro-services and then package them into a single pod to act as a single application. A unique IP address would be assigned to each pod. While containers on the same pod can see each other on localhost, containers on different pods communicate using pod's IP addresses. Additionally, containers on a single pod share the same volume directories and resources. Hence, a pod resembles a virtual host machine including all resources needed for its containers.

- **Replication Controller**

  Replication Controller, is often abbreviated as *'rc'*, is responsible to make sure that the specified number of pods are always up and running, and if not, it will replicate required number of pods [76]. It also terminates some of the pods if too many of them are running. Each pod's replication is called a replica. Replicas will be managed based on the rules defined in the pod's template. A pod, which is created manually, can be evicted in case of any failure, while using a replication controller, we can define pods that will be replaced if they fail or be deleted for any reason. Hence, it would be a good practice to define a replication controller instead of manual pod creation. This will assure health of the application even if it contains only one pod.

- **ReplicaSet**

  ReplicaSet is an API object in Kubernetes, which manages scaling of pods. It checks state of pods and maintains desired number of them at any given time. According to the Kubernetes documentation [77], ReplicaSets are the next generation of Replication Controllers and provide more features.

- **Deployment**

  Deployment is a higher-level concept than replicaSet and pod. Defining a deployment, we can declare updates for Pods and ReplicaSets. In other words, we describe a desired state and deployment controller checks the actual state against the desired state. Instead of using pods

or ReplicaSets directly, it is recommended to define them by deployments.

- **Services**

  A service is an abstract way to expose applications running on the pods to the users. As mentioned earlier, a unique IP address is assigned to each pod within a cluster. However, since pods can be created or removed by replication controller, their IP addresses are not stable. Each newly created pod will receive a new IP; hence it would not be a suitable approach for users to connect to the applications via pod's IPs. This is where Kubernetes services solve the issue offering an endpoint API, which lets services being accessible externally. Moreover, assigning a single DNS name to each service, Kubernetes provides an internal service discovery mechanism and developers do not need to utilize an external approach for that [78].

2.8.2 Master Node Components

Master node, the controlling unit of Kubernetes cluster, is responsible to maintain and mange the state record of all objects in the system. To do this, it performs continuous control loops to respond to the changes.

Hence, the control plane is where users and system administrators interact with Kubernetes [72]. It will accept client requests and make sure to take the actual state of cluster towards its desired state as described by users via the Pod Lifecycle Event Generator (PLEG) [79]. The Kubernetes master node adopts a collection of processes, which manage cluster's state. These components are described as follows:

- **API Server**

  The API Server exposes a REST API, which makes communications of all cluster components possible. It also allows users to configure and validate all cluster objects such as pods, replication controllers, services etc. [80]. Furthermore, the API server handles communications between the master and worker nodes.

- **Controller Manager**

  A controller manager is a control loop, which runs on the master and using the API server checks the state of cluster. If any change arises, controller manger moves the current state of cluster towards the desired state. There are different controllers in the Kubernetes including

replication controller, namespace controller, endpoint controller and service-accounts controller [81].

- **Scheduler**

  Scheduler acts as a resource controller and handles the workload of a cluster. More precisely, it is responsible for assigning pods to the different worker nodes based on several metrics such as nodes computing resources, policy constraints of pods and quality of service requirements. To this goal, scheduler also keeps track of the general overview of resources to see which are free or occupied.

- **etcd**

  etcd is a lightweight, consistent, highly available and distributed key-value data store, which is used to store all cluster data including configurations and state information of cluster. To operate as a data-store, etcd acts based on the Raft consensus algorithm [82].

2.8.3 Worker Node Components

Worker nodes are the machines on which the application would be running. User often has no interaction with these nodes and master node controls each of them. Each worker node consists of a few components described as follows.

- **Container Runtime**

  As the first component, a container runtime is required be installed on each worker node. Container runtime is the software responsible for running containers. Several container runtimes are supported by Kubernetes; however, Docker is the most popular one.

- **Kubelet**

  Being responsible for management of pods and their containers in each node, makes Kubelet the most important component of worker nodes. Kubelet receives its instructions from master node and interacting with etcd, it updates the configurations. Based on acquired information, it makes sure that pods are healthy and are running properly. Then, it also reports status of nodes to the cluster.

- **Kube-Proxy**

  Kube-Proxy is a network proxy, which takes care of network rules on each worker node. These rules allow network communications to the pods from inside or outside of the Kubernetes cluster. In other words,

with mapping containers to the services and utilizing load-balancing mechanisms, it provides access to the deployed application from the external world. These network proxies work based on TCP and UDP streams.

Chapter 3

Design And Implementation

To study suitability of Docker and Kubernetes for deployment of the current IoT system, a practical proof-of-concept is conducted. This chapter explains the steps taken towards implementation of this proof-of-concept. The overall procedure is divided into five main sections.

First, the containerization of the system and the applied approach using Docker is described in Section 3.1. Section 3.2 studies the requirements for deploying the system using Kubernetes. A possible use-case scenario is described and the relevant architecture details and design decisions are discussed according to this use-case. General steps for setting up a Kubernetes cluster and detailed instructions are provided in Section 3.3. This includes guidelines for deploying the cluster, and follows with detailed explanations for installation phases in the master and worker nodes separately. The rest of required configurations to accomplish deployment of the system are discussed in Section 3.4. Deployments and service definitions, and also the used approaches for exposing the services are explained in this section. Finally, the required steps for deploying the Kubernetes Dashboard, which is a useful tool for monitoring state of the cluster, are explained in Section 3.5.

3.1 Nokia's IoT Pub/Sub System in Containers

Currently, the Pub/Sub IoT microservice-based system is deployed on different VMs. Since the system is designed based on the micro-service architecture, each of the services could be run separately on a VM. However, the current approach of deploying has two disadvantages.

Firstly, as discussed in Section 2.5.1, a VM needs to be booted up like a normal OS and this makes it slower compared to the containers. Secondly,

the team encountered still another issue in the process of deployment of the project on top of VMs. The system is developed with Java programming language. The first round of development and deployment has been completed on machines with CentOS as their operating system. When another team member tried to deploy the project with the same created 'Jar files' on other machines with Ubuntu operating system, it failed. The reason was the different Java versions on different machines, where jar files were built and were run. Also, the OS might have an impact in some cases. These are known as the *'dependency hell'* problem in software development and CI/CD workflows could be solved using containers. A containerizing tool, such as Docker, bundles up an application and it's dependencies together in a self-contained unit. Hence, a containerized application can be executed in any platform and has no dependency on the infrastructure.

All the above-mentioned problems motivated us to take a step further in the deployment and containerize the current system.

3.1.1 The System's Containerization Procedure

In this section, first, we present the current deployment architecture of the system. Then, the new containerized design and the steps taken for its implementation are discussed in detail.

The current deployment of the system is conducted on five virtual machines, where each is assigned to one of the services, Kafka-Zookeeper, Dashboard, Coordinator, Data Fetcher and Data Hub. The related architecture is shown in Figure 3.1. These VMs are managed by Nokia based on OpenStack, a free and open-source software platform for cloud computing [83]. Jar file of each component is built and runs on a single machine, and machines know each other and communicate via their IPs, which are defined in the configuration files of each module. To get rid of running jar files, which can be a platform-dependent approach, we aim to containerize these services.

Docker, as one of the most popular available technologies, is utilized for our containerization purpose. Due to the micro-service architecture of the system, making docker images out of each service is not so complicated. First, we make a Docker file for each service, and from the file, we create the Docker image. For Kafka and Zookeeper, we use the well-known public Docker images.

As the next step, to check if images are built properly and the system works upon containers, we run all containers on the same machine using Docker-Compose. Figure 3.2 illustrates the proposed architectural design of the containerized system.

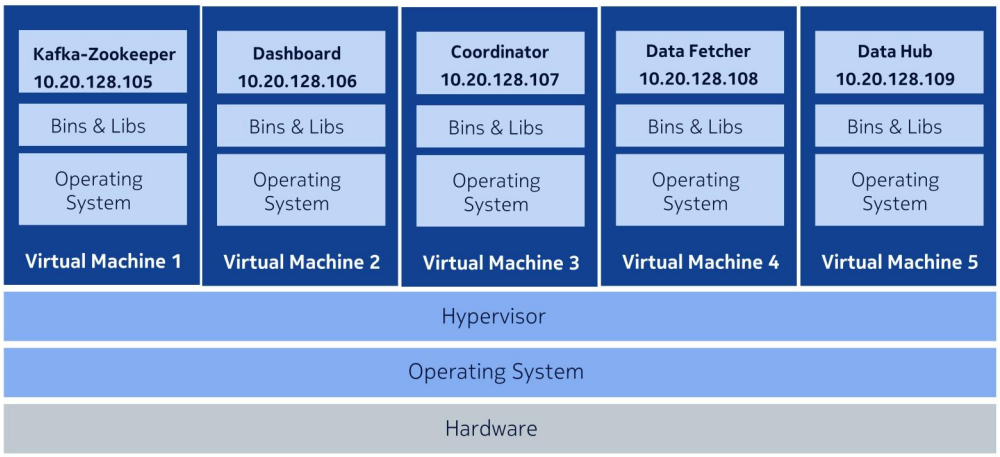

Figure 3.1: The current deployment of the system in VMs.

According to the Figure, Docker-Compose creates a single shared network, where all containers can join. Different services running on the same Compose network are reachable by other containers, and they are discoverable at a hostname defined by the container name [84].

To start Docker-Compose, we create a YAML file to configure our services. Image name, network name, the ports that should be exposed for the service, and hostname are among components, which should be defined in this file, for each service. Since our services are dependent on each other for their run times, we only include Kafka and Zookeeper services in the compose file.

Starting the Docker-Compose, we bring up Kafka and Zookeeper as is shown on the left side of the Figure. Then, we run each of other services separately on the same Docker-Compose's network. This way, we locate all of the system's micro-services under a shared network. Finally, we can access two of our services, which have a UI, namely *'Dashboard'* and *'DataHub'* on the localhost and their exposed ports.

3.1.2 Setup a Private Docker Registry

As described in Section 2.6.2, there are two ways to keep and share Docker images. Public and private Docker registries. For the former, there is a ready-to-use registry, where you only need to sign up and freely use the available images or share your own images. However, to use the latter, we need to set up it ourselves on top of Docker's open-source project, named 'Docker Registry'. Since Nokia's system is private, we cannot keep its corresponding

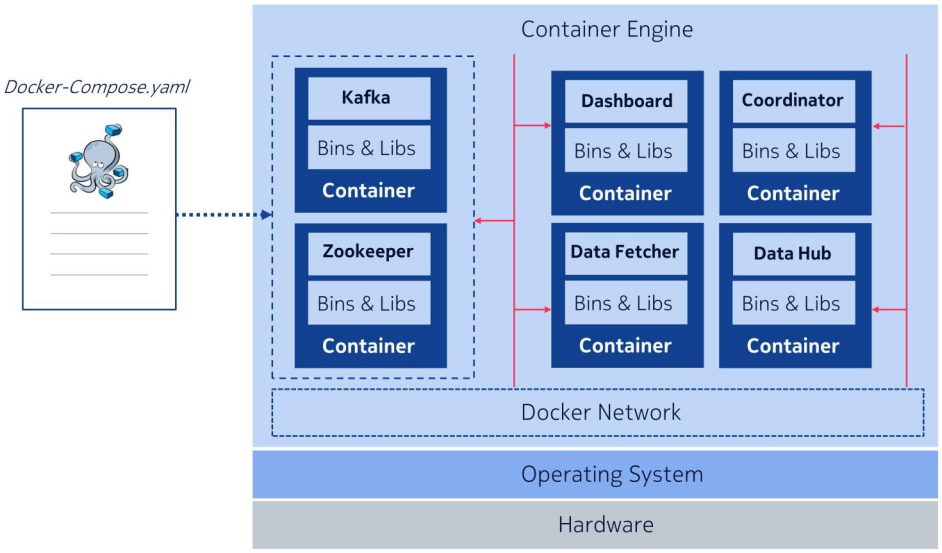

Figure 3.2: The architectural design of the containerized system.

Docker images in a public registry. Hence, in this section, we explain how we set up a private Docker Registry in one of the Nokia's servers.

The open-source Docker Registry acts as a storage and distribution system for Docker images and holds various versions of the same image, which are tagged differently [85]. The registry stores its data locally by default, hence, users can pull their images from its local filesystem and push new images to it as well.

It is possible to set up such a registry without any security restrictions. However, the better approach is to use the available mechanisms to make our registry secure. To this goal, we apply three main features that are provided by Docker. These features are explained as follows.

