Master's thesis

Master's Programme in Computer Science

Scalable and High Available Kubernetes Cluster in Edge Environments for IoT Applications

Hyeongju Lee

June 10, 2021

FACULTY OF SCIENCE

UNIVERSITY OF HELSINKI

Supervisor(s)

Prof. Petteri Nurmi, Dr. Naser Hossein Motlagh

Examiner(s)

Prof. Petteri Nurmi

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Utbildningsprogram — Study programme Master's Programme in Computer Science

Tekijä — Författare — Author

Hyeongju Lee

Työn nimi — Arbetets titel — Title

Scalable and High Available Kubernetes Cluster in Edge Environments for IoT Applications

Ohjaajat — Handledare — Supervisors

Prof. Petteri Nurmi, Dr. Naser Hossein Motlagh

Työn laji — Arbetets art — Level Master's thesis	Aika — Datum — Month and year June 10, 2021	Sivumäärä — Sidoantal — Number of pages 57 pages

Tiivistelmä — Referat — Abstract

The number of IoT and sensor devices is expected to reach 25 billion by 2030. Many IoT applications, such as connected vehicle and smart factory that require high availability, scalability, low latency, and security have appeared in the world. There have been many attempts to use cloud computing for IoT applications, but the mentioned requirements cannot be ensured in cloud environments. To solve this problem, edge computing has appeared in the world. In edge environments, containerization technology is useful to deploy apps with limited resources.

In this thesis, two types of high available Kubernetes architecture (2 nodes with an external DB and 3 nodes with embedded DB) were surveyed and implemented using K3s distribution that is suitable for edges. By having a few experiments with the implemented K3s clusters, this thesis shows that the K3s clusters can provide high availability and scalability. We discuss the limitations of the implementations and provide possible solutions too.

In addition, we provide the resource usages of each cluster in terms of CPU, RAM, and disk. Both clusters need only less than 10% CPU and about 500MB RAM on average. However, we could see that the 3 nodes cluster with embedded DB uses more resources than the 2 nodes + external DB cluster when changing the status of clusters.

Finally, we show that the implemented K3s clusters are suitable for many IoT applications such as connected vehicle and smart factory. If an application that needs high availability and scalability has to be deployed in edge environments, the K3s clusters can provide good solutions to achieve the goals of the applications. The 2 nodes + external DB cluster is suitable for the applications where the amount of data fluctuate often, or where there is a stable connection with the external DB. On the other hand, the 3 nodes cluster will be suitable for the applications that need high availability of the database even in poor internet connection.

ACM Computing Classification System (CCS)
Computer systems organization → Embedded and cyber-physical systems
Human-centered computing → Ubiquitous and mobile computing

Avainsanat — Nyckelord — Keywords

IoT, edge computing, container, Kubernetes, high availability, scalability

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking study track

Contents

1 Introduction

The advent of the Internet of Things (IoT) has connected physical objects to the Internet. As time passes, many IoT applications have been developed. For the recent few years, connected vehicles and IIoT (smart factory) are the most popular IoT applications in terms of the number of projects. For those IoT applications, in order to provide a service and commoditize it to use in real life, many aspects have to be considered. First, high availability is a very important factor. If a connected vehicle cannot accomplish high availability and there is a downtime even shortly, it may be the cause of a car accident. In the case of smart factory that has to be operating 24/7, too, downtime of the system means cost and financial loss (Liberg et al., 2020, Masters, 2017, Lerner, 2014). Second, scalability is also an essential key for those applications. In the case of connected vehicles, the amount of resources has to be scaled according to the amount of data that fluctuate often. When the vehicles are not being driven, the amount of data is small. However, when they are driven, a huge volume of data is generated in a short time (IBM, 2018). If the amount of resources assigned for the system does not change based on the situation, the system will not be able to provide any good experience for drivers. On the other hand, a smart factory requires scalability in another sense. When adding more devices that generate more data to be processed, it is necessary to scale up/out the resources. In order to prevent hindering growth from the difficulties of scaling the system, scalability has to be considered when introducing the IIoT system (Ferner, 2021).

Cloud computing resources had been used for providing solutions for IoT systems at first. But such IoT applications above have other important requirements such as low latency and security that cloud computing cannot fulfill by nature (Sasaki et al., 2016). For these reasons, edge computing has appeared in the world. By putting computing resources at the edge side of the network, it is possible to fulfill the requirements of the IoT applications mentioned above.

In edge environments too, there are challenges caused by the limited amount of resources. Since edge devices are small and have a small amount of resources, it is necessary to utilize the resources effectively (Xu et al., 2017).

VM has been used in cloud computing environments for such purposes, but it is too heavy for edge devices to run VMs. Container technology has grown so much in a few recent years, and it is regarded as a proper way to deploy applications on edge devices (Morabito, 2017). Docker opened a new era where container technology is used for deploying applications, and Kubernetes has appeared as a tool to manage the containers automatically. In succession, more lightweight way to create Kubernetes clusters on edges such as K3s and MicroK8s distributions, and KubeEdge platform.

In this thesis, two architectures that can make Kubernetes clusters high available and scalable were surveyed. By implementing clusters in the limited number of machines that have limited resources, this thesis shows why K3s clusters are suitable for IoT applications deployed in edge environments where high availability and scalability have to be guaranteed. First, we show how the clusters can provide high availability and scalability through a few experiments. Second, we show how much resources each cluster use. Third, based on the results, we discuss which cluster is more suitable for specific applications.

Following contents of this thesis are organized as follows. Chapter 2 provides background knowledge on IoT, cloud computing, and edge computing. Chapter 3 explains containerization technology and why it is necessary for edge computing environments. Chapter 4 introduces Kubernetes, a container orchestration tool, in terms of the basic architecture, objects, etc. Chapter 5 shows the implementation of K3s clusters. We provide the results from a few experiments that the implemented K3s clusters provide the high availability and scalability, the resource usage of each cluster, and discussion on the results.

2 Background on IoT and Computing

2.1 Internet of Things

Internet of Things (IoT) is the network of ’things.’ Here, the ’things’ can be computing devices, machines, or physical objects such as machines, animals, or people. In addition, identifiers such as MAC or UUID, through which the thing can be identified uniquely, have to be assigned to each thing (Gillis, 2020). In an IoT system, devices sense by themselves without the intervention of humans and collect data. It is different from the data generated manually collected by humans with the behaviors such as typing or scanning a barcode. The data generated from IoT devices is shared with other devices via the Internet and collected, and humans can utilize it to track something, make a loss and cost lower, etc (Ashton et al., 2009).

The primitive model of IoT appeared in the world in 1982. A Coca-Cola vending machine connected to ARPANET was invented by the students of Carnegie Mellon University. The students wanted to remotely check how many bottles of cola are left in the vending machine, how cold the cola in the machine, etc.

A few years later, in 1990, John Romkey, who created the first TCP/IP stack, developed the first IoT device. He connected a toaster to the Internet using TCP/IP protocol and could control the amount of time that electricity flows so that the bread can brown as much he wanted.

The term "IoT" was used first time by Kevin Ashton in his presentation at Procter & Gamble (P&G) in 1999 (Ashton et al., 2009), and it has become a standard term. At that time, he already saw the vision that the physical world will be connected in a network in the next generation (Engels et al., 2002).

As he foresaw, the number of connected devices grew up rapidly. By the end of 2020, there were about 8.74 billion IoT devices, and it is predicted that the number will be over 10 billion in 2021 (Holst, 2021) and 25.4 billion in 2030 (Jovanović, 2021). As the 5G network will be provided more and more and popularized, it is also assumed that the number of connected devices and IoT devices will increase more rapidly (G., 2021).

As time passes and technologies have been developed, various IoT applications have appeared in the world. Among many, the following section shows two applications that require high availability, scalability, low latency, and security.

2.2 Examples of IoT Applications

2.2.1 Connected Vehicles

The traditional definition of ’vehicle’ is focused on transporting or carrying something from a place to another place faster. The driver of a vehicle has to decide everything in some situations until he/she arrives at the destination. However, connected vehicles aim towards going beyond the basic definition of a vehicle. There are many terms related to this area, such as connected cars, internet of vehicles, or smart vehicles, but "connected vehicles" will be used in this thesis.