1. **Get a Self-Signed SSL Certificate**

   Without a certificate, we need to configure the client machines (machines which aim to use the Docker registry) and the Docker engine to trust our registry. There is a specific parameter, namely 'insecure-registrie', which should be defined for this purpose. However, we do not set such a parameter. Instead, we add our self-signed certificate to either the client machines or the Docker engine. Generating a certificate results in the creation of a public and a private key.

   We copy the corresponding public and private keys to the Docker engine. This way, Docker trusts our certificate. Later, to use the registry

from a remote client, we need to copy registry's certificate in that machine as well.

2. **Create Login Credentials**

   The next possibility to provide more security is to define credentials and make the Docker Registry password protected.

3. **Setup the Docker Registry**

   After adding SSL encryption support and defining basic authentication for the Docker Registry in the two previous steps, we can run the registry. The registry service will be run as a container itself. Also, another container, corresponding for UI of the registry, will be run separately. Users can interact and access data from the registry using their browsers. Instead of running each of these services separately, we use the Docker-Compose. Two different containers namely *'registry'* and *'registry_frontend'* are defined in the Docker-Compose configuration file as two separate services and then, are linked together.

3.2 System Deployment with Kubernetes

In this section, firstly, we describe a use-case scenario for deployment of the Pub/Sub IoT system. Next, the architecture for the deployment using Kubernetes is explained and the design decisions are discussed.

The micro-service architecture of the current system and containerizing each service separately, bring several advantages for the deployment, however managing multiple independent containers might cause even more challenges especially when we aim to scale the system. Ensuring network connectivity of the containers to each other and to the external world, constant monitoring of their state, resource optimization for each and finally, scaling the number of containers are from the challenges.

Making sure to meet all the requirements for managing containers, means to have an eye on hundreds of containers at the same time. This is fault-prone and difficult task. Kubernetes is a container orchestration tool, which will take care of managing containers and their connectivities. Using its self-healing mechanism, Kubernetes makes sure that all required services are always up and running. Furthermore, Kubernetes gives fine-tuned control over utilizing cloud resources for each of the containers. Finally, with Kubernetes, we can scale the system with more similar services with minimum effort.

Apart from the challenges that managing containers bring, deploying the system considering its current architectural state is challenging. Currently, the micro-services in the system are introduced to each other in a manual process. Each service has one or more configuration files, where the IP address of its related components are defined. This way each service knows the next destination that it should get connected. Deployment of such a system in scale, for different customers and in different infrastructures, demand for a long manual process, where we have to go through various files and change the defined IP addresses. Using containers, this means even more work since we need to rebuild the docker images after any change being applied to the configuration files of services.

Hence, eliminating this workload in the deployment is one of our biggest motivations to use Kubernetes in this thesis. Providing an internal service discovery, Kubernetes allows us to define names for the micro-services instead of IPs.

3.2.1 The Use-Case Scenario

For simplicity and keeping the consistency of the work, we explain one of the possible use-cases of Nokia's system. The rest of this thesis is explained based on this use-case.

The main goal of Nokia's IoT Pub/Sub system is to let one group of users publish their data, while another group subscribe their favorite ones from the available data catalogs. Such a general definition can be extended to a real business example. A company (Company X) generates some valuable data in its office. Then, it decides to make some profit out of such data using Nokia's IoT system. This company installs different IoT gadgets all around its office and data will be fetched from these devices using the *'DataFetcher'* module. Other business units (Company Y) could become its customers and be willing to use this data. They only need to have access to the *'DataHub'* module of the system.

A general overview of such a use-case is shown in Figure 3.3. Keeping this business use-case in mind, we deploy the system using Kubernetes.

3.2.2 System Architecture and Design Decisions

The current system is developed in a way to benefit from capabilities offered by MEC architecture. To be more precise, the Data Fetcher module is supposed to be established on an IoT gateway. An IoT gateway acts as the connection point between IoT devices and the cloud. The pre-processing and aggregation of data can be executed locally at the edge before sending it to

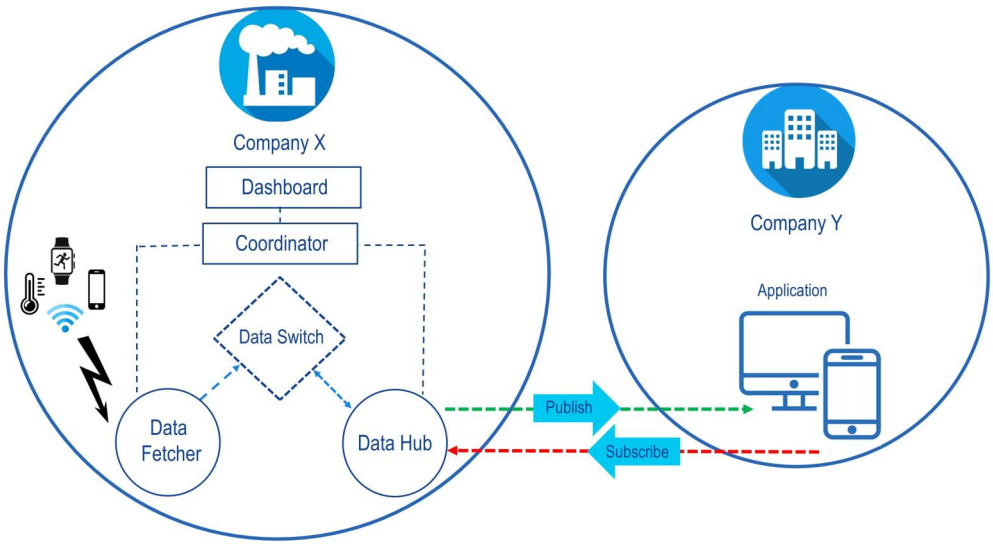

Figure 3.3: General overview of the use-case scenario.

the cloud. This will reduce the volume of transmitted data as well as the network overhead [86]. Hence, it is reasonable that a gateway be placed close to where IoT devices reside and it is supposed to have some computation power as well as network connectivity.

To deploy the system considering such an architecture, the host infrastructures for Data Fetcher and the rest of the modules are separated. However, we do not use an actual gateway in the scope of this thesis. Instead, we have used a separate machine that is assigned specifically to the Data Fetcher. This way, we study and showcase the possibility of such a scenario with Kubernetes, while it can be later followed by other settings and equipment.

Moreover, the host infrastructure for the Data Hub module is also separated. The reason for such a design is the way we expose this service for the end-user. More details are provided in the next section, which explains service definitions.

To implement this proof-of-concept, we have used Nokia's private cloud infrastructure; Nokia Engineering and Services Cloud (NESC) [87]. NESC, being based on the OpenStack, provides a fully functional IaaS cloud. A general architectural overview of the proposed scenario using Kubernetes is demonstrated in Figure 3.4. According to the architecture, we have four virtual machines, which together form our Kubernetes cluster. One of the machines acts as the master node and the three others are the worker nodes. We deploy each of the different modules of the system as a separate service

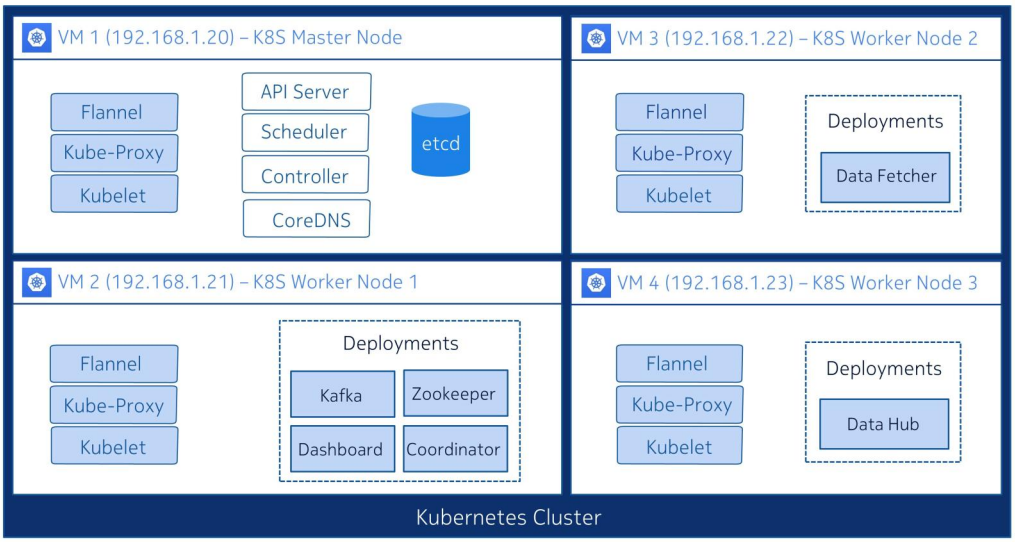

Figure 3.4: Architectural overview of the proposed deployment scenario with Kubernetes.

within the Kubernetes cluster. Hence, we have services including Kafka, Zookeeper, Dashboard, Coordinator, all deployed in the *'Worker Node 1'*, the Data Fetcher in the *'Worker Node 2'* and the Data Hub in the *'Worker Node 3'*.

In Figure 3.4, processes related to the configuration of the Kubernetes cluster can be seen as separate white and blue boxes in each machine. The white boxes show processes, which are specifically for Kubernetes master node, while the blue boxes show the common Kubernetes processes in all machines. These processes are previously described in Section 2.8. Furthermore, each machine has its own Kubernetes Deployment(s), and they are represented in blue boxes all within another box with the title 'Deployments'.

3.3 Setting Up the Kubernetes Cluster

All steps required for the deployment of our proof-of-concept scenario are described in this section. First, the conducted procedure for setting up a Kubernetes cluster is explained. This includes different instructions for master and worker nodes, and each is discussed separately. After that, we explained all configurations needed to deploy our micro-services on the different nodes in a Kubernetes Cluster.

3.3.1 Kubernetes Setup

There are several approaches to set up and start with Kubernetes. Minikube [88] is one of the easiest ones. It runs a single-node cluster inside a VM and helps users to install Kubernetes locally. It might be the best and fastest suggestion for those who want to try out Kubernetes for the first time. However, Minikube is not a production-ready tool and is developed for specific use-cases. Moreover, creating only one single node, Minikube does not allow us to explore the required conditions for our scenario. Hence, this tool is not the most suitable approach for us. Instead, we will use Kubeadm to set up a single control-plane cluster. We can install Kubeadm on any type of device either a server or a gateway with different architecture [89]. This makes it very well-suited for our scenario.

To start the implementation, we create four virtual machines on the NESC cloud environment. Since we would need to access these machines from the outside world, we assign each machine a public IP. Then, to make our machines ready to act as nodes in the Kubernetes cluster, we need to apply some prerequisites on all the nodes. This procedure starts with setting a unique hostname for each node and also adding IPs and corresponding hostname of each node to the other ones. Next step is to open specific ports on the nodes. To do so, we need to add rules to the established firewall system. The list of ports, which must be opened, is different for master and worker nodes. As the other requirements, the swap memory for all Linux machines must be disabled. Moreover, since we are using CentOS operating system, the SELinux, which is active by default, must be disabled.

After making sure of applying these perquisite on all nodes, is the time for installing a container runtime, which for Kubernetes is Docker by default. Docker version *18.09*, which is the most stable version compatible with Kubernetes at the time of writing this thesis, is chosen and installed on all machines. Finally, we install Kubernetes on all machines. This includes installing three different tools including Kubeadm, Kubelet, and Kubectl [90]. Kubeadm is the tool, which allows us to bootstrap a cluster in the master node and to join the created cluster in the worker nodes. Kubelet, which is described in Section 2.8.3 and introduced as one of the important processes being run on each worker node, is used to start pods and containers. Kubelet runs on all nodes of the cluster. Lastly, Kubectl is the command-line tool for users to interact with the Kubernetes cluster. These tools should have the same version to prevent any version skew occurring.

3.3.2 Master Node Installation

Upon having Kubeadm installed on all nodes, we can initialize our control-plane node. Before explaining the initialization process for control-plane, we will explain some prerequisite concepts as follows:

- **Cluster Networking**

  Based on the networking model in Kubernetes, each pod gets a unique IP. This way, pods on one node can communicate with all pods on the other nodes. Since IPs are assigned to pods, all containers within a pod share the same IP and communicate with each other using localhost. In Kubernetes, this is called the *'IP-per-pod model'* [91]. Kubernetes does not provide a default network implementation. Instead, it only defines the model and other tools must be used for the implementation. There are multiple ways to implement such a networking model for Kubernetes. These solutions are offered by external network plugins and interfaces, also known as network add-ons. The Kubernetes Add-ons are described next.

- **Kubernetes Add-ons**

  In general, add-ons are used to extend the functionality of Kubernetes [92]. Different add-ons are served for different purposes. Network plug-ins such as Flannel or Calcio, DNS service managers such as coreDNS, and Kubernetes Dashboard, which is Kubernetes UI, all are examples of Kubernetes add-ons. While installing some of these add-ons is a must to run a Kubernetes cluster properly, there are several optional ones that provide additional services. Installing a network add-on is one of the required steps in the Kubernetes setup. There is a long list of such add-ons listed in [92] and we choose Flannel for our work.

- **Flannel**

  Flannel creates a virtual network, which runs over the host network, and is called an overlay network. This network is responsible to assign unique IPs to the pods. It does this running a small agent called *'flanned'* on each host node. This agent allocates a unique IP subnet (by default /24) to each node from a larger address space. Then, all pods within a node use an IP from this range. Using these IPs, pods can communicate with each other in a cluster. To explain this further, we can refer to Figure 3.5, which shows the overlay network architecture and IPs assigned. All our VMs are created in the network subnet *192.168.1.0/24* and they have been assigned IPs within this range.

Flannel overlay network has a default subnet range *(10.244.0.0/16)*, which must be stated while initiating the control-plane. From this large subnet, flannel assigns another smaller subnet to each host node. As it can be seen in the Figure 3.5, master node is assigned with subnet *10.244.0.0/32*, worker node 1 is assigned with *10.244.1.0/32* and worker node 2 has the subnet *10.244.2.0/32*. Next, each pod within a node will be allocated with a unique IP from its host node's IP subnet. The containers within one host node can communicate with each other using the docker bridge *docker0*. Moreover, flannel's architecture makes the cross-node pod-to-pod communications possible. This means that pod 1 with IP *10.244.0.1* can see pod 6 with IP *10.244.2.2* through this overlay network.

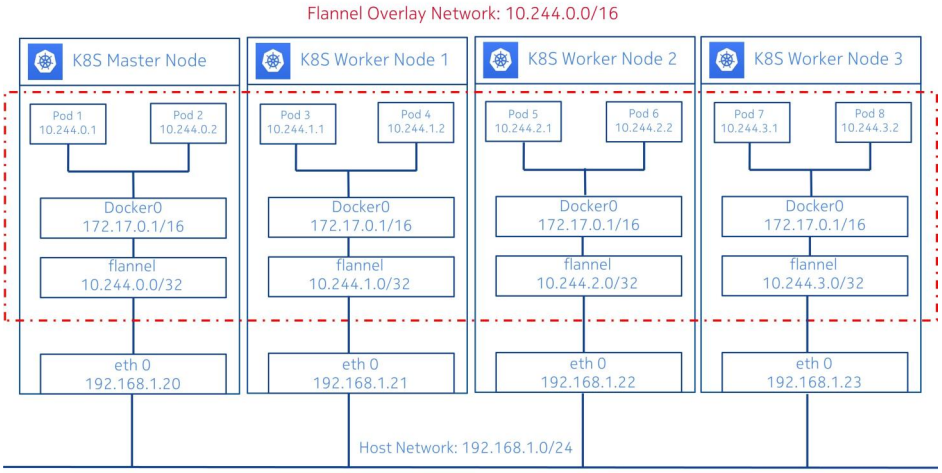

Figure 3.5: Overlay network architecture in Flannel.

- **DNS manager - CoreDNS**

  Service discovery is the process of automatically detecting available services on a network. Kubernetes has a built-in DNS service that will be launched automatically by the add-on manager. This internal service discovery is one of the main motivations that we chose Kubernetes for the deployment of our system. It will let us connect micro-services of our system by names instead of IPs.

  As of Kubernetes version *1.12*, CoreDNS is the default DNS server being used with kubeadm. CoreDNS is one of the Kubernetes add-ons that monitors the Kubernetes API and automatically creates DNS records for any new services. Services are the objects that get a DNS

name in a Kubernetes cluster. This enables service discovery across a cluster and pods are able to perform name resolution automatically for all services. According to this definition, we need to deploy each of our components as a service in the Kubernetes cluster. This way, all services across a cluster can reach each other by name, and it does not matter on which host node they are located.

To initiate cluster on the master node we issue the *'kubeadm init'* command. Since we aim to use flannel add-on as the cluster networking, we define the subnet that flannel must use to create its virtual overlay network, which is *10.244.0.0/16*, while defining *'–pod-network-cidr'* parameter.

This will bring up all components of a master node, described in Section 2.8.2, including etcd, apiservice, controller-manager, kube-proxy, kube-scheduler and coreDNS. However, CoreDNS will not start up before we install a network add-on. Hence, the next step is to install flannel applying its specific configuration file on our cluster. After this, listing pods in the cluster, we should see all these modules are up and running.

3.3.3 Worker Node Installation

While the master node is the control-plane that manages all interactions of a Kubernetes cluster, worker nodes are where the workloads such as containers, pods, and services run. Now that we have our cluster with a master node ready, we must add worker nodes to the cluster. Upon initialization, the master node issues a join command. Running this command in our worker nodes will add them to the cluster.

After running this command on all the worker nodes, we can see all nodes are in the *'ready'* state.

3.4 Deployment and Service Definitions

Now that we have our cluster ready and all nodes are up and running, it is the time to deploy our system on top of the Kubernetes cluster. First, we need to define a pod per component. However, instead of defining pods directly, as it is suggested by Kubernetes documentations [93], we use Deployments. Using deployments, we can describe the desired states, such as scaling, for a pod and the deployment controller takes care of applying it.

Next, we need to define each component of our system as a service so that we can benefit from the internal DNS service of Kubernetes. This way, Kubernetes will automatically implement the service discovery that is required

in the micro-service architecture of our system. Moreover, through exposing services to the external world, we will give access to the end-users to the corresponding UIs of the system.

The details for the definition of services and deployments for each component of the system are described as follows.

3.4.1 Common Fields in the Deployment and Service Definition

The creation of different Kubernetes objects, such as deployments and services, is done by Kubernetes API. The request body that is sent to the API, must have a JSON including all the required information for creating an object. However, we do not use the API directly. Instead, we provide the information in a YAML file and use the kubectl tool to deploy objects. Kubectl converts the information described in the YAML file to the JSON format and makes the API request.

The YAML file for defining Kubernetes objects includes several required parameters. Generally, the file starts with *'apiVersion'*, which defines the Kubernetes API version that we use to create the object. *'kind'* refers to the object type that we aim to create such as Deployment, Service, Job, pod or else. Then, a name will be given to the object using the *'metadata'*. We have named all deployment objects with a *'-dep'* suffix and all service objects with a *'-svc'* suffix after their main names. For example, we use *'zookeeper-dep'* and *'zookeeper-svc'* in the YAML file definitions.

Finally, the *'spec'* illustrates the building components of the object. The precise format of it, is different for various Kubernetes objects. This format is explained in detail for deployment and a service object separately in the following sections.

3.4.2 Deployment Definition

The *'spec'* definition for a deployment object includes categories such as *'replicas'*, *'selector'* and *'template'*. Explanation of each and what we have set for our Deployments are discussed as follows.

- **spec.replicas** Using this parameter, a Deployment controller allows us to specify the number of replicas (pods) that should always exist in the cluster. We give the value of '1' to it in all our deployments.

- **spec.selector** Using selector, the deployment finds the Pods that it has to manage. A label will be assigned to the Pods. Assigning the

corresponding label using *'matchLabels'*, we let the deployment to find its related Pods.

- **spec.template** It has two sub-categories:
  - **metadata** Using *'metadata'*, we label Pods with *'metadata.labels'*. In our work, we label all Pods in the format: *'app:name'*. For example, in case of *'Zookeeper'*, we have *'app:zookeeper'* under the field *'template.metadata.labels'*.
  - **spec** The template's specifications will be defined under this parameter. Spec might include different sub-categories itself. Among them, *'spec.containers'* and *'spec.nodeSelector'* are defined for all of our deployments. The *'spec.imagePullSecrets'* parameter is required to be defined for those, which do not use a public container image.
    - **containers**
      name of the container, the image that it needs to use, corresponding ports that should be exposed to the container and environmental variables are among the information that will be provided under this field. However, specifying the ports that should be exposed is informative and not specifying a port here does not prevent that port from being exposed.
      For Zookeeper and Kafka, there are pre-built and public Docker images that we use in our project. Among the available images, we chose *'digitalwonderland/zookeeper'* and *'wurstmeister/kafka'*.
      Some important environment variables that should be mentioned in the 'kafka-dep' includes *KAFKA_ADVERTISED_HOST_NAME* and *KAFKA_ZOOKEEPER_CONNECT*. As the value for the former, we give name of Kafka service object and the latter should be specified with the name of Zookeeper service object and its used port in the form *Service name:Port*. This is where we avoide specifying IPs and instead, we use name of services. Kubernetes CoreDNS will take care of resolving the corresponding IPs for these names.
      For the rest of micro-services, information under this field are specified with name of corresponding images that we previously have built and ports that they use. It is worth mentioning that docker images of Dashboard, Coordinator, DataFetcher and DataHub components are re-built and are different from

the ones that we used for running the system locally with Docker. This is because, all IPs are removed from the corresponding configuration files and are replaced with name of Kubernetes services. This eliminates the repetitive phase of changing IPs and re-building images according to the different deployment infrastructures.

* **nodeSelector**
  spec.nodeSelector specifies a map of key-value pairs. By using it, we constraint running of our Pods to a specific node within the cluster. If we are not aiming for such a constraining, we do not need to specify this parameter at all. Then, Kubernetes will automatically take the responsibility of distributing Pods to the nodes based on their workload.
  Before using this nodeSelector, we should make sure that nodes are assigned a key-value pair as their labels. In Section 3.2.2, we defined the required architecture for our Kubernetes cluster and to follow it, we need to constraint each of our Deployments (Pods) to its corresponding node. As explained before, Zookeeper, Kafka, Coordinator and Dashboard must be run on the *'Worker Node 1'*. Data Fetcher on *'Worker Node 2'* and Data Hub on *'Worker Node 3'*. Hence, as an example, the field *'spec.nodeSelector'* for the Dashboard Deployment that has to be run on the *'Worker Node 1'* is defined as: *'dedicated:worker1'*. The same procedure is taken for the other Deployments corresponding to the architecture.

* **imagePullSecrets**
  Kubernetes will automatically pull the specified image if it is public. This works for Zookeeper and Kafka, but as mentioned in Section 3.1.2, for the rest of our components we use private Docker images.
  To let Kubernetes access our private Docker registry, we need to create a *'secret'* for Kubernetes based on the registry's login credentials. Using the *'spec.imagePullSecrets'*, we define such a secret in the Deployment definition for the corresponding image.

3.4.3 Service Definition and Exposing Services

Defining deployments, Pods can be created and destroyed dynamically. This causes Pods to be changed frequently and hence, they will be assigned different IPs every time. This way, the Pods cannot keep track of each other and provide functionality to the other Pods. To overcome this issue, we use Services in Kubernetes. A Service is an abstraction that defines a set of Pods and a policy that they can be accessed through. As explained in Section 3.3.2, using an add-on, which currently is CoreDNS, Kubernetes setups a DNS service for the service discovery purpose. This way, Kubernetes implements an internal service discovery mechanism. The Kubernetes cluster monitors Kubernetes API for new services and creates a set of DNS records for each of newly created services. Each service will be assigned a unique IP address and a DNS host name. Moreover, an internal load balancer on top of a service will take care of load balancing traffic to the service's corresponding Pods.

A service definition starts with *'apiVersion'*, *'kind'* and *'metadata'*, which are explained as the common parameters for all object's definitions before. Hence, we go straight to the *'spec'* definition. The *'spec'* for a service object includes categories such as *'selector'*, *'ports'*, *'type'*, and *'externalTrafficPolicy'* and each is discussed as follows.

- **spec.selector** Assigning a key-value pair as the label of pods, this field specifies set of Pods that the service should target. Then, a continuous scan will be done by the service selector controller to find those Pods, which their label matches this definition. Upon finding such matches, the controller POSTs the recent updates to them. As an example the *'spec.selector'* for the Zookeeper Service is defined as *'app:zookeeper'*. The same procedure is followed for all other service objects.

- **spec.ports** Specifying a port or ports, we can expose the service internally within the cluster on such port(s). In other words, the service becomes visible on this port for the other services. Hence, the requests that are sent to this specific port, will be forwarded to the pods that are selected by the service.

  Furthermore, there are services that work with more than one port and are called *'Multi-port Services'*. In our system, Dashboard service is of this type and we need to expose more than one port for it. Each ports definition includes some sub-categories such as *'ports.name'*, *'ports.protocol'* and finally *'ports.port'*. We can assign any name to the port and the used protocol such as TCP or HTTP will be defined

under *'ports.protocol'*. Some of our micro-services do not expose any port publicly, such as Coordinator and Data Fetcher. In the service definition of such components, we do not need to define any port.