The most basic function of connected vehicles is to let drivers arrive at their destinations safer by detecting possible dangers while driving and more efficiently by considering the current traffic. There are a few types of communications that can help to accomplish the goals. In order to detect dangers from other objects, such as other vehicles or pedestrians that a driver cannot see through windows or mirrors, it is possible to use sensors and cameras. With them, vehicles can collect information and may exchange it with other vehicles (V2V, abbreviated from "Vehicle To Vehicle") or pedestrians (V2P). Since it is important to consider the traffic density to make the driver arrive at the destination faster, "Vehicle to Infrastructure" (V2I) communication and V2V can be used for mitigating traffic congestion (Barrachina et al., 2015, VTS, 2021).

Another example of functions of connected vehicles is parking assistance or self-driving called "autopilot" in many companies. In this case, cameras and sensors need to collect data such as traffic lights, traffic signs, road markings, or the distance between the vehicle and other vehicles nearby. Based on the collected data, information such as traffic density can be considered to provide a better experience to drivers. Such data has to be processed and analyzed in real-time.

In addition, there can be more functions such as vehicle management, entertainment, well-being, etc. (Richard Viereckl, 2014). Communications such as "Vehicles to Cloud" (V2C) or "Vehicles to everything" (V2X) can also be useful in order to achieve such goals (CAAT, 2021).

There are many challenges of connected vehicles to be solved for providing the functions mentioned above.

Security: One of the major issues to be considered is the security of data. Since the data collected from sensors, cameras, or GPS can be personal, companies have to do their best to protect data safely.

Latency and Availability: Another major challenge is the latency of the network that may be directly related to traffic accidents. (Sasaki et al., 2016) shows that 150ms of latency caused vehicles to deviate from the path even 40%. Therefore, it is essential to achieve low latency for the functions.

Availability can be understood in the same context. If there is a downtime of the service of a connected vehicles, it may cause a serious traffic collision.

Scalability: The amount of data generated by a vehicle is different depending on the situation. For example, when a vehicle is parked, it does not generate much data. However, a lot of data has to be generated and processed while driving a vehicle (IBM, 2018). It means that the amount of resources assigned to the system for processing data has to be changed automatically according to the situation.

According to IoT Analytics, "Transportation/Mobility" was ranked No.2 in the global share of enterprise IoT projects in 2020. The application occupied 15% of all projects in 2020, whereas 11% (and was ranked 4) in 2018 (Scully, 2018, Scully, 2020). There have already been many commercial products, and more and more companies are getting involved in this area.

2.2.2 Smart Factory

As IoT gets developed, industries also introduced IoT in their systems in order to produce meaningful data and increase productivity. Industrial IoT (IIoT) means to make industrial processes and entities connected to the internet (Aazam et al., 2018a). Among many applications, smart factory occupies the biggest market of IIoT (Research, 2019). The goal of smart factory is to develop traditional manufacturing industries with IoT systems in order to make them more agile and efficient. For fulfilling such goals of smart factory, there have to be sensors and actuators through which inefficiencies can be discovered faster (Posey, 2021). The paper (Aazam et al., 2018b) says that smart factory also includes the components such as robots, machines, devices, business processes, and people; whereas these components are connected with IoT and generate a huge volume of data 24/7.

Therefore, this application requires two features: high availability and scalability.

In addition, security, latency are also important issues to be considered in smart factories. The followings explain them in more detail.

Security: As the number of connected devices grows, there will be more threats such as intrusions into sensitive private business data for accessing or destroying (Basir et al., 2019). There has to be a way to protect such data.

Latency: For smart factories, it is important to have a strong data center to process a huge amount of data. Since there must be much latency-sensitive data that has to be processed in real-time, to make the latency between the devices that generate data and the data center acceptable is an important matter (C. Shi et al., 2018).

Availability and reliability: Availability and reliability are essential matters to be considered to get succeeded for many manufacturing factories that are operating 24/7 (Liberg et al., 2020, Masters, 2017). There has to be no downtime, and it should be recovered as soon as possible, even if it happens.

Scalability: Since it costs a lot to adopt new technology and is not guaranteed to earn more financially by adopting IIoT to their system, they will start with a small scale (businesswire.com, 2021). When the owner of a manufacturing business wants to increase the size of the system, there have to be more resources that can process the increased amount of data generated in the expanded environment. However, if scalability was not considered when the business had introduced an IIoT system, there will be malfunctions caused by the lack of resources and overload of the resources. In the end, it will hinder the growth of the business (Ferner, 2021).

According to IoT Analytics, "Manufacturing/IIoT" was the most popular application that occupied 22% of all IoT projects (Scully, 2020) in 2020. Considering the trend that it has been ranked No.1 or 2 since 2016 (Bartje, 2016, Scully, 2018, Scully, 2020), smart factory (or IIoT) is a very promising IoT application.

2.3 High Availability (HA) and Scalability in IoT

High Availability (HA): High availability is, as the term infers, the characteristics of a system that can maintain the intended services always available even in situations that some components of the system get failures. It is possible to achieve high availability by removing single points of failure. In order to fulfill such a goal, the system components that have the possibility of failures have to be redundant and distributed so that other components can cover the failure of a node. HA is indispensable in any production environment because a failure to provide services leads to financial loss. According to Gartner, the cost of network downtime for a company is about $5,600 per minute on average (though it may vary depending on the business) (Lerner, 2014).

As mentioned in the sections above, high availability is essential for both connected vehicles and IIoT. In the case of the autopilot of connected vehicles, downtime of the system may cause serious accidents, which may threaten the life of drivers. In the case of IIoT, too, many factories run 24/7, which means that the system has to be always available without any failure (Liberg et al., 2020, Masters, 2017). In addition to the mentioned examples of IoT applications, configuring the system high-available is the key to success for many vertical IoT applications (Yang and Kim., 2019).

Scalability: Scalability can be understood in two ways: assigning idle resources more to the system to process increased data and adding more resources to the existing system to increase the capability.

In an IoT application where the amount of data fluctuates unexpectedly, it is important to have the ability to assign more resources when needed and withdraw the resources after the surge ends. On the other hand, in an IoT application that contains numerous devices that generated a huge amount of data, it is important to have the ability to scale out or up the amount of resources according to the amount of data to be processed.

Let us again consider the IoT applications mentioned above. A connected vehicle generates data much more when the owner drives than when it is parked (IBM, 2018). If the amount of assigned resources stays the same, the amount will be lacking when driving, and it will be idle when parked. So, according to necessity, the amount of resources has to be scaled elastically. In IIoT, the amount of generated data will be proportional to the number of connected machines. It means that more resources will be required when adding new machines to the production line. If the existing system has to be re-organized and re-structured whenever putting the additional resource to the processing system, it will cost a lot whenever more resource has to be added, or the existing one has to be changed (Ferner, 2021).

In summary, a high available system can provide the intended services without downtime, and a scalable system can control how much resource is assigned based on the amount of data and add new resources gracefully without affecting the existing system. High availability and scalability are essential matters to be considered in many IoT applications.

Especially the IoT applications such as connected vehicles and IIoT require high availability and scalability for fulfilling the goals of the applications and provide functions that give good experiences to the users.

2.4 IoT and Computing

2.4.1 Cloud Computing and IoT

Definition of Cloud Computing: Cloud computing is a platform that provides computing resources such as servers, storage, databases, and services, etc., via Internet (Mell and Grance, 2011; Frankenfield, 2020). The reason why the resources are called cloud is that users access the resources remotely. When a user needs some computing resources, it is provisioned anytime (on-demand). In addition, cloud computing also has the characteristics such as fast elasticity, automized and optimized control of resource utilization (Mell and Grance, 2011). There are four deployment models of cloud computing—private cloud, public cloud, hybrid cloud, and community cloud, and three service models —Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS) (Mell and Grance, 2011).

Benefits of Using Cloud Computing: IoT devices have clear limitations in computing capacity and storage, which means that it is not possible to process all data locally or to save the generated data in their local storage (Shu et al., 2020). For these reasons, data generated by IoT devices have been moved, processed, and analyzed by using cloud computing techniques in general (Ghosh et al., 2019). The most basic and common way of using cloud computing is to configure a network that has a gateway that bridges the cloud and IoT devices. The IoT devices sense or collect data and send it to the gateway. Then the gateway forwards the data to the connected cloud to analyze and generate meaningful results based on collected data (Pourqasem, 2018).

Limitations of Cloud Computing in IoT: There are limitations of cloud computing that are fundamentally inevitable because of its structure.