- **spec.type** Service type specifies how we aim to publish each service. Some of our services only need to communicate with the other services, while some of the services should be exposed to an external IP and be accessible from outside of the cluster. According to the publishing approach, we have to define the service type. Possible types are explained as below:

  - **ClusterIP** Using ClusterIP, we can expose a service on an internal IP within the cluster. In other words, we do not give any external access to the service and it will be accessible only from within the cluster. If we do not specify any service type, the service will be assigned with the *'ClusterIP'* type by default. For Zookeeper and Kafka, we do not define any type and they will be assigned a *'ClusterIP'* service type by default. This is because we do not need any external access to these services from outside of the cluster.

  - **Headless Services** Defining a service, Kubernetes assigns an internal IP address to it by default. Then, through this IP, the service will proxy and load-balance the requests to the service's Pods. To define a 'Head-less' service, we assign value of 'None' to this IP address, defined by 'ClusterIP' service type. This way, it tells Kubernetes that this service does not need proxying or load balancing over Pods. Instead, it is enough that Kubernetes just route traffic to the first available Pod.

    Two of the micro-services, namely Coordinator and Data Fetcher, do not use any port and we cannot define any port in their service definition. To define services for such modules we need to define them as *'Headless'* services.

  - **NodePort** Finally, we have the NodePort as a service type that allows us to expose our service on an external IP for accessing from outside of the cluster. NodePort exposes the service on the Public IP of all nodes and a static port that is specified in the service definition. To specify such port, we need to define an extra parameter under 'spec.ports' that is named *'nodePort'*. Hence, such a service will be exposed externally over the address: *NodeIP: NodePort*.

Two of our micro-services namely Dashboard and Data Hub, which have their corresponding UIs and are required to be exposed externally, are defined from this type. For a service from this type, if we do not define the *'spec.ports.nodePort'*, Kubernetes automatically assigns a port that the service can be exposed to it. However, the assigned port would be different in each run time of the service. Since we aim to give access to these services to the end-user, it is reasonable to define the ports ourselves.

The default port range that Kubernetes has defined for this purpose is between *'30000 –32767'*. According to this, we had a challenge for our use-case. A specific port is hard-coded in the definition of the Dashboard UI's front-end code and for that reason, we need to expose the Dashbord service on the same port that is used in the code. To solve the problem, we have changed the Kubernetes's default port range for NodePort. After applying this change, we are able to expose the Dashboard and Data Hub services on their specific ports.

- **LoadBalancer** NodePort is mostly used as the service exposure solution when Kubernetes cluster is built on-premise. Using loadBalancer service type is another possible option if we deploy the Kubernetes cluster on top of a public cloud provider's infrastructure, which supports external load balancers. Defining a service from the type of loadBalancer, Kubernetes sends a request to the cloud provider to provision a network loadBalancer for the service. Upon receiving the request, the cloud provider automatically deploys a load balancer to route traffic to the service ports and return that information to Kubernetes. The loadBalancer has an external IP that can be provided to users to access the services. However, a separate loadBalancer is required for exposing each new service. Hence, scaling can inversely affect the cloud bill.

  A loadBalancer can also be supported by private clouds, such as OpenStack. However, activating a loadBalancer on OpenStack requires more configurations and settings since a loadBalancer would not be assigned by the cloud provider automatically.

- **Ingress** We can also use Ingress for exposing services. However, this is not a service type. Ingress is an API object in Kubernetes that routes HTTP and HTTPS traffic from outside of the cluster to the services within the cluster. The traffic routing can be

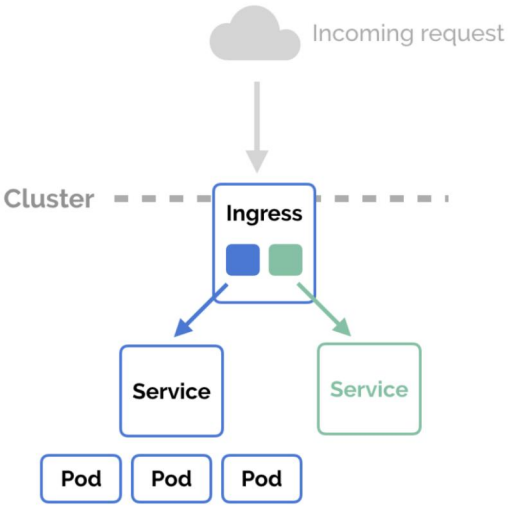

Figure 3.6: Using Ingress for exposing services in Kubernetes [94].

managed by defining traffic rules on the Ingress resource [95]. In other words, Ingress is a single entry-point to the cluster and acts as a reverse proxy. Figure 3.6 illustrates the relation of an Ingress with a Kubernetes cluster.

An *'Ingress Controller'* is required to satisfy an Ingress in Kubernetes. One of the possible options is to use NGINX Ingress Controller. The main function of Ingress is to route requests and provide externally-reachable URLs for services. However, other functionalities can be served behind an Ingress, such as load balancing traffic or handling SSL/TLS termination.

- **spec.externalTrafficPolicy** After going through the explained procedure for publishing our services externally, we still have another challenge. As explained in the previous section, the NodePort exposes our service on the public IP of all nodes of the cluster and the defined port. In other words, to access either of Dashboard or Data Hub services, we use any of the node's IPs.

  However, this is not exactly what we are aiming for. According to the definitions of micro-services of our system in Section 2.4.2, the Dashboard is a UI for monitoring state of the system including all micro-services and the data that is transferring among producers and consumers. Hence, access to this service should only be given to the company X in the described use-case (The company that we aim to deploy the system there and owns the data sources). This is while

the UI for the Data Hub service is designed for consumers of data, meaning the company Y in the described use-case. Therefore, we need to expose these services in a way that each is reachable on a specific IP and port. As a solution, we define the *'spec.externalTrafficPolicy'* field in the service definition of these two services and assign *'Local'* as its value. Assigning such value, Kubernetes only proxies requests to the local endpoints and never forwards traffic to the other nodes of the cluster. This way, Kubernetes preserves the client's original source IP address.

Finally, the end-user in company X would be able to access the Dashboard service on the *'Worker Node 1's IP:Port'*. Also, the end-user in company Y has access to the Data Hub service on the *'Worker Node 3's IP:Port'* using a VPN to the company X's internal network. Figure 3.7 shows an overview of our approach for exposing Dashboard and Data Hub services externally.

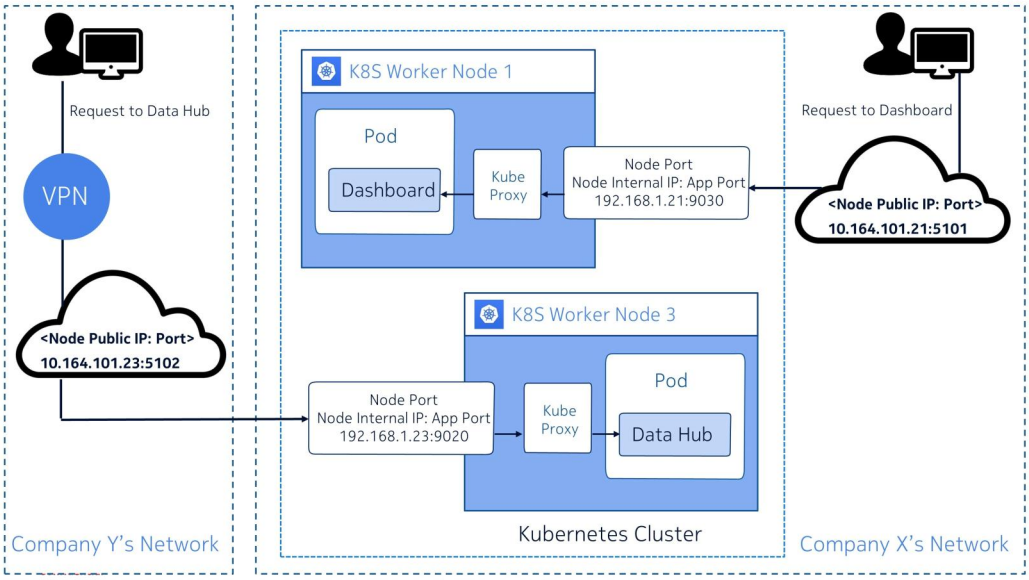

Figure 3.7: General overview of exposing services.

3.5 Kubernetes Dashboard

Kubernetes has a web-based UI, called Dashboard. In general, we use it to get an overview of running applications on the cluster. It provides a

visual perspective of the cluster, its nodes and all deployed services and containerized applications. Moreover, we can monitor the resource usages by different components of the system and troubleshoot possible problems. It is also possible to scale deployment or initiate the rolling updates through Dashboard.

While creating a Kubernetes cluster, Dashboard is not initiated automatically. We deploy it separately applying its specific configuration file on our cluster.

The usual way to access Dashboard, is to use kubectl command-line tool and running command *'kubectl proxy'*. This gives us access to the Dashboard UI on the localhost. However, since we are using CentOS in our host machines, we cannot see the UI locally and we need to expose the Dashboard service externally. Exposing a service, as will be explained later, will give us access to that service externally from outside of the Kubernetes cluster. To do so, we edit *'Kubernetes-Dashboard Service'* to use *'NodePoet'* instead of *'ClusterIP'*. This will expose the Dashboard service to one of the cluster nodes' IPs and a specific port. Hence, we check to see on which node Dashboard is running. Then, we also checklist of available services to find the port that this service is exposed on. Finally, we can access Dashboard using this *IP:PORT*. The UI of the Dashboard is shown in Figure Figure 3.8.

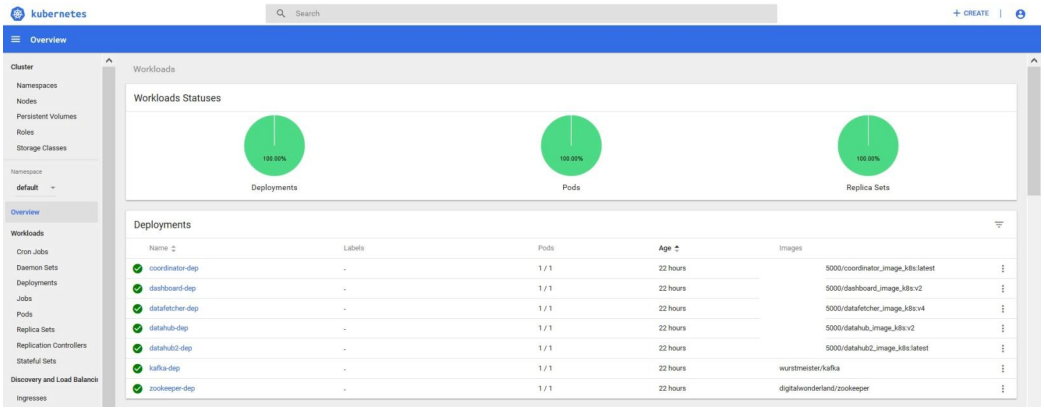

Figure 3.8: Overview of Kubernetes Dashboard.

By default, the Dashboard will be deployed with minimum user role privileges. This is due to protecting cluster's information. To find admin access to Dashboard, we need to define a user with such permissions. We create the user using the *'Service Account'* mechanism of Kubernetes. Upon initializing a cluster using kubeadm tool, the *'ClusterRole'* of *'admin-Role'* already exists in the cluster. We only need to bind it to the user. So, to combine these two steps, we define a YAML file to create a ClusterRoleBinding for our

ServiceAccount. We name this file *'dashboard-adminuser.yaml'* and apply it to our cluster. Currently, Kubernetes only supports login to the Dashboard via a Bearer Token. Issuing a specific command in Kubernetes, we can create such a token for our created admin-user.

Chapter 4

Test and Evaluation

In this chapter, the proposed solution, to deploy the IoT system using Kubernetes, is evaluated based on different metrics. The focus for choosing metrics and designing test scenarios is towards the deployment process. Hence, some sections include measurements, while some are discussion-based and focus on show-casing advantages of the used method for deployment.

The discussed metrics include performance both in the runtime and the cost efficiency, lifecycle management including system updates and fault-tolerance of the system, resource management and scalability of the solution. Moreover, a comparison with the previous deployment solution (where Kubernetes is not used, and each micro-service is run in a separate VM) is conducted for some of the discussed parameters.

4.1 Experimental Setup

The experimental setup for the proposed proof-of-concept is explained in this section. The deployed Kubernetes cluster includes one master node and three worker nodes. The VMs, which are hosted for the Kubernetes cluster, are created on top of Nokia's private cloud.

Kubernetes version *1.14.3*, Docker version 18.09 and Docker-compose version *1.24.0*, as the most stable versions at the time of writing this thesis, are installed on all four machines.

Table 4.1 illustrates the hardware characteristics of each VM and its public and private IPs. (To preserve privacy of the company, real IPs are replaced with example ones.)

VM Name	OS	Hardware Characteristics			HostName	IP	
		CPU	RAM	DISK		Private	Public
Master Node	CentOS 7.6.1810. x86-64	2	6 GiB	32 GiB	k8s-master	192.168.1.20	10.164.101.20
Worker Node 1	CentOS 7.6.1810. x86-64	2	6 GiB	32 GiB	k8s-worker1	192.168.1.21	10.164.101.21
Worker Node 2	CentOS 7.6.1810. x86-64	2	6 GiB	32 GiB	k8s-worker2	192.168.1.22	10.164.101.22
Worker Node 3	CentOS 7.6.1810. x86-64	2	6 GiB	32 GiB	k8s-worker3	192.168.1.23	10.164.101.23

Table 4.1: Hardware components used in the proposed architecture.

4.2 Performance Analysis

The performance of the proposed deployment solution is evaluated based on two metrics: The time, which it takes to issue a subscription in the Pub/Sub IoT system and the cost management for the deployment approach. In both cases, we have compared the results with the previous deployment solution, where Kubernetes is not used. The conditions for each test case are discussed in detail in the following sub-sections.

4.2.1 Runtime Speed

The time that it takes to issue and accomplish a subscription is measured in the current and the previous deployment solution. By the previous deployment solution, we mean where containerization and Kubernetes, as the orchestration tool, are not used, and we simply run the corresponding 'Jar file' of each micro-service in a separate VM. According to the behavior of the system, before the actual data transmission, consumers must subscribe to the data that they are interested in. More precisely, Data Hub service acts as a data catalog for consumers where they can see the available data and issue subscription. Upon issuing a subscription, there would be some message communications between Data Hub, Coordinator, Kafka and Data Fetcher.

Currently, we are provided with data from two different data sources. We have repeated the subscription for both data sources and repeated the test five times for each. The results are calculated as the average of these tests.

State	Average Subscription Time (S)
Previous Deployment Setup	19.2
Current Deployment Setup	9.8

Table 4.2: Runtime Comparison between the previous and current deployment setup.

Results show an average of 19.2 seconds as the time that takes for completing a subscription in the previous deployment solution. Using the proposed deployment solution, this time has reached 9.8 seconds on average. The acquired results are presented in Table 4.2. We can interpret such a decrease (about 9.4 seconds) as a result of the designed architecture in the two solutions. In the previous approach, each micro-service is running in a separate VM. Hence, all traffic and communications between them are external. This is while, we have three of these micro-services namely Dashboard, Coordinator, and Data Switch resided on the same VM in the proposed solution. This eliminates part of the external traffic since communications among these three services are local. Finally, it results in faster response time for a subscription accomplishment.

4.2.2 Cost Efficiency

One of the important factors for a deployment solution is its overall cost requirements. Ultimately, we aim to deploy such a system for different customers, and the cost-efficiency of the proposed solution becomes notable. Especially when it comes to scaling the system, for instance with more Data Hub services. Considering the technical requirements, we have calculated the overall cost for the proposed deployment approach. Also, a comparison with the previous solution is presented in case of cost management.

Azure cloud is considered as the provider of the required infrastructure for deploying such a system. The minimum hardware requirements for setting up Kubernetes on a VM includes 2 cores CPU and 2 GB RAM [96]. Hence, we need to choose machine instances that at least have these characteristics. The smallest unit that provides our demand in Azure cloud is the 'B2S' instance that has 2 cores CPU and 4 GB RAM. As explained in Section 3.2.2, for the proposed solution we would need four machines, where one is specifically for the master node of the cluster and three other machines acting as the worker nodes to host our micro-services. Among the worker node machines, one is for hosting Data Fetcher service, one for Data Hub service, and the last one will include Zookeeper, Kafka, Coordinator, and Dashboard services.

If we consider the previous deployment solution, we would need five machines, where each includes one of the micro-services (Kafka and Zookeeper reside in the same machine). If we go further, we have suggested a better way of exposing services in Kubernetes in the future works, Section 5.2. Using that approach, we would not need an additional VM for the Data Hub service and hence, only three VMs would be enough. The monthly cost for maintaining machines using these three different approaches is presented in Table 4.3.

Solution	Number of Instances	Instance Type	Monthly Price ($)
Current Solution	5	B2S (2 Cores, 4 GB RAM)	164.25
Proposed Solution with NodePort	4	B2S (2 Cores, 4 GB RAM)	131.40
Future Work with Ingress	3	B2S (2 Cores, 4 GB RAM)	98.55

Table 4.3: Deployment cost in the proposed solution with Kubernetes [97].

According to Table 4.3, the price difference for the previous and current deployment approaches is 32.85 $ in a month. Comparing the previous approach and possible edition of the current solution as the future work, even shows more difference, about 65.7 $ in a month. This price difference might seem negligible. However, if we think of the scalability of the solution, where we would need to have more Data Hub services, the difference becomes significant.

Node	Instance Type	Monthly Price ($)
Master	B2S (2 Cores, 4 GB RAM)	98.55
Worker Hosting Data Fetcher	B2S (2 Cores, 4 GB RAM)	98.55
Worker Hosting the Rest	B4MS (4 Cores, 16 GB RAM)	132.86

Table 4.4: Monthly price for a scalable solution using Kubernetes and Ingress [97].

As discussed in the implementation chapter, Section 3.4.3, in this thesis work, we exposed the Data Hub service externally using NodePort. Because of that, we need one VM per Data Hub service. However, in future work, Section 5.2, we discuss a better way of exposing services using Ingress. Doing so, we can deploy multiple Data Hub services on only one VM. In other words, we can deploy the whole of the services, except Data Fetcher, all in one node. This eliminates the need for purchasing lots of VMs and has a huge impact on cost management. However, we would need a VM with more computation and I/O power to host all services. For this reason, an instance of type 'B4MS' with 4 cores CPU and 16 GB RAM is chosen. For the rest of the machines, we chose the same type as before, B2S. It results in 329.96 $

in total as the monthly cost for this scenario. The corresponding details are shown in Table 4.4.

We have calculated the total prices considering these three while scaling the number of Data Hub services. The results are shown in Figure 4.1. The figure also gives an overview of the total impact of this scenario on the cost management. It is visible that a deployment solution with Kubernetes in future works, where Ingress is used for exposing services, supports both scalability and managing costs.

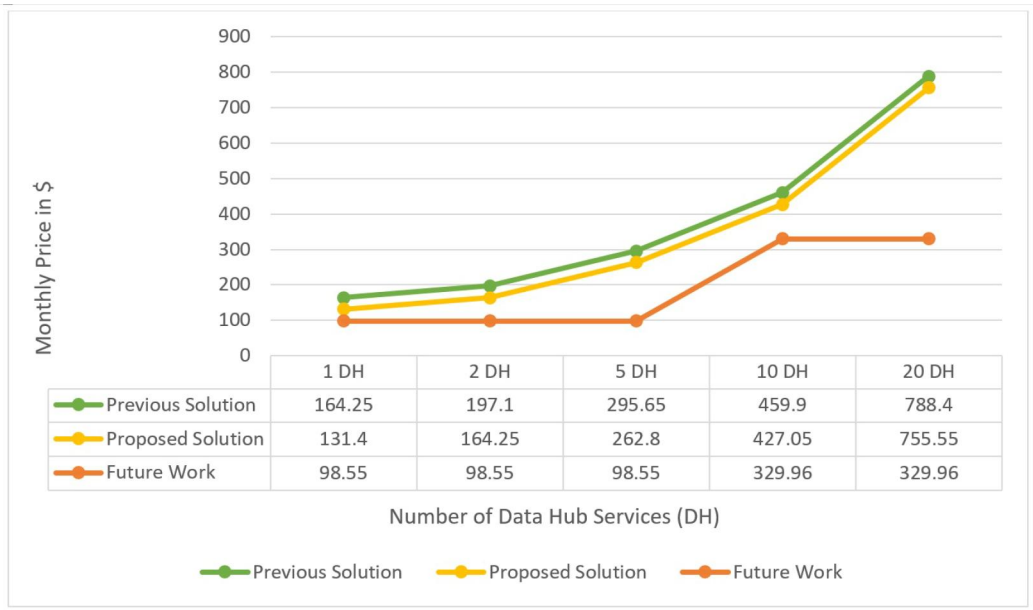

Figure 4.1: Price Comparison for Different Deployment Solutions .

4.3 Life-cycle Management

To design a suitable deployment procedure, we should consider life-cycle of the system, instead of focusing only on the first-time required actions. To this goal, in this section, we study two metrics having an impact on the system's management during different phases of development. First, we investigate how the deployment approach is flexible with system updates. This is especially important for the CI/CD workflow. Next, fault tolerance of the system is studied in two levels, for Pod and Node failures. To do these test-studies we have considered Kubernetes specifications since it is used for the deployment.

4.3.1 System Updates

We deploy the system for any customer once. After that, the system might have different updates and releases. This is highly possible especially since the system is developed based on the micro-service architecture. Hence, any of the services can be changed and there should be a possibility for a quick update based on the latest releases. Kubernetes will give us such a possibility to update our micro-services smoothly without disturbing a system that is already running. The latest Docker image of each micro-service will be stored in the private Docker registry of the project.

In Kubernetes, the updating procedure is done with 'Rolling Updates'. We can define new images in the 'Deployment' definition. Kubernetes, using 'Rolling Updates', incrementally updates the running Pods instances with the new ones based on the image version that is mentioned recently. This way deployments would be updated with zero downtime while has no effect on the system's availability. Moreover, it is totally possible that deployments be reverted to their previous versions and this is called 'Rollback' in the Kubernetes terminology. We have tested the rolling updates to different versions and rolling back to the previous ones for all the micro-services. The average time that takes the new Pod corresponding for the updated service be created is around 4.4 seconds that is reasonable. As a result, we can make sure that the CI/CD workflow of the system could be done very fast and with zero downtime.

4.3.2 Failure Recovery

A failure, for an intentional or unintentional reason, may happen any time after deploying the system. Hence, monitoring the state of running services and corresponding containers is one of the tasks that should be supported by the deployment method. Moreover, a fast-enough mechanism for recovery of the failed components is required. Kubernetes, as a container orchestration tool, supports monitoring health of containers. Moreover, it provides mechanisms for their recovery.

Failure recovery in Kubernetes is supported in the Pod level, and each Pod usually contains one or more containers. However, Pods are not resilient to machine failures by themselves and we use controllers to make them resilient. Pods, after creation, do not disappear unless they are destroyed, or an involuntary disruption happen. An involuntary disruption includes any kind of unavoidable hardware or software error. For instance, the Pod or the node (that pod is running on) could be deleted by mistake by the cluster administrator, or there might be a hardware failure in the machines. Even,

the node might disappear from the cluster due to a cluster network problem.

To observe the effect of such disruptions on our scenario, the following test cases are designed. Details of the tests and results of the failure recovery, provided by Kubernetes, are discussed as follows.

- **Pod Failure**

  In Kubernetes, Pods are not resilient to machine failures, but controllers, such as deployment, are [98]. Deployment provides declarative updates for Pods. Hence, in case of a pod-failure, deployment kills those pods that do not respond and instead, it re-creates and restarts another corresponding Pod and container that is defined by the deployment definition. To benefit from the self-healing and failure recovery characteristics of Kubernetes, which can be provided by the controller objects, we have used deployments. So, as mentioned in Section 3.4.2, to create corresponding container of each component in the system, we define 'Deployments' instead of defining a Pod directly. To make sure

Failed Pod	Average Recovery Time (S)
Zookeeper	2.6
Kafka	2.4
Dashboard	3
Coordinator	2.8
Data Fetcher	2.4
Data Hub	3
Whole of the System	50.6

Table 4.5: Average recovery time for the failed pods.

that such a failure recovery mechanism is present in our implementation and to see the effect of it on the system, we simulated failures by removing them on intention. First, for each component and then, for the whole of the system together. In both cases, we measured the time that takes for pods to be recovered. The simulated failures are repeated five times each and results can be seen in Table 4.5. A delay between 2 to 3 seconds was the usual case for recovering one single pod corresponding to one of the components. Also, removing pods, corresponding for all components, together at the same time, resulted in 50.6 seconds on average for the whole system to be up and running again. This means that if in the worst case, whole of the pods fail together, then the time that takes for the whole system to be recovered is close to one minute.

- **Node Failure**

  In Kubernetes, failure of a worker node to respond to the master node is defined as the node failure. This would happen due to network failure, often called a network partition [73]. The Pods that run on a node, will fail due to node failure.