Latency: According to (Vu et al., 2019), the network latency would be up to 100ms more when the regions where communication happens are different (east side and west side of U.S.). Of course, the farther the physical distance between endpoints is, the more increased latency there will be. When providing a service from a cloud, it is impossible to make all users of the service be in the same region unless the service is very small and restricted to a small place. Therefore, cloud computing is not appropriate for latency-sensitive services in general. The IoT application "connected vehicle" explained above in section 2.2.1 is one of the representative latency-sensitive services. By using only cloud computing, it cannot fulfill low enough latency. The latency of cloud computing does not satisfy the requirements of IIoT either (Caiza et al., 2020).

Security: All devices connected to a network are exposed to various types of attacks. Since IoT is fundamentally based on collecting data and the data can contain important and confidential matters, security is the most important matter to be considered. There can be threats and attacks while using cloud computing for IoT devices. There can be three types of security threats in IoT using a cloud. The first type is attacks while an IoT device sends/receives data from the cloud. The second type is attacks against the cloud where data is saved. The third type is attacks directly against IoT devices (Choudhury et al., 2017).

Network Bandwidth and Scalability When a large volume of data generated from IoT devices is transmitted to the platform deployed in the cloud, the ingress network bandwidth of the cloud has to be high enough. It will cost a lot to provide the service without any problem. Let us think about the case of a self-driving vehicle. It is known that google's autonomous car generates around 1GB of data per second (when driving) (Rijmenam, 2013). Even if only a few tens of vehicles use the service simultaneously, the ingress bandwidth demand to the cloud will be really high if all of the generated data has to be processed in a cloud. If it is a few hundred vehicles, it will be very infeasible to provide the service normally.

In the case of a connected vehicle, network bandwidth actually does not need to be high for 24 hours since it does not generate much data when a vehicle is parked. However, in the case of an application that statically generates a high volume of data 24 hours every day, the network bandwidth has to be maintained high all the time, causing high cost. Smart Factory of IIoT mentioned above in section 2.2.2 can be an example of the case. Therefore, using cloud computing for such IoT applications is not appropriate to achieve their goals and implement the functions they provide.

As explained in the sections above, scalability is the ability to scale the resources according to the amount of workload. Since the network bandwidth has to be also scalable for the system to make scalable (Zenlayer, 2017), it costs extra when putting additional resources into the system. Suppose a system that can automatically put additional computing resources. The number of devices connected to a cloud of the system rises, so the amount of workload on the platform also grows. At some point, there have to be more computing resources to distribute the workload. Even though more resources are added to scale up the system automatically, adding more resources is meaningless because the network bandwidth still stays the same.

In cloud computing environments where the resources are in a remote area, just putting additional computing resources to deal with the increased amount of data cannot solve the problem completely. It is because the network bandwidth has to be also increased together along with the amount of data traffic to scale up the system, as an example scenario showed right above this paragraph.

Many cloud services already provide auto-scaling techniques, including the expansion of network bandwidth. However, considering the cost, using cloud computing may not be a good option for many IoT applications, including connected vehicles and smart factories.

2.4.2 Edge Computing and IoT

The number of IoT applications that require faster response time or generate a lot of data even in a second has been increased. As mentioned above, the requirements of many applications cannot be fulfilled in cloud computing systems due to their architectural limitations. In order to overcome the limitations, it is natural to decide to find ways to process data at the close side of where data is generated (Singh, 2017). It was the main reason why edge computing emerged into the world.

Definition of Edge Computing: Weisong Shi, in his paper, defines "edge" as "any computing and network resources along the path between data sources and cloud data centers" (W. Shi et al., 2016). In edge computing environments, the resources for processing data are at the edge side of a network. It is possible to process the data generated in the local network much more efficiently by using edges to do computational tasks instead of the cloud. Edges may completely or partially substitute for the cloud according to the type of systems. There is another term "fog computing," used for representing a similar concept. As the term implies, "fog" represents the resources that are closer to the devices that generate data than "cloud" computing resources. Many researchers see "fog" resources the same as edges (Giang et al., 2018), but many others see it as the layer between edges and cloud (Caprolu et al., 2019).

Benefits of Using Edge Computing:

Low Latency: Since edges are at the edge side of the network and data is generated from the devices that are in the same network, latencies are much lower in edge computing environments than in cloud computing environments. Subhadeep Sarkar and his team had in the paper (S. Sarkar et al., 2018) had an experiment to compare edge computing and traditional cloud computing. When the number of terminal nodes is $4 * 10^4$, the latency of the case that 95% of data is processed in edges is only 1.7 seconds, whereas the latency of when using cloud computing is 9 seconds, and for the number is $10 * 10^4$, 10 seconds and about 1 minute. Therefore, edge computing shines at processing data with low latency when there are many terminal nodes. It means that edge computing is an ideal technology for IoT.

Security and Privacy: When edges can process and store data generated by the devices in the same network, some security and privacy matters disappear. Since the only way to steal data is to access the edges or the local network, the range of attacks becomes much narrower than when using cloud computing. Of course, there are also many challenges to protect data at the edges since they have limited resources, and there are also limited methods for the limited resources (W. Shi et al., 2016). However, by just setting a WiFi password of the network more complicated where edges are located and using a proper wireless security protocol such as WPA3, it is possible to protect the network safer from intrusions and attacks (W. Shi et al., 2016). Encrypting data between devices and edges will enhance security because attackers can utilize only insufficient information unless the credentials used for accessing edges or the system are revealed.

Network Bandwidth and Scalability: The fact that data is processed and analyzed at the edge side means that only a small amount of filtered data will be transmitted to the cloud. Therefore, the network bandwidth is greatly lower than using cloud computing (Satyanarayanan, 2017). As explained in the previous section, it can be related to the matter of scalability. Since the increased amount of data will be processed at the newly added edge devices, the data traffic to the cloud does not increase much. Then the network bandwidth does not need to be increased. In other words, it costs less when scaling up in edge environments than scaling up the computing resources in cloud environments.

Limitations of Edge Computing: Edge devices have only limited resources because of their size. Initial edge devices were small single-board computers in many cases, such as Raspberry Pi or Odroid (Pahl et al., 2016, Morabito, 2017). Because of the lack of resources, some regarded the attempts to use such devices as resources for processing data generated at the edge side as implausible theories (Varghese et al., 2016). However, the capability of single board computers is growing fast and has become powerful enough to be used as edge devices (Qureshi et al., 2019). It is still important to use the limited resources efficiently in edge computing environments (Xu et al., 2017). One good solution for utilizing resources effectively is using containers for deploying applications. The section 3.2 will deal with this matter in more detail.

2.4.3 Overview of Cloud Computing and Edge Computing

In this subsection, in order to highlight the differences between cloud computing and edge computing, we compare them using the Table 2.1. This comparison is performed to highlight the potential of cloud computing and edge computing in supporting scalability and high availability concerns in IoT applications. The table compares the architecture, resources, latency, security, bandwidth, and scalability.

Table 2.1: Comparison of Cloud Computing and Edge Computing.

	Cloud Computing	**Edge Computing**
Architecture	Centralized	Distributed
Resources	Less Limited	Limited
Latency	High	Low
Security	Low	High
Bandwidth	High	Low
Scalability	Yes, high cost considering the network bandwidth	Yes, by adding new edge resources

3 Containers in Edge Environments

Containerization technologies were acknowledged first in cloud computing environments. However, it is regarded as a suitable virtualization tool in edge computing environments too. It is because containers are lightweight even though they can provide many benefits such as fast deployment and good performance in terms of fault tolerance and caching (Ismail et al., 2015).

This chapter shows what containerization technology is and why it is better than VM technology. Furthermore, the last section explains the reason why containers are suitable in edge environments.

3.1 Containerization Technology

3.1.1 Definition

Containerization is one of the virtualization techniques, which has got popular in recent years along with micro-service architecture and DevOps (Logic, 2019). A container is a unit that packages a single application that performs specific tasks. Each container can also have all of the necessary things for running the application, such as libraries, files, runtime, and configurations (Google, 2021c). Comparing to virtual machines (VMs), the major virtualization technology, containers are very lightweight because a container does not include anything else than what is needed for running an application. Every VM needs to have its own operating system (OS), including the kernel. It means that there has to be more storage for OS, the usage of resources such as CPU and RAM will be higher than the necessity for the deployed application in a VM. In other words, using VMs costs more than using containers for deploying applications. This factor led many systems to move from using VM-based virtualization techniques to containerization techniques (H, 2020).

3.1.2 Docker

Docker[*] is the most popular and dominant containerization technology. The starting point that containerization technologies got popular was also the release of Docker as open-source in 2013. According to Datadog, more than 85% of companies that have more than 1000 hosts have at least tried using Docker (Datadog, 2018).

Docker is created based on LXC (Linux Containers [†]), which uses the features of the Linux kernel (especially cgroups and namespace). Docker isolates users, processes, network, etc, and using cgroups using namespaces and hardware resources using cgroups (Docker, 2021, Lab, 2021).

Docker provides high portability so that the application can be independent of the configurations on specific machines. In other words, applications can be deployed in any machines which can run Docker containers. In addition, though Docker can be run only on Linux when the first version of Docker was released in 2013, it can be run on Windows OS and Mac OS too since 2016 (Docker, 2021).

3.1.3 Open Container Initiative (OCI)

Docker indeed led many people to the new world where container technologies are used for software deployments. However, it could not satisfy all of the engineers' needs, and not everyone agreed to use the container format of Docker as the standard (Busser, 2020). In order to solve such problems, Open Container Initiative (OCI)[‡] was founded in 2015 June by the members such as Docker, CoreOS, Google, AWS, Microsoft, etc., in order to define the industry standard on container image formats and runtimes. Therefore, there are two specifications offered by OCI. The first specification is 'OCI image spec' that defines 'how an OCI Image has to be generated.' The second one is 'OCI runtime spec' that defines 'how the decompressed OCI image bundle on disk has to be run' (A. Sarkar, 2019, OCI, 2021). If any container image is created following the OCI image spec, it is possible to run the image on any container runtime that is implemented following the OCI runtime spec.

---

[*] https://www.docker.com

[†] https://linuxcontainers.org/lxc/introduction/

[‡] https://opencontainers.org

3.2 Why Containers on Edges

It is more important to use the resources efficiently in edge computing environments than cloud computing environments because edges have limited resources (Xu et al., 2017). Virtualization technology is useful for fulfilling the goal to use resources effectively in terms of cost.

In cloud computing environments, both VMs and containers are good options for virtualizing servers and enhancing the utilization of hardware resources since the amount of resources is less limited than edge computing environments. However, as mentioned in section 3.1, running VMs is heavy for edge devices. This problem will be a too big disadvantage for edge computers that have a limited amount of resources. On the other hand, containers are lightweight. By using containerization technology, it is possible to have the advantages of virtualization in edge environments. In addition, the performances in terms of CPU, memory, disk I/O, and network do not get lowered much. Comparing to bare-metal machines, the difference in terms of performance is small and negligible. In addition, there are only small insignificant differences when testing on bare-metal and containers in the matter of power consumption too. In some cases, running containers show even better performance than running on bare-metal machines (Morabito, 2017). Therefore, it is beneficial to use containers for and deploying and running applications in edge environments.

4 Kubernetes

4.1 Container Orchestration

Container orchestration is the management of containers' lifecycle. It will be impossible to deploy containers manually when there hundreds or thousands of containers. Therefore, it is mandatory to automate these kinds of works. With an orchestration tool, it is possible to automate not only deployments of containers but also recovery of failed containers, load balancing, allocation of resources, etc. In addition, it is possible to monitor the health of containers (RedHat, 2021). There are three well-known tools Kubernetes, Docker Swarm, and Apache Mesos. Among them, Kubernetes is the most dominant container orchestration tool (StackRox, 2021). This thesis will deal with only Kubernetes in detail.

4.2 What is Kubernetes

Kubernetes is a container orchestration tool originally designed by Google engineers as their internal platform named 'borg' based on their accumulated knowledge and experience on containerization. Google open-sourced Kubernetes in 2014 (Google, 2021d). In 2015, Cloud Native Computing Foundation (CNCF) was established, and Google donated Kubernetes to CNCF, so CNCF maintains and controls the Kubernetes project so far (CNCF, 2015). As Docker used to be regarded as the standard of containerization technology, Kubernetes has become an unofficial standard for container orchestration. Comparing to the other dominant tool Docker Swarm, it is more complex for installing and slower for deploying containers. However, it is more tolerant to faults, provides more functionalities, and supports auto-scaling (Naser, 2017). Therefore, many large-scale companies such as Google, Reddit, and the New York Times that need a number of microservices use Kubernetes in order to provide their services and maintain their system.

Kubernetes is called K8s, where 8 is the number of letters between K and s. A K8s cluster is installed and configured as follows in general. First, Container Runtime (such as Docker, containerd, or CRI-O) should be installed. And then, the components for the master node (Control Plane) are installed, and Worker Nodes join the master node. These processes are generally done with the tool kubeadm in a production environment. And then, by deploying Container Network Interface (CNI), an overlay network is created to implement networking in the Kubernetes cluster.

When K8s cluster is created using kubeadm K8s, at least 2GB RAM per machine and 2 CPUs are required to install the components and configure the cluster.

The following subsections will explain the necessary components for deploying applications and other important concepts in more detail.

4.3 Object

Kubernetes objects are the 'records of intent' used to represent the state of a cluster. "Object spec" states the intent (characteristics) of the object, and "object status" represents its current state. The most fundamental work the Kubernetes cluster does is to check regularly the status of objects and make them the same as defined in the object spec. The Control plane in a cluster takes this role (Kubernetes, 2020b).

When an object is generated, edited, or removed, all requests have to pass through Kubernetes API. API requests have to contain the information on the "intent" of the object in JSON format. In most cases, API is used through the tool "kubectl," and object spec is given in YAML format and converted to JSON during the API request (Kubernetes, 2020b).

Pod: A Pod is the most basic unit for deployment. In other words, Kubernetes does not separately deploy or manage each container but does with the unit Pod. A Pod has to contain at least one container, and it is also possible to run multiple containers related to each other in a Pod. Each Pod has its own IP address, and containers in a Pod share the IP address. In addition, containers in a Pod share a disk volume and network resources (Kubernetes, 2021m).

Volume: Volume can be simply described as an external disk of a container. Local storage is defined again whenever a Pod is generated or restarts. This means that the data in the local volume does not remain. Therefore, if there are some contents that need to be permanently stored, a volume object is necessary. A volume in a Kubernetes cluster can be shared among containers in a Pod (Kubernetes, 2021t).

Namespace: A namespace is used for dividing a Kubernetes cluster. Pods and Services can be generated and managed differently in different namespaces. For example, if there are three different environments in a cluster, it can be managed by dividing a cluster into three namespaces. It is possible to assign a quota of resources to each namespace and make users have different access rights on different namespaces (Kubernetes, 2021l).

Service Service is the object that exposes the application running on Pods. It can be the exposure to other components only in the cluster or to outside the cluster.

In general, there are multiple Pods in a cluster, and Pods can have labels. All the Pods that have the same label specified in the ’selector’ field of a Service are the members of the Service (Google, 2021a). It means that all traffic to a Service is distributed to the Pods that are the members of the Service. Kubernetes official documents explain four types of Service. The following items are detailed explanations of the four types of Service (Kubernetes, 2021q).

- ClusterIP

  It is the default type of Service. A virtual IP is assigned to the Service, which is accessible only inside the cluster. For this reason, this type of Service is used basically for exposing Pods to other apps in the same cluster. However, by using kube-proxy (which will be explained in detail below in section 4.5), it is possible to make the Service accessed from outside of the cluster (Kubernetes, 2021q).

- NodePort

  As the name "NodePort" explains, it uses the ports of nodes in order to expose a Service to external traffic. All traffic coming through the port designated in the nodePort field is forwarded to the "NodePort" Service. There are a few drawbacks of using NodePort Services. Only a limited range of ports (30000-32767) can be used, and it is not possible to provide multiple Services through one port. In addition, if a VM’s IP address is changed, the Service IP address is also changed (Kubernetes, 2021q).

- ExternalName

  This Service is generally used when accessing outside of the cluster from inside. The Service forwards requests to the external IP in the "externalName" field (Kubernetes, 2021q).

- LoadBalancer

  LoadBalancer type Service exposes an application outside the cluster using a load balancer. An external IP is assigned to the Service, which is the IP address of a load balancer. All traffic to the IP will be delivered to the Pods that are the members of the Service (Kubernetes, 2021q).