  In Kubernetes, the controller manager is responsible for monitoring the state of the nodes in the cluster. There are a few default metrics, which are configurable, and have a role in determining the state of the nodes. First, based on the *'–node-monitor-period'* parameter, each worker node must respond to the master node every 5 seconds, which is the default time. Then, if the worker node fails to respond for a period of 40 seconds (it is defined by the *'–node-monitor-grace-period'* parameter as the default value), its state would be marked as *'Unknown'*. The last effective parameter is *'–pod-eviction-timeout'* and is defined as 5 minutes by default. After this time, all Pods that were running on this node would be deleted. After that, the scheduler will re-assign them to another available node in the cluster that has the required resources and is accordance with the specifications of the Pod. To observe the effect of node failure on our scenario, we simulated failing one of the worker nodes by shutting down its corresponding instance. To do so, we have chosen the *'Worker Node 2'* that hosts the Data Fetcher Pod.

  In Kubernetes, by default, any worker node that has the required resources can host any running pod. However, as discussed in Section 3.4.2, to serve the goals of our specific use-case, we have labeled each worker node. Then, the deployments are constrained to be run on specific nodes using these labels. Hence, to let the node recovery happen, we have created another VM, joined it to our cluster and called it *'Worker Node 4'*. Then, the *'Worker Node 2'* and *'Worker Node 4'* were attached with similar labels. We shut downed the *'Worker Node 2'*. Initially, we did not notice any changes and watching the list of running Pods, they all appeared to be running fine. After about 40 seconds the state of *'Worker Node 2'* was shown as *'NotReady'*. Finally, it took five minutes until we notice the corresponding Pod failed and another one was created in 2 seconds on the *'Worker Node 4'*. The observed life-cycle of node failure are shown in Figure 4.2.

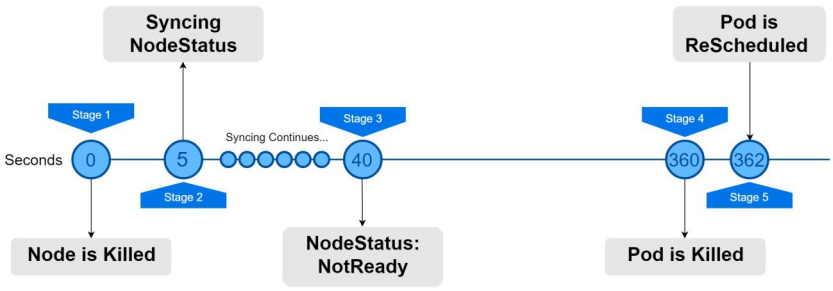

Figure 4.2: Timeline of Pod life-cycle in case of node failure.

4.4 Resource Management

Upon deploying a system, resource usage is one of the important metrics that can be monitored and measured. Such information gives us deep insights into how the system behaves. Having a real-time view of resource usage, we can prevent errors related to the insufficient resources. Hence, such monitoring is crucial for providing a reliable service and helps for scaling the system. Moreover, we would be able to optimize the resource consumptions and control corresponding costs.

In the current version of Kubernetes, the resource usage metrics, such as CPU and memory usage, are available through the Metrics API [99]. This API provides information about the resources that are currently used by a Pod or a node. However, such metrics are stored in memory and are not accessible for a long time period. The Metrics API would be available if the 'Metric Server' is deployed in the cluster. In Kubernetes, the *'Metric Server'* [100] is a cluster level component that gets these metrics from all Kubernetes nodes via Kubelet through the Summary API. Later, the metrics will be aggregated and stored in memory. Finally, they will be served in the Metrics API format. To use the *'Metric Server'*, we deployed in our cluster separately. Results that are observed through this tool is shown and discussed as follows.

4.4.1 Resource Consumption

In Kubernetes, it is possible to monitor resource consumptions for each of the micro-services. This monitoring helps us to understand how the system performs on top of Kubernetes. It also allows us to prepare the proper hardware requirements for a real-world scenario. For instance, monitoring nodes of the cluster, we observe that one of the worker nodes has a higher resource consumption than the basic requirements for a costume node in a Kubernetes cluster. This becomes important especially when we aim to

choose the right hardware resources.

We have observed and recorded amount of CPU and memory usage of cluster nodes in the different phases of a runtime. This includes three stages: Before deploying the system in the cluster, upon deployment and after issuing multiple subscriptions on the running system. Results are shown in Table 4.6. The values are calculated based on the maximum amount that is observed in each phase. Amount of the used CPU is based on the percentage, while memory usage is shown in Megabytes.

Based on the results, almost all worker nodes use most of their CPU power in stage 2, when the system is deploying. This is especially the case in the *'Worker Node 1'* since it hosts most of the micro-services. Then, in the stage 3, while we use the system to issue subscriptions, *'Worker Node 3'* has the most CPU usage, about 27 percent. This is reasonable since it hosts the main active component upon issuing a subscription, meaning the Data Fetcher.

The amount of memory consumption upon deploying the system reaches over to 2 GB in the *'Worker Node 1'*. According to this result, a VM with 2 GB RAM, which is mentioned as the minimum requirements previously, is not enough for this use-case and should be considered in a real-world setup. The memory usage in *'Worker Node 2'* and *'Worker Node 3'* reaches to around 1276 at maximum. Finally, upon using the system and issuing subscriptions, memory usage in the *'Worker Node 1'* goes even higher and reaches to 2100 MB. In the *'Worker Node 2'*, this number reaches to 1931 MB, which is close to the limitation of 2 GB. This observation on *'Worker Node 2'* is important for our use-case. According to Section 3.2.2, the ultimate aim in this use-case is to deploy the Data Fetcher module on a gateway, instead of running it on a VM. The gateway that we are planning to use in the future has capabilities of 2 cores CPU and 2 GB RAM. Hence, studying the possibility of such a scenario has been one of the goals of this research study.

4.4.2 Resource Optimization

In practice, containers have no upper bound on their CPU and memory usage. They can consume all the resources that are available on the node, where it is running on. In case of memory, this might invoke the *'Out of Memory (OOM)'* killer and there is a possibility for the container to be killed. To avoid this, Kubernetes gives us the possibility to manage the amount of resources that would be consumed by containers. By configuring resource limits for the containers running on our cluster, we avoid losing jobs. Moreover, this way, we can make efficient use of the available resources on our cluster's nodes. We define such resource constraints in the Pod definition,

Node	CPU (%)			Memory (MB)		
	Stage 1	Stage 2	Stage 3	Stage 1	Stage 2	Stage 3
Master	4	6	4	1689	1693	1714
Worker 1	1	99	7	789	2057	2100
Worker 2	1	91	27	567	1276	1931
Worker 3	1	71	7	594	1224	1629

Table 4.6: Resource usage in three stages of a runtime for different nodes. Stage 1: Before deploying the system in the cluster; Stage 2: Upon deployment; Stage 3: After issuing multiple subscriptions on the running system.

with two sub-categories: *'requests'* and *'limits'*. In *'requests'*, we define a reasonable value for the memory and CPU requests of the Pod to be running properly. Defining this value is a help for scheduling of Pods in the nodes with appropriate available resources. Then, in the *'limits'* field, we define the upper bound that the Pod can use. Hence, the containers would not be allowed to use the whole of available resources on the node.

We have defined such constraints for the Data Fetcher Pod. The constraints are defined only on memory usage. The motivation behind this is the low capabilities that the real device (for running Data Fetcher) has and is discussed before. While the memory request is defined by 1000 MB, the memory limit is tested with different values. Results show that defining the limit with a low value causes the corresponding Pod to fail. Finally, with a limitation value of 1200 MB, the Data Fetcher Pod was working properly. To make sure that the system works fine under such resource constrains, multiple subscriptions were issued, and the runtime was observed for different periods of time.

Observing the memory usage of Data Fetcher Pod, before defining any constraints, shows that the Pod can use up to 1555 MB, while the memory usage of the *'Worker Node 2'*, which hosts this Pod, reaches to 2023 MB at latest. This result is recorded under a system runtime pressure, where about 20 different subscriptions are issued with a low free time in between. After defining the memory limit for 1200 MB, the maximum memory usage in the Data Fetcher Pod reaches to 1157 MB, while the memory usage in *'Worker Node 2'* goes up to 1628 MB. Hence, according to the results, we are able to constrain resources while the system continues to work properly.

Memory usage By	Before Defining Resource Limitations (MB)	After Defining Resource Limitations (MB)
Data Fetcher Pod	1555	1157
Worker Node 2	2023	1628

Table 4.7: Memory usage before and after defining resource constraints.

4.5 Scalability

In this section, we have studied and evaluated how the used approach for deploying the IoT system is scalable. We have categorized the section into two general sub-sections: *'Scaling Pods'* and *'Scaling Services'*. In the former, the goal is to analyze the scalability of individual components of the micro-service system, while the latter concentrates on the whole solution to be used on a larger scale in the future.

4.5.1 Scaling Pods

In Kubernetes, we can scale the individual Pods corresponding to different micro-services independently. This would be beneficial in case one of the Pods in the cluster has more user traffic and requires more resources. Hence, resources would not be wasted on scaling all the micro-services together, while scaling only one of them is needed. To test how the system reacts to such scaling we have used two possible approaches. First, we have tried a manual scaling and results are provided. Next, the auto-scaling as the other possible option in the Kubernetes is activated and the requirements are discussed.

- **Manual Scaling**

  The manual scaling is possible through updating number of replicas in a controller definition, for instance in the Deployment definition in our case. This way, the number of Pods will be increased to the new desired state.

  After deploying the system in the cluster, we scaled out each individual Pod from 1 replica to 2 replicas and the time that takes for them to be deployed is recorded as an average of five times repetition of such test scenario. The test was repeated for scaling from one to 5 replicas. Then, we also calculated the time for scaling in each Pod from 2 and 5 replicas to one. Finally, we have repeated these test cases for scaling

out and scaling in the whole of the system's Pods together at one attempt. Results show that the time, which takes for scaling in, is always less than the time that takes for scaling out a Pod and this makes reasonable. However, based on our observations, this is not the case for the *'Zookeeper'* Pod. Scaling in of this Pod always take a longer time than its scaling out. This also affects the scale in time for the whole system as the *'Zookeeper'* Pod scales in very slowly. Hence, while it takes 38.8 seconds for the whole Pods to be scaled out to 5 replicas each, the time for scaling in is 58 seconds, in contrast of our expectation. However, no reason has found based on our studies for this but should be studied in the future.

Scaled Pod	Scale Out Time For 1 to 2 Replicas (S)	Scale In Time For 2 to 1 Replicas (S)	Scale Out Time For 1 to 5 Replicas (S)	Scale In Time For 5 to 1 Replicas (S)
Zookeeper	4.6	30	15	35
Kafka	5.6	2.8	12.8	7.8
Dashboard	2.6	1.8	6.6	4.6
Coordinator	3.8	2.8	9.6	6.2
Data Fetcher	2.8	1.8	4.8	3.2
Data Hub	2.6	1.2	4.6	3.2
All together	26.2	34.2	38.8	58

Table 4.8: Average time for scaling out and scaling in the Pods.

- **Auto-Scaling**

In addition to the manual scaling, Kubernetes provides an auto-scaling mechanism by implementing a *'Horizontal Pod Auto-Scaler (HPA)'*. The HPA is implemented as a Kubernetes API and a controller. The controller periodically checks the number of replicas in a deployment (or other types of controller). Then, it adjusts this number in order to matches the observed CPU utilization with the target that is specified by the user.

To let HPA access the resource metrics and work in Kubernetes, the *'Metric Server'* should be deployed on the cluster. For HPA to work in Kubernetes, it needs to have access to the resource metrics such as CPU and RAM. As explained in Section 4.4, to get the resource metrics in Kubernetes, the *'Metric Server'* must be deployed on our cluster.

We have set up a horizontal auto-scaler for one of the deployment objects (the Data Fetcher), in our cluster. The aim is to see how such HPA acts in

presence of a high workload and if it starts to work automatically in practice. In the definition of the HPA, we have defined the minimum number of Pods as one, and maximum as three. The condition for starting the HPA is defined by the CPU usage to be reached to 10 percentage.

Firstly, watching the amount of CPU usage by the corresponding Pod, we have observed that the CPU usage went above the 10 percent but the HPA did not apply and the number of Data Fetcher Pods stayed at one. Later, we have noticed for the HPA to work, it is required that a CPU request be specified in the Deployment definition. This way, we could observe that the number of Data Fetcher Pods were scaled from one to three when the target CPU usage is met. The time that it took for such a scaling was not more than 5 seconds in different repetitions of the test-case.

4.5.2 Scaling Services

Apart from the scalability of each individual component, the whole approach should be applicable for the same described use-case scenario in a larger scale. In our scenario, which is explained in Section 3.2.1, we aim to give access to such Pub/Sub IoT system to a company, such as company X, and it would be the producer of data and has access to the corresponding UI of the Dashboard service. This is while this company should let its customer, such as company Y, access and consume this data. To do so, the company Y should have access to the corresponding UI of the Data Hub service. In Section 3.4.3, we have seen that this is a possible scenario while we expose these two services using NodePort. To prevent access to both services on the same IP, we have used a separate VM for hosting each. Hence, they have their specific IP:PORT.