LoadBalancer type Service by default generates NodePort and ClusterIP together since LoadBalancer is based on NodePort, and NodePort is based on ClusterIP (Google, 2021b). As the Figure 4.1 shows, traffic from outside the cluster arrives at the port of the LoadBalancer type Service. And the traffic is delivered to the Service via NodePort. Since NodePorts are allocated, it is also possible to access the Service directly via the NodePorts. In recent versions of Kubernetes (since v1.20), it is possible to disable to allocate NodePorts for LoadBalancer type Service (Kubernetes, 2021q).

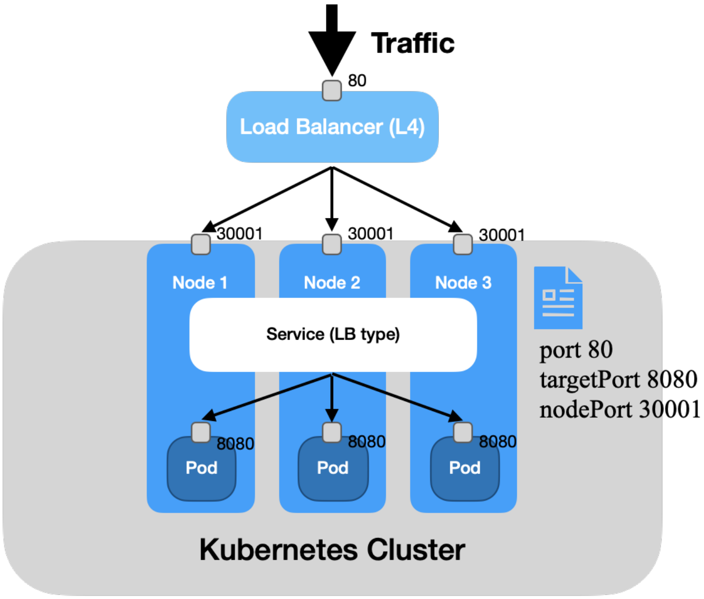


Figure 4.1: LoadBalancer and a Kubernetes Cluster. ref: Kubernetes Advocate, medium

Kubernetes clusters are generated in a production environment using a cloud providers' platform in general, and Kubernetes provides a function to provision a load balancer from a cloud infrastructure automatically. Therefore, load balancing services provided by the cloud providers can be configured to assign an IP address for a LoadBalancer type Service (Kubernetes, 2021q). For example, when a Load Balancer type Service is created in a cluster using Amazon Elastic Kubernetes Service (EKS), Network Load Balancer (NLB) or Elastic Load Balancer (ELB) service of Amazon Web Services (AWS) is automatically provisioned in the cluster. In similar ways, a cluster in Google Kubernetes Engine (GKE) can use Cloud Load Balancing of Google Cloud Platform (GCP), and a cluster in Azure Kubernetes Service (AKS) can use Standard and Basic Load Balancer for LoadBalancer type Services.

On the other hand, when running Kubernetes clusters on bare metal machines, a load balancer has to be configured manually. The load balancer implementation has to be installed and deployed over the cluster in order to assign external IP to a LoadBalancer type Service. MetalLB[*] is the most well-known network load balancer implementation in on-premise Kubernetes clusters, but it is still a young project whose current version is beta. MetalLB provides manifest files, and it is possible to apply the provided manifests by using Kubectl apply command. This process will add the "metallb-system" namespace and MetalLB components (a Deployment controller and a speaker DaemonSet) under the namespace (MetalLB, 2021). After then, a ConfigMap object that states the configuration of MetalLB, such as the type of protocol, the range of IP address that will be assigned to Services, etc., has to be created. In MetalLB, there are two modes, Layer 2 and Border Gateway Protocol (BGP). Layer 2 uses virtual IP addresses, and BGP uses a physical load balancer and its IP address (CONFIGURATION, 2021).

By using such load balancers, it is possible to assign an external IP to LoadBalancer type Services.

For Layer 7 routing or load balancing, there has to be also Ingress and Ingress controller, which is explained below in this section.

Ingress: Ingress is the way to expose Services outside the cluster. Layer 7 traffic can be handled by using Ingress resources. For example, it is possible to define TLS/SSL certificate configuration or HTTP(S) routing when creating Ingress objects. It is also possible to expose multiple Services and route traffic to an appropriate Service based on HTTP(S) paths (Kubernetes, 2021e). As Figure 4.2 shows, routing rules are defined in an Ingress object, and traffic is routed to the Services based on the rules. Since Ingress is just a declarative object which defines rules, it does not work by itself. There has to be also Ingress Controller that is the actual part for handling traffic and routing the traffic to Services. Ingress Controller will be dealt with right below.

Ingress Controller: Ingress Controller is not a type of object, but we explain in this section since it is closely related to Ingress object. As explained in the content "Ingress" right above, Ingress Controller is necessary in order to use an Ingress object. Only after an Ingress Controller is configured, the rules stated in an Ingress object can become activated.

---

[*] https://metallb.universe.tf

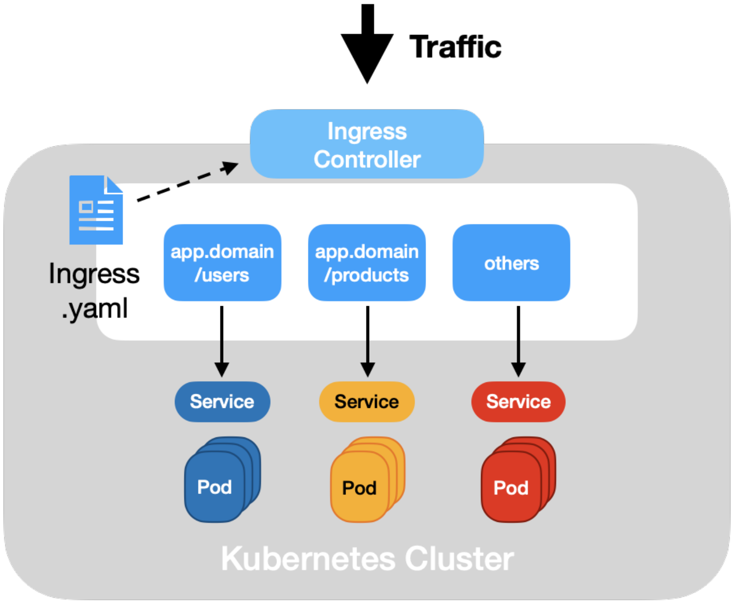


Figure 4.2: Ingress in a Kubernetes Cluster. ref: NGINX

As Figure 4.2 shows, the real component which handles the requests is Ingress Controller. The controller routes HTTP(s) traffic based on the routing rules of the Ingress object (Kubernetes, 2021f).

When using a cloud platform to run Kubernetes clusters, it is possible to provision Layer 7 load balancing services of the cloud provider. GKE provides the service ’Ingress Controller,’ and EKS provides Application Load Balancer (ALB). AKS does not provide any Layer 7 load balancing service.

On the other hand, Kubernetes clusters in bare metal machines need to deploy the Ingress Controller manually on the cluster. Unlike the load balancer for LoadBalancer type Service, there are many open-sourced Ingress Controllers that can be installed on a Kubernetes cluster. The Ingress Controller that is officially managed by Kubernetes is NGINX Ingress Controller, but Kong, HAProxy Ingress, Traefik, and many Envoy-based ingress controllers are used in many production environments.

Since there are many options for Ingress Controller, many Kubernetes users deploy such open-sourced ones even if they run clusters in a public cloud service.

Controller: It is possible to deploy applications using the objects mentioned above. In order to manage objects to maintain the "intended" states that are defined in their object specs, controllers are necessary to use.

- Replication Controller (RC)

It is the most basic controller of Kubernetes. This controller is comprised of three major fields (Kubernetes, 2021o).

  - Replicas

    The value in the field "replicas" decides the number of Pods. RC generates or removes Pods in order to maintain the number of Pods as stated in the field.

  - Selector

    Label in the selector field decides which Pods the controller will manage. In other words, the controller controls all of the Pods that have the label stated in the selector field.

  - Template

    When a Pod is newly generated, information on the Pod (container image, labels, port, etc.) is needed. Such information is written in the field "template."