In a real-world scenario, the company that produce the data aims to give access to more than one consumer. Hence, we must be able to expose more than one Data Hub service instead of only one. Such scalability has evaluated in this section. Currently, different Data Hub services are distinguished with an ID and using that, they communicate with the whole system. Hence, to add the second Data Hub service, first, we have created its corresponding Docker image. Then, a *'Deployment'* and a *'Service'* definition is created correspondingly. We have used another VM for hosting the second Data Hub service. However, trying to expose it on the same port, which we have used for the first Data Hub service, was not successful. Later, we have noticed that when we assign a port to a service using NodePort, Kubernetes opens that specified port on all the worker nodes within the cluster. Therefore, we can only expose one service per port using NodePort.

Finally, we were able to expose the second Data Hub service on the

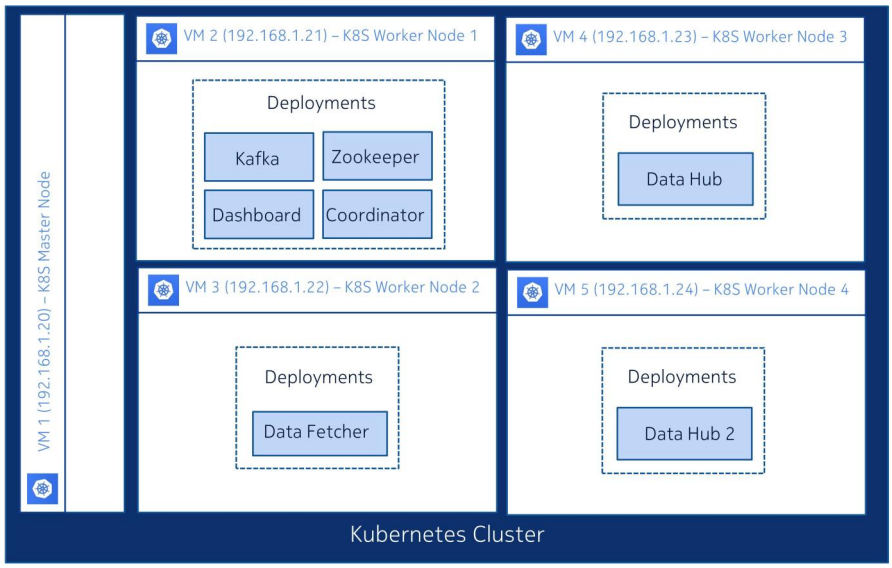

Figure 4.3: General architecture for an scalable scenario.

'Worker node 4' using a different port. The architecture of such a case is shown in Figure 4.3. Having two Data Hub services, the system was working properly, and the approach is scalable. However, a downside to the used method is that we must use one VM per Data Hub and it makes the approach expensive. In future works, we have discussed a better way of exposing service in Kubernetes that will eliminate the need for the additional hardware equipment.

Chapter 5

Conclusion and Future Work

Mobile telecommunication technologies are rapidly evolving. The shift from 4G to 5G is a big advancement that enables new ecosystems. Introducing a trade-off between speed, latency and cost by 5G opens opportunities especially for IoT scenarios. Hence, widespread adoption of 5G, as the enabler of IoT, makes it possible for IoT scenarios to become more feasible.

One of such scenario is an IoT system based on the pub/Sub messaging architecture that is being developed in Nokia Bell Labs. The system aims to provide efficient data transmission between data producers and consumers in an IoT ecosystem and is designed based on the micro-service architecture. Development of the system is finalized, however, its deployment in the real-world scenarios has some challenges and introduces some research questions.

In this chapter, we summarize the main conclusions of the research questions in Section 5.1 and discuss some ideas for the future works in Section 5.2.

5.1 Conclusions

We present the conclusions according to the research questions that are presented in Section 1.2.

- **RQ1.** *How can we ease deployment of the Pub/Sub IoT system and eliminate system's limitations for a large-scale deployment?*

  In this thesis, we studied that containerization bundles an application and its dependencies together in a self-contained unit. Hence, as an alternative to running the corresponding Jar file of each micro-service directly on the infrastructure, we containerized the Pub/Sub IoT system.

Furthermore, we studied whether the deployment of the Pub/Sub IoT system using Kubernetes will provide a service discovery among the micro-services of the system to ease its deployment. Kubernetes has an internal built-in DNS service, named CoreDNS, that is applied on the 'Service' level. 'Service' is one of the Kubernetes object types that is defined by the Kubernetes API. CoreDNS monitors the Kubernetes API and automatically creates DNS records for any new services. This enables service discovery across a cluster and pods are able to perform name resolution automatically for all services.

Hence, defining each of the micro-services in the current IoT system as a 'Service' object type in Kubernetes, we can utilize Kubernetes's service discovery for our purpose. This allows us to change the hard-coded IPs to the name of Kubernetes services. So, changing IPs per deployment is not required anymore and this is especially beneficial for the large-scale deployments.

- **RQ2.** *Which containerization and Kubernetes mechanisms should be used in the deployment procedure to serve requirements of the system?*

  To containerize the Pub/Sub IoT system, Docker Container Engine is chosen as the containerization tool, since it works well with Kubernetes. Then, to preserve the privacy of the project and to keep the created Docker images within the company, we decided to set up a private Docker Registry. We have seen that setting up a private registry was a straightforward procedure, using the Docker's open-source project, named 'Docker Registry'.

  To deploy the Pub/Sub IoT system using Kubernetes, we were required to consider some specifications. One of such specifications is the location of Data Fetcher service since it is developed to be performed at the edge of the network. To serve such capability for DataFetcher, we must make sure that it is running on a separate node within the cluster. To dedicate a micro-service to be run in a particular node in the Kubernetes cluster, we labeled each node. Then, defining 'nodeSelector' in the Deployment definition of each micro-service and assigning the right label, we could manage such dedication.

  Public exposure of two micro-services, Dashboard and DataHub, is from the other requirements of the Pub/Sub IoT system. To this goal, we defined their corresponding Service definition from the NodePort type. However, NodePort exposes a service on the specified port and IP address of all worker nodes. This is not fulfilling requirements of the system since we aim to give unique access of each service to a particular

end-user. To solve the issue, we defined the 'externaltrafficpolicy' to be set as 'Local' in the Service definition of DataHub and Dashboard. As a result, Kubernetes only proxies requests to the local endpoints and never forwards traffic to the other nodes of the cluster.

Furthermore, the default port range of NodePort service type in Kubernetes is between 30000-30767. For a service of type NodePort, if a port is not specified, Kubernetes will assign a port within this range automatically. A port is hard-coded in the front-end of the Dashboard micro-service. Hence, we were required to expose Dashboard service on that particular port. Since the port was not within the default NodePort range of Kubernetes, we have configured this default port range in Kubernetes.

- **RQ3.** *What are the implications of the system's deployment based on Kubernetes for the efficiency of the system and its scalability?*

  Upon deployment and running the system, we evaluated the runtime for issuing a subscription. Results show a faster response time. This is because there are less external communications in the Kubernetes-based approach compared to the previous deployment method. Moreover, we calculated the corresponding costs for each of the previous and proposed deployment approaches. We have seen that using Kubernetes for deployment, is cheaper especially when the solution scales.

  Kubernetes provides better life-cycle management for the deployment using metrics, such as updating of the system and its failure recovery. For system's updating, Kubernetes uses concept of 'Rolling Updates'. Results show that updating any of the system's micro-services takes under 5 seconds.

  Furthermore, using Kubernetes, it is possible to monitor resource consumptions and also utilize the resource usages for each of the services. We noticed that the 'Worker Node 1' has a higher resource consumption than the basic requirements for a custom node. This is reasonable since most of the workload including four of the micro-services are running on this node. This allows us to prepare proper hardware requirements for a real-case scenario. Furthermore, we could manage the resource usage of the Data Fetcher service. In the future, we aim to run the Data Fetcher on a gateway. Such gateway has specific hardware requirements including 2 cores CPU and 2 GB RAM. Hence, we should make sure that we can keep Data Fetcher's usage in this range. We saw that Kubernetes features allow for such resource management. Finally, regarding scalability of the proposed approach, we have seen that it is

possible to define more than one Data Hub service and each can be exposed on a separate IP and port.

5.2 Future Work

Our work shows several benefits as results of deploying the current Pub/Sub IoT system using Kubernetes. However, there are still some limitations in the proposed solution towards the deployment for a real-world scenario. Hence, we suggest the following direction for future works.

1. While we defined an architecture for the proposed deployment solution in Section 3.2.2, we discussed the important design decisions. One of such decisions was related to the physical location of Data Fetcher service in our infrastructure. Providing pre-processing and aggregation for data, this service is designed to benefit from capabilities offered by MEC architecture. Hence, in a real-world scenario, Data Fetcher's container is supposed to be established on an IoT gateway. An IoT gateway would be placed close to where IoT devices are resided and acts as the connection point between IoT devices and the cloud. Such a gateway has some computation power as well as network connectivity.

   The device that we had accessible at the time of writing this thesis, was Nokia's gateway with 1 core CPU and 1 GiB RAM. However, according to the Kubernetes documents, the minimum requirements for a node to be run in a Kubernetes cluster is 2 cores CPU and 2 GiB RAM. Hence, despite using such gateway, in this work we simply used a VM to be the host for Data Fetcher's container. As future work, we plan to deploy Data Fetcher's service on a real gateway with the proper hardware requirements. During our research, we learned that a multi-architecture Kubernetes cluster is required for this purpose. This is because the CPU architecture of the intended gateway is different from the other nodes (VMs) in the cluster.

2. In the current Pub/Sub IoT system, two of the micro-services have their corresponding UIs that must be accessible by the end-users. In a scalable approach, we aim to have more than one DataHub service. Hence, the addresses of different DataHub URLs must not be the same. In Section 3.4.3, we discussed the conducted approach for service exposure in Kubernetes based on NodePort. NodePort exposes a service on a port and to access the service, we use combination of IP of the VM and the specified port.

One limitation of this approach is that remembering IP and port is not easy for users. As the solution, we can configure the DNS server to point a domain name to the IP of the ports. Moreover, if we expose more than one Data Hub service on the same VM, we will have the same IPs and only ports will change. Hence, to provide such scalability, we have used a VM per Data Hub service in the proposed architecture with NodePort and the 'externaltrafficplocy' is defined as 'Local' for these services. in Section 4.2.2, we discussed that this approach is not cost-efficient for scaling the deployment.

As an alternative approach, we suggest using a loadBalancer for exposing services. LoadBalancer is another service type that is explained in Section 3.4.3. Using a loadBalancer for exposing services, we can deploy as many Data Hub services as we like on only one VM, and instead, use different LoadBalancers for each. This way, we will have different IP addresses to access each service. However, this approach still has limitations. Firstly, the DNS server should be configured for each newly added service and new loadBalancer. Moreover, from the perspective of cost-efficiency, using a loadBalancer per service still could inversely increase the cloud bill.

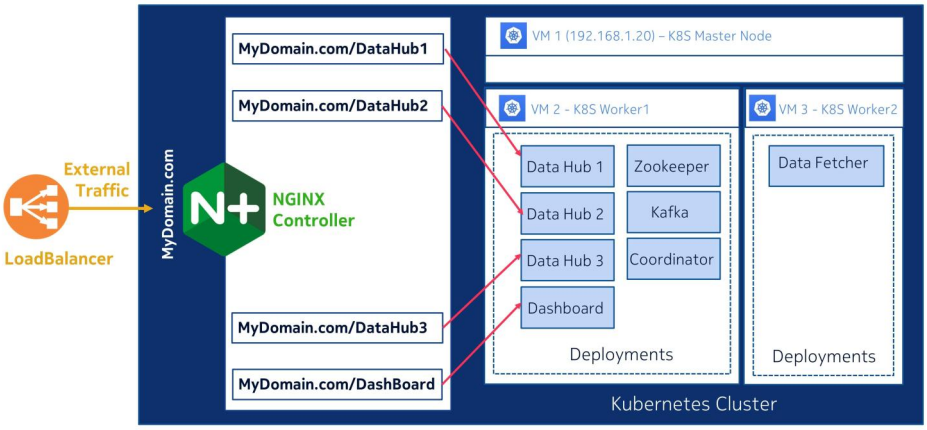

Figure 5.1: Architecture overview of the proposed scenario with Ingress.

Finally, there is still another approach for exposing services in Kubernetes, namely *'Ingress'* and we suggest it to be used as the best approach in the future works.

The concept of *'Ingress'* is explained in Section 3.4.3. To satisfy an Ingress in Kubernetes, an Ingress controller is required. NGINX Ingress controller is one of the available options. Ingress will act as the single-

entry point to the cluster. Then, defining different paths for each service, it routes the related incoming external request to the right service. This way we do not need to use multiple IP addresses per service as it is with loadBalancer. Hence, it is very cost-efficient. Furthermore, Ingress will be configured as an internal Kubernetes object and being the single entry-point to the cluster, it is the only place that will be affected upon adding a new service.

Using an Ingress we can use a single domain name and assign each service a separate path. Following this approach in the future works for deploying the Pub/Sub IoT system still requires an improvement in the system development's level. Since the addresses will be unique only in their paths, an authentication mechanism should be developed for accessing both Dashboard and Data Hub services.

The architecture overview of the proposed deployment solution using NGINX Ingress Controller is shown in Figure 5.1. An external loadBalancer can be used to access the ingress controller, which is resided inside the Kubernetes cluster. Ingress has the domain name of *'MyDomain.com'* and to access different services, different paths are defined.

Bibliography

[1] H. Singh, "Statistics that prove iot will become massive from 2018." `http://customerthink.com/statistics-that-prove-iot-will-become-massive-from-2018/`, 2018. [Online; accessed 3-April-2019].

[2] Ericsson, "Ericsson mobility report." `https://www.ericsson.com/en/mobility-report/reports/november-2018`, 2018. [Online; accessed 3-April-2019].

[3] Nokia, "Nokia bell labs." `https://www.bell-labs.com/`. [Online; accessed 25-September-2019].

[4] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison," in *21st Twente Student Conference on IT*, vol. 1, pp. 1–7, 2014.

[5] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, pp. 342–346, IEEE, 2015.

[6] S. Muralidharan, G. Song, and H. Ko, "Monitoring and managing iot applications in smart cities using kubernetes," *CLOUD COMPUTING 2019*, p. 11, 2019.

[7] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications," *arXiv preprint arXiv:1901.04946*, 2019.

[8] S. Otaniemi, "Smart otaniemi." `https://smartotaniemi.fi/`. [Online; accessed 25-September-2019].

[9] B. Finland, "Business finland." `https://www.businessfinland.fi/en/`. [Online; accessed 25-September-2019].

[10] S. Otaniemi, "Smart otaniemi." `https://smartotaniemi.fi/pilots/platforms-connectivity/`. [Online; accessed 25-September-2019].

[11] I. Global, "What is cellular network." `https://www.igi-global.com/dictionary/cellular-network/3547`. [Online; accessed 1-August-2019].

[12] Wikipedia, "Cellular network." `https://en.wikipedia.org/wiki/Cellular_network`. [Online; accessed 1-August-2019].

[13] SearchNetworking, "radio access network (ran)." `https://searchnetworking.techtarget.com/definition/radio-access-network-RAN`. [Online; accessed 1-August-2019].

[14] R. N. Mitra and D. P. Agrawal, "5g mobile technology: A survey," *ICT Express*, vol. 1, no. 3, pp. 132–137, 2015.

[15] R. S. Karki and V. B. Garia, "Next generations of mobile networks," *International Journal of Computer Applications*, vol. 975, p. 8887, 2016.

[16] A. Gawas, "An overview on evolution of mobile wireless communication networks: 1g-6g," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 3, no. 5, pp. 3130–3133, 2015.

[17] A. Damnjanovic, J. Montojo, Y. Wei, T. Ji, T. Luo, M. Vajapeyam, T. Yoo, O. Song, and D. Malladi, "A survey on 3gpp heterogeneous networks," *IEEE Wireless communications*, vol. 18, no. 3, pp. 10–21, 2011.

[18] Wikipedia, "Small cell." `https://en.wikipedia.org/wiki/Small_cell`. [Online; accessed 1-August-2019].

[19] A. S. W. Marzuki, I. Ahmad, D. Habibi, and Q. V. Phung, "Mobile small cells: Broadband access solution for public transport users," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 190–197, 2017.

[20] F. Al-Turjman, *Smart Things and Femtocells: From Hype to Reality*. CRC Press, 2018.

[21] K. ISHIZU, H. MURAKAMI, K. IBUKA, and F. KOJIMA, "2-1 next generation mobile communications system to realize flexible architecture and spectrum sharing," *Journal of the National Institute of Information and Communications Technology Vol*, vol. 64, no. 2, 2017.

[22] S. Li, L. Da Xu, and S. Zhao, "5g internet of things: A survey," *Journal of Industrial Information Integration*, vol. 10, pp. 1–9, 2018.

[23] M. Agiwal, A. Roy, and N. Saxena, "Next generation 5g wireless networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1617–1655, 2016.

[24] Nokia, "5g new radio network." `https://onestore.nokia.com/asset/205407`. [Online; accessed 1-August-2019].

[25] D. Wang, D. Chen, B. Song, N. Guizani, X. Yu, and X. Du, "From iot to 5g i-iot: The next generation iot-based intelligent algorithms and 5g technologies," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 114–120, 2018.

[26] Qwilt, "The mobile edge cloud 5g and mec." `https://qwilt.com/5g-mec/mec-and-5g/`. [Online; accessed 1-August-2019].

[27] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing?a key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[28] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[29] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for internet of things realization," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 2961–2991, 2018.

[30] Q. H. Mahmoud, *Middleware for communications*, vol. 73. Wiley Online Library, 2004.

[31] "Mqtt-essentials-part2-publish-subscribe." `https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/`, 2015. [Online; accessed 23-April-2019].

[32] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[33] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting iot platform requirements with open pub/sub solutions," *Annals of Telecommunications*, vol. 72, no. 1-2, pp. 41–52, 2017.

[34] "Apache kafka documentation." `https://kafka.apache.org/documentation/`. [Online; accessed 24-April-2019].

[35] "Ibm official page." `https://www.ibm.com/products/mq`. [Online; accessed 24-April-2019].

[36] "Mqtt official page." `http://mqtt.org/`. [Online; accessed 24-April-2019].

[37] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg, "Content-based publish-subscribe over structured overlay networks," in *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pp. 437–446, IEEE, 2005.

[38] Wikipedia, "Apache kafka." `https://en.wikipedia.org/wiki/Apache_Kafka`. [Online; accessed 1-August-2019].

[39] Apache, "Zookeeper." `https://zookeeper.apache.org/`. [Online; accessed 1-August-2019].

[40] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[41] M. Fowler and J. Lewis, "Microservices." `https://martinfowler.com/articles/microservices.html`. [Online; accessed 1-August-2019].

[42] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *Ieee Software*, vol. 33, no. 3, pp. 42–52, 2016.

[43] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[44] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and ulterior software engineering*, pp. 195–216, Springer, 2017.

[45] V. Kojola, S. Kapoor, and K. Hätönen, "Distributed computing of management data in a telecommunications network," in *International Conference on Mobile Networks and Management*, pp. 146–159, Springer, 2016.

[46] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[47] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing?the business perspective," *Decision support systems*, vol. 51, no. 1, pp. 176–189, 2011.

[48] S. Sharma and Y. Park, "Virtualization: A review and future directions executive overview," *American Journal of Information Technology*.

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

[50] M. Eder, "Hypervisor-vs. container-based virtualization," *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, vol. 1, 2016.

[51] P. Dash, *Getting started with oracle vm virtualbox*. Packt Publishing Ltd, 2013.

[52] A. Muller and S. Wilson, "Virtualization with vmware esx server," 2005.

[53] A. Velte and T. Velte, *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[54] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, vol. 37, pp. 164–177, ACM, 2003.

[55] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor.," in *USENIX Annual Technical Conference, General Track*, pp. 1–14, 2001.

[56] J. Honeycutt, "Microsoft virtual pc 2004 technical overview," *Microsoft, Nov*, 2003.

[57] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, 2005.

[58] Docker, "What is a container?." `https://www.docker.com/resources/what-container`. [Online; accessed 1-August-2019].

[59] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, "Security of os-level virtualization technologies," in *Nordic Conference on Secure IT Systems*, pp. 77–93, Springer, 2014.

[60] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[61] Docker, "The industry-leading container runtime." `https://www.docker.com/products/container-runtime`. [Online; accessed 5-August-2019].

[62] Docker, "Docker overview." `https://docs.docker.com/engine/docker-overview/`. [Online; accessed 5-August-2019].

[63] T. Bui, "Analysis of docker security," *arXiv preprint arXiv:1501.02967*, 2015.

[64] C. Anderson, "Docker [software engineering]," *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015.

[65] Docker, "About storage drivers." `https://docs.docker.com/storage/storagedriver/#images-and-layers`. [Online; accessed 5-August-2019].

[66] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *SoutheastCon 2016*, pp. 1–5, IEEE, 2016.

[67] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[68] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: an experience report," in *European Conference on Service-Oriented and Cloud Computing*, pp. 201–215, Springer, 2015.

[69] E. Casalicchio, "Autonomic orchestration of containers: Problem definition and research challenges.," in *VALUETOOLS*, 2016.

[70] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," 2016.

[71] Elastisys, "Setting up highly available kubernetes clusters." `https://elastisys.com/wp-content/uploads/2018/01/kubernetes-ha-setup.pdf`. [Online; accessed 6-August-2019].

[72] Kubernetes, "Kubernetes concepts." `https://kubernetes.io/docs/concepts/`. [Online; accessed 6-August-2019].

[73] Kubernetes, "Pods." `https://kubernetes.io/docs/concepts/workloads/pods/pod/`. [Online; accessed 6-August-2019].

[74] CoreOS, "Overview of a pod." `https://coreos.com/kubernetes/docs/latest/pods.html`. [Online; accessed 6-August-2019].

[75] Kubernetes, "Viewing pods and nodes." `https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/`. [Online; accessed 6-August-2019].

[76] Kubernetes, "Replication controller." `https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/`. [Online; accessed 6-August-2019].

[77] Kubernetes, "Kubernetes replicaset." `https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/`. [Online; accessed 19-August-2019].

[78] Kubernetes, "Services." `https://kubernetes.io/docs/concepts/services-networking/service/`. [Online; accessed 6-August-2019].

[79] Kubernetes, "Kubernetes control plane." `https://kubernetes.io/docs/concepts/#kubernetes-control-plane`. [Online; accessed 16-August-2019].

[80] Kubernetes, "Kubernetes api server." `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/`. [Online; accessed 6-August-2019].

[81] Kubernetes, "Kubernetes controller manager." `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/`. [Online; accessed 6-August-2019].

[82] etcd, "etcd." https://etcd.io/. [Online; accessed 7-August-2019].

[83] Wikipedia, "Openstack." https://en.wikipedia.org/wiki/OpenStack. [Online; accessed 20-August-2019].

[84] Docker, "Networking in compose." https://docs.docker.com/compose/networking/. [Online; accessed 20-August-2019].

[85] Docker, "About registry." https://docs.docker.com/registry/introduction/. [Online; accessed 20-August-2019].

[86] Whatis, "Iot gateway." https://whatis.techtarget.com/definition/IoT-gateway. [Online; accessed 20-August-2019].

[87] Nokia, "Nokia engineering and services cloud." https://learningstore.nokia.com/employee/item/n.1486421574681. [Online; accessed 20-August-2019].

[88] Kubernetes, "Installing kubernetes with minikube." https://kubernetes.io/docs/setup/learning-environment/minikube/. [Online; accessed 20-August-2019].

[89] Kubernetes, "Single control-plane cluster with kubeadm." https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/. [Online; accessed 20-August-2019].

[90] Kubernetes, "Installing kubeadm." https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/. [Online; accessed 20-August-2019].

[91] Kubernetes, "Cluster networking." https://kubernetes.io/docs/concepts/cluster-administration/networking/. [Online; accessed 20-August-2019].

[92] Kubernetes, "Installing addons." https://kubernetes.io/docs/concepts/cluster-administration/addons/. [Online; accessed 20-August-2019].

[93] Kubernetes, "Deployments." https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Online; accessed 08-September-2019].

[94] matthewpalmer, "Kubernetes ingress vs loadbalancer vs nodeport?." https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html.

[95] Kubernetes, "What is ingress?." https://kubernetes.io/docs/concepts/services-networking/ingress/. [Online; accessed 29-September-2019].

[96] Kubernetes, "Installing kubeadm." https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/. [Online; accessed 20-September-2019].

[97] M. Azure, "Pricing calculator." https://azure.microsoft.com/en-us/pricing/calculator/#virtual-machines90ea3354-5786-410d-b59b-c9bf638bf484. [Online; accessed 17-September-2019].

[98] Kubernetes, "Pod lifecycle." https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. [Online; accessed 20-September-2019].

[99] Kubernetes, "Resource metrics api." https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/resource-metrics-api.md. [Online; accessed 22-September-2019].

[100] Kubernetes, "Kubernetes metrics server." https://github.com/kubernetes-incubator/metrics-server. [Online; accessed 22-September-2019].