- ReplicaSet (RS)

  It is not much different from the Replication Controller. RS uses a selector based on set (in, notin, exists), whereas RC uses a selector based on equality (=, ==, !=) (Kubernetes, 2021n). Kubernetes officially recommend using higher-level API object "Deployment" rather than using Replication Controller or ReplicaSet directly (Kubernetes, 2021p). The Deployment object is also dealt with below in the same section.

- DaemonSet (DS)

  Unlike RS, DS controller runs only one Pod on each node. In other words, when running Pods by using DaemonSet, all nodes in a cluster always have one Pod. (It is possible to set it as zero, but it is impossible to increase the number of pods.) A Pod is automatically added and run on the node when a node is added to the cluster. In addition, when a node is removed, a Pod in the node is removed, not moved to another node. Because of these kinds of characteristics, the DaemonSet controller is generally used for such purposes: 1. to run Pods which have to be always in a cluster (system Pods, Pods for monitoring nodes, running logging collectors, etc.) 2. to run Pods which have to be run only on specific nodes that have specific resources (e.g., nodes which have SSD, not HDD) by using nodeSelector field. (Kubernetes, 2021c)

- Deployment (Deploy)

Deployment is the most basic controller for deploying stateless apps. By using this controller, it is possible to track all deployments and updates of Pods. Using the history, it is also possible to roll back for a deployed Pod. In short, the Deployment controller can be understood as ReplicaSet + history (Kubernetes, 2021d). The official document of Kubernetes suggests generating objects using a Deployment controller rather than generating each object independently. In a production environment, Deployment is used rather than using Replication Controller or ReplicSet directly (Kubernetes, 2021p).

- Stateful Set (SS)

  Stateful Set is similar to Deployment. However, Deployment is stateless, so the data generated in a Pod will be removed when the container is removed. By using a Stateful Set controller, a Volume is mounted, and data can be stored and maintained even if a Pod restarts (Kubernetes, 2021r).

- Job & Cron Job (CJ)

  When a job is completed after one-time execution, Pods do not need to run all the time like a web server. It is possible to run Pods only when executing some works. Job controller handles such workload model. When a Job controller terminates, Pod also terminates (Kubernetes, 2021g).

  Cron Job is like a "cron" command in the Linux/Unix system, which automates batch processes. In addition to the settings for a Job controller, there is a "schedule" field that states the repetition condition (Kubernetes, 2021b).

4.4 Taints and Tolerations

In a Kubernetes cluster, taints and tolerations are used in order to schedule Pods. A node can be tainted with key, value, and effect through kubectl with the command "kubectl taint nodes node-name key=value:effect". A Pod can have tolerations in the spec in YAML form. Pods can be scheduled in nodes only when the Pods' tolerations correspond to the node's taints. If there is no node, the Pod cannot be scheduled and will be in "pending" status until any tainted node can accept it (Kubernetes, 2021s).

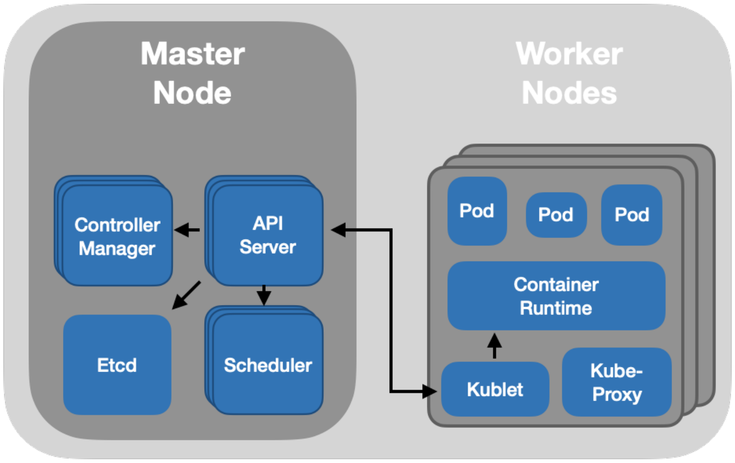


Figure 4.3: Architecture of a Kubernetes Cluster. ref: Kubernetes Official Document

4.5 Architecture

In order to understand Kubernetes, it is important to know the architecture of a cluster. A cluster of Kubernetes is comprised of the master node(s) and worker nodes. A master node controls and manages the whole cluster. Worker nodes are physical or virtual machines where containers are deployed, and Services can be provided. Figure 4.3 shows the architecture of K8s briefly. The rest of this section explains and describes each component in the architecture of K8s.

Master Node:

- API Server

  In Kubernetes, all functions are provided through RESTful API, and API Server in a master node serves the operations. Therefore, all of the components in a Kubernetes cluster have to send an API request to the API Server in order to interact with another component in the cluster, as Figure 4.3 shows (Kubernetes, 2021k).

- Etcd

  Etcd is a distributed key-value storage, which works as a database in a Kubernetes cluster. All of the configurations or states of the cluster are saved. It communicates only with the API server as Figure 4.3 shows. It means that other modules can access the data of etcd only via the API server (Kubernetes, 2021k).

- Scheduler

In Kubernetes, scheduling stands for matching Pods to the most appropriate node considering available resources. Scheduler first filters which nodes have enough resources to run the considering Pod and then scores all of the nodes. Based on the score of each node, the scheduler selects the node which got the highest score and allocates the Pod to the node (Kubernetes, 2021k).

- Controller Manager

  As the name implies, a Controller Manager manages controllers. Here the 'controller' does not mean only the controllers for managing Pods explained in "Controller" part in section 4.3. In addition to workloads, controller manager manages the state of nodes, namespaces, service accounts, etc. (Kubernetes, 2021h, (Kubernetes, 2021k)).

Worker Node:

- Kubelet

  Each worker node has a Kubelet, the agent which communicates with the master node's API server. A Kubelet carries out the requests of the master node by making containers keep their status as stated in the manifest files that describe Pods. In addition, a Kubelet sends the status of the worker node to its master node (Kubernetes, 2021j).

- Kube-proxy

  Each worker node has a Kube-proxy daemon that manages network traffics between a master node and worker nodes. When a Service is provided via ClusterIPs or NodePorts, Kube-proxy routes and load-balances it to appropriate Pods (Kubernetes, 2021i). 'Userspace' was a default mode at first, but it is not recommended anymore because it is old and slow. Nowadays, 'iptables' is a default mode, which reduces system overheads. Kube-proxy also supports the 'IPVS' mode that supports higher network traffic throughput than other modes. IPVS mode, in addition, has lower latency than any other mode when proxy rules are synchronized and supports various algorithms for load balancing that are not provided in other modes (Kubernetes, 2020a). IPVS shows better performance for clusters that provide hundreds or thousands of Services. For these reasons, IPVS is suitable for such big clusters.

- Container Runtime

  In Kubernetes, there is a container runtime that runs the container images stated in the spec of a Pod (Kubernetes, 2021a).

Docker has been used as the most common option for container runtime, but Kubernetes is deprecating Docker as a container runtime after v1.20 that was released in December 2020. Furthermore, Kubernetes does not support Docker anymore from v.1.22 that will be released in late 2021. The reason why Kubernetes does not support Docker anymore is that Docker does not follow the Container Runtime Interface (CRI) standard (Jorge Castro, 2020).

CRI was introduced into Kubernetes in late 2016 in order to support more runtimes than Docker only. By implementing the Kubernetes CRI, any container runtime that is compatible with OCI can be connected to Kubernetes and used as a container runtime in Kubernetes (cri-o, 2021). CRI-O[*] is a generic implementation of CRI. OpenShift set CRI-O as a default container runtime (OpenShift, 2021).

There is another commonly used container runtime "containerd" [†]. It is possible to use containerd runtime with the plugin "cri," the implementation of CRI. GKE and AKS have set containerd as the default runtime (Cloud, 2021, Jorge Palma, 2020). Containerd was a part of Docker daemon at first, but it became a separate CNCF project. It became one of the 'graduated' projects in 2019.

4.6 Kubernetes in Edge Environments

There have been many pieces of research on edge computing and various attempts to run Kubernetes clusters on edges. However, Kubernetes clusters on edges are too heavy to edge devices that have only limited resources. For this reason, lightweight Kubernetes distributions have appeared.

4.6.1 K3s

K3s[‡] is a Kubernetes distribution, which was developed by Rancher Labs and fully certified by CNCF. It is specialized for edges and IoT that have limited resources. K3s needs only half memory comparing to K8s (the name originates from this fact), and the minimum requirement for memory is only 512MB though it is recommended to have more than 1GB (Rancher, 2021a). In addition, the size of the binary is less than 40MB with maintaining

---

[*] https://cri-o.io

[†] https://containerd.io

[‡] https://k3s.io

complete compatibility with K8s. In order to achieve these, K3s replaced heavy components with lightweight ones and removed unnecessary features. For example, the default storage backend of K3s is sqlite3 that is more lightweight than etcd3 in K8s. Default Container Runtime is containerd, and ingress controller is traefik (Rancher, 2021b). Another benefit of using K3s is that it is easy to install. It is possible to configure a K3s cluster just by running provided shell scripts, whereas K8s installation using kubeadm requires complicated works.

4.6.2 MicroK8s

MicroK8s[*] is a High Availability Kubernetes distribution that was developed by Canonical, the publisher of Ubuntu. Like K3s, it is a minimized distribution of which the size is only about 200MB. What MicroK8s emphasizes more is high availability or resilence for making the cluster production grade. In MicroK8s, etcd, the database storage is replaced by Dqlite, which is HA SQLite (Canonical, 2021b). MicroK8s officially says that the distribution is suitable for edges and IoT, but, ironically, the minimum requirement for memory is 4GB (Canonical, 2021a). It is even higher than the requirements of K8s.

4.6.3 KubeEdge

KubeEdge[†] is a Kubernetes platform that is also specialized for edges, as the name implies. It was originally developed by Huawei based on their commercial IoT edge platform and contributed to CNCF. It was accepted as a sandbox project in March 2019 and approved as an incubating project in September 2020. Unlike K3s or MicroK8s, KubeEdge's cluster is comprised of cloud side and edge side. A cluster possesses a master node on the cloud side (though it can be extended) and worker nodes on the edge side. Its storage footprint is 66MB and memory footprint less than 30MB (even about only 10MB on the edge side agent (Lai, 2020). KubeEdge takes a different approach to communication than K8s. It communicates through full-duplex asynchronous channels using WebSocket that is message-based. In this approach, the communication overhead is lower (Manaouil and Lebre, 2020). In order to achieve this, there are more components included in the architecture. EdgeController works as a bridge between API Server and EdgeCore. CloudHub is a WebSocket server on the cloud side, and EdgeHub is a client on the edge side. In

---

[*] http://microk8s.io

[†] https://kubeedge.io

KubeEdge, Edged, the more lightweight version of Kubelet, runs on the edge side as the agent which manages containers. There is also a component called MetaManager between Edged and EdgeHub, which works as a manager of metadata (storing to SQLite or retrieving from it) and as a message processor. In addition, KubeEdge supports offline mode so that nodes on the edge side can maintain their functions even when there is no connection with the node of the cloud side (Xiong et al., 2018).

4.6.4 Overview of K8s, K3s, MicroK8s, KubeEdge

Based on the official documents that give information on system requirements or supported OS, this section provides a tabulated overview of K8s, K3s, MicroK8s, and KubeEdge in Table 4.1. The cells of the table were left empty if there is no official data for some items of the table.

Table 4.1: Overview of K8s, K3s, MicroK8s, KubeEdge

	K8s+kubeadm	**K3s**	**MicroK8s**	**KubeEdge**
Company	Google->CNCF	Rancher->CNCF	Canonical	Huawei->CNCF
CPU/Master	2 CPUs	20% of Pi4B BCM2711, 1.50Ghz	—	—
RAM/Master	2GB	512MB	4GB	
CPU/Worker	—	10% of Pi4B BCM2711, 1.50GHz	—	—
RAM/Worker	—	256MB	—	—
Storage Footprint	—	< 40MB	< 200MB	additional 66MB
Supported Architecture	386, ppc64le, s390x, ARM, ARM64	x86-64, ARM, ARM64	x86-64, ARM, ARM64	x86-64, ARM, ARM64
Container Runtime	containerd (default), Docker (deprecating), CRI-O	containerd (default), Docker	conainerd	Docker, containerd, CRI-O, Virlet
Default Ingress Controller	—	Traefik	Nginx	—
Database	etcd	SQLite	etcd	etcd
OS	Linux	Linux	Linux, Windows 10 or MacOS(Multipass), LXD (container)	Linux, Windows 10 or MacOS (Multipass)
High Availability	3+ control plane nodes	2+ server nodes and 0+ agent nodes	< 3+ nodes	HA CloudCore
Suitable for IoT/Edge	X	O	O	O

5 Building High Available and Scalable Kubernetes Clusters on Edges

In this chapter, we show how to configure high available and scalable Kubernetes clusters. We implement the clusters in two different ways and evaluate them in terms of high availability and scalability. In addition, we compare the resource utilization of each cluster. Based on the results, we discuss which cluster can be a better option in specific cases.

5.1 High Available Kubernetes Cluster

It is possible to configure a high available Kubernetes cluster by making both worker nodes and master nodes high available.

The high availability of worker nodes can be achieved when there are at least two worker nodes. It is because an alive node still has to run Pod(s) to provide a service even if one node fails.

On the other hand, the high availability of master nodes can be achieved in two ways. The first way is to cluster at least two master nodes with an external DB. The second way is to make each node of the cluster have embedded DB (such as etcd). In this case, the number of embedded DB in the cluster has to be at least three and always an odd number.

5.2 Scalable Kubernetes Cluster

Kubernetes was basically designed considering the matter of scalability. In Kubernetes, scalability can be achieved in two ways.

The first way is to make the cluster elastically change the number of Pods or the amount of resources assigned to Pods. Each Pod will have only a limited amount of resources according to the spec of its controller. When the amount of data to be processed grows up, and existing Pods cannot process all, the total amount of resources also should increase by increasing the number of Pods or the amount of assigned resources to each Pod. As mentioned in the section 4.3, each namespace may also have a limited quota of resources to prevent Pods in one namespace take all of the resources. So, the total amount of resources assigned to Pods in a namespace will not be more than the quota given to the namespace.

The second way is to add more nodes to the existing cluster. When the total amount of resources has to increase, additional nodes have to be put into the cluster. When using cloud platforms for Kubernetes such as EKS or GKE, the number of nodes will be automatically scaled according to the settings of users. However, the tasks of adding new nodes should be manually dealt with in edge environments. If the process takes a long time and is complicated, Kubernetes may not be suitable for edge environments that need scalability. However, putting additional nodes into the cluster is an easy process that needs only a few lines of commands. A Kubernetes cluster has a token needed for joining a new resource, and a new node can join the cluster using the token value. Both master and worker nodes can be added in this way (but different options have to be used). As mentioned in the section 5.1, when master nodes have to be scaled up, the cluster has to maintain the number of embedded databases (such as etcd, SQLite) as an odd number. It enables that there can be a majority against failed nodes. According to Rancher, having too many etcd members in a cluster can make the cluster slow since etcd uses the Raft consensus algorithm for synchronization of nodes (KIM, 2019).

5.3 Goal

In this thesis, two architectures of the HA Kubernetes cluster were surveyed and implemented in a way to configure at a minimum expense in edge computing environments. The first implementation is with two nodes and an external DB, and the second one is with three nodes with embedded etcds.

The goal of the implementations is to show how the clusters maintain the system available and how the clusters can be scaled. And then, we show that why Kubernetes is a good option for deploying IoT applications that require high availability and scalability.

There are also limitations of each architecture that affect accomplishing complete high availability and scalability of the cluster. They will be discussed and compared, and possible solutions will be provided.

In addition, we compare the resource utilization of each cluster. By comparing two clusters, we discuss which architecture will be suitable for specific applications.

5.4 Implementations

5.4.1 Environments of the Implementations

As explained in section 5.1, it is necessary to have three different machines to configure high available Kubernetes cluster. Since the nodes of clusters have to be on different machines for testing high availability, VM machines were used to imitate multiple physical machines. Multipass*, a lightweight VM managing tool, was used for configuring nodes. Each VM ran Ubuntu 20.04 LTS as an operating system (OS) in order to set up K3s clusters.

Machines were installed on top of the MacBook Pro 2013 Late, which has 2,6 GHz Quad-Core Intel Core i7-4960HQ, 16GB DDR3 1600MHz RAM, SSD storage(read/write speeds are respectively about 1400MB/s 1250MB/s). The resources allocated to each VM are 1 CPU, 1GB Ram, 5GB Disk.

5.4.2 Configurations for K3s, Load Balancers, and Taints & Tolerations

This section explains what Kubernetes distribution and load balancers were used for implementing clusters and why they were used for both test environments.

K3s

In the paper (Böhm and Wirtz, 2021), the author had experiments to compare K3s and MicroK8s. He showed that the utilization of CPU, RAM, and Disk was higher when using MicroK8s than when using K3s. Based on the information, we chose K3s distribution in order to implement two Kubernetes clusters in this thesis. If a K3s master node is installed without any options, some components such as servicelb (load balancer) and traefik (ingress controller) are automatically deployed in the namespace kube-system. However, these components are not related to the tests in this thesis, so the components were disabled. We still kept other components, such as core-dns, local-path-provisioner, or metrics-server, running in test clusters. Default container runtime containerd was used without changing to another runtime.

Load Balancer for Master Nodes

---

*https://multipass.run

HAProxy[*] was used in order to have load balancers that distribute requests to masters. In order to make the load balancer work, the configuration file has to have information on the port that listens to the ingress traffic and the IP addresses of master nodes. The field 'frontend' of the configuration file contains the information on the port, and the field 'backend' has the data of IP addresses. In addition, round-robin algorithm was used for load balancing.

Each node works as a load balancer, and the node elected as a leader serves the load balancing function. In order to make an entry point through which all of the nodes can be accessed, a virtual IP address (VIP) should be assigned to the nodes. We used keepalived[†] for assigning VIP to nodes.

All the nodes have the same VIPs, and each node has a different priority for being elected as a leader. The node that has higher priority than other nodes is elected as the leader node, and all of the traffic to the VIP passes through the leader node. When the elected node is dead, the node which has the highest priority among the remaining nodes is elected and continues the work as the previously elected node did.

With the combination of HAProxy and keepalived, it was possible to have a load balancer for master nodes and make the cluster high available.

Load Balancer for Worker Nodes

As mentioned in Load Balancer part of section 4.3, MetalLB is commonly used in order to expose Pods with LoadBalancer type Services in bare metal Kubernetes clusters. MetalLB supports Layer 2 mode and BGP mode for load balancing. In this thesis, Layer 2 mode was used because BGP mode needs a physical load balancer.

Taints & Tolerations

As explained in section 4.4, Kubernetes uses taints and tolerations in order to make Pods be scheduled in proper nodes. There are built-in taints of which the keys are about the status of nodes such as "node.kubernetes.io/not-ready", "-/unreachable", etc. Pods made by controllers automatically have tolerations that have the field "tolerationSeconds=300" for both keys mentioned above (Kubernetes, 2021s). It means that Pods will not be evicted in 300 seconds after a node fails. Therefore, in the spec of a Deployment controller that creates Pods, it is necessary to override tolerations with the 'tolerationSeconds=0' so that the Pod in a dead node can be created immediately in an alive node.

---

[*] http://www.haproxy.org

[†] https://www.keepalived.org/index.html

5.4.3 2 Nodes (Master + Workers) + 1 External DB Cluster

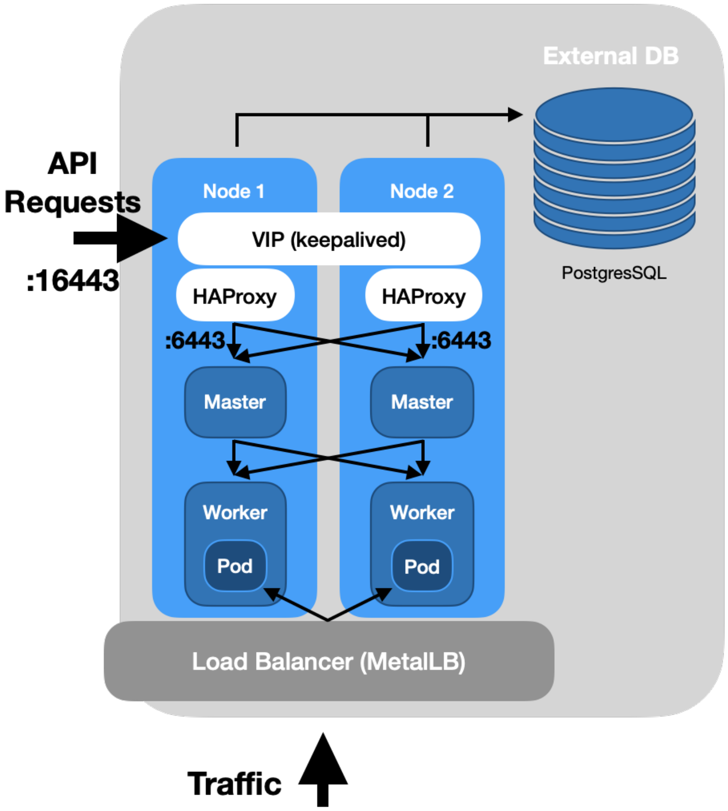


Figure 5.1: Architecture of K3s Cluster with Two Nodes + External DB

External DB

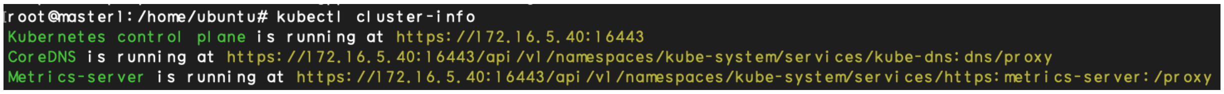
K3s officially supports three types of external DB, MySQL (MariaDB too), etcd, and PostgreSQL. PostgreSQL has a lower image size and consumes less memory and CPU than MySQL and MariaDB. In addition, PostgreSQL is faster for both reading and writing (Jahoda, 2021). For these reasons, PostgreSQL was used for implementing clusters in this thesis. We ran the database node also on a multipass VM with the same amount of resources as other nodes. Since an external DB is used, nodes do not need to have any embedded database. As Figure 5.1 shows, two nodes were connected to the PostgreSQL database to use it as a storage for saving the status of clusters.

In the implementation, the IP address of the node for external DB was 172.16.5.24.

VIP and HAProxy

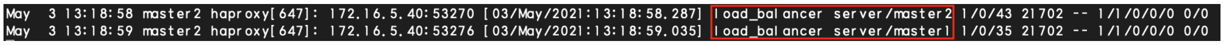
HAProxy listens 16443 port and routes all the traffic. As Figure 5.1 shows, the destinations of the traffic are <Node 1's IP address>:6443 and <Node 2's IP address>:6443 because master nodes use 6443 port for its API server. As Figure 5.3 shows, HAProxy alternately routes between the two nodes.

We assigned 172.16.5.40 to VIP in the implementation as Figure 5.2 and Figure 5.3 show.



```
root@master1:/home/ubuntu# kubectl cluster-info
Kubernetes control plane is running at https://172.16.5.40:16443
CoreDNS is running at https://172.16.5.40:16443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Metrics-server is running at https://172.16.5.40:16443/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy
```

Figure 5.2: All API requests pass through the load balancer.



```
May  3 13:18:58 master2 haproxy[647]: 172.16.5.40:53270 [03/May/2021:13:18:58.287] load_balancer server/master2 1/0/43 21702 -- 1/1/0/0/0 0/0
May  3 13:18:59 master2 haproxy[647]: 172.16.5.40:53276 [03/May/2021:13:18:59.035] load_balancer server/master1 1/0/35 21702 -- 1/1/0/0/0 0/0
```

Figure 5.3: HAProxy log shows that requests are distributed to backend server (2 nodes).

Nodes

As Figure 5.1 shows, both nodes have same VIP. We put priority 1 to Node 1 and 2 to Node 2. Since Node 2 has a high priority, it is chosen as the leader node. When Node 2 fails, Node 1 takes the VIP after being elected as the leader node. Both nodes connect to the external DB when creating a cluster.

In the implementation, Node 1's IP address was 172.16.5.5, and Node 2's one was 172.16.5.6. In section 5.5, we use the name master1 for Node 1 and master2 for Node 2.

Load Balancer (MetalLB)

In the implementation, we assigned 172.16.5.41-172.16.5.49 to be used as the external IP of a LoadBalancer type Service. As the bottom part of Figure 5.1 shows, it is possible to access the applications running on Pods through the MetalLB load balancer.

5.4.4 3 Nodes (Master + Etcd + Workers) Cluster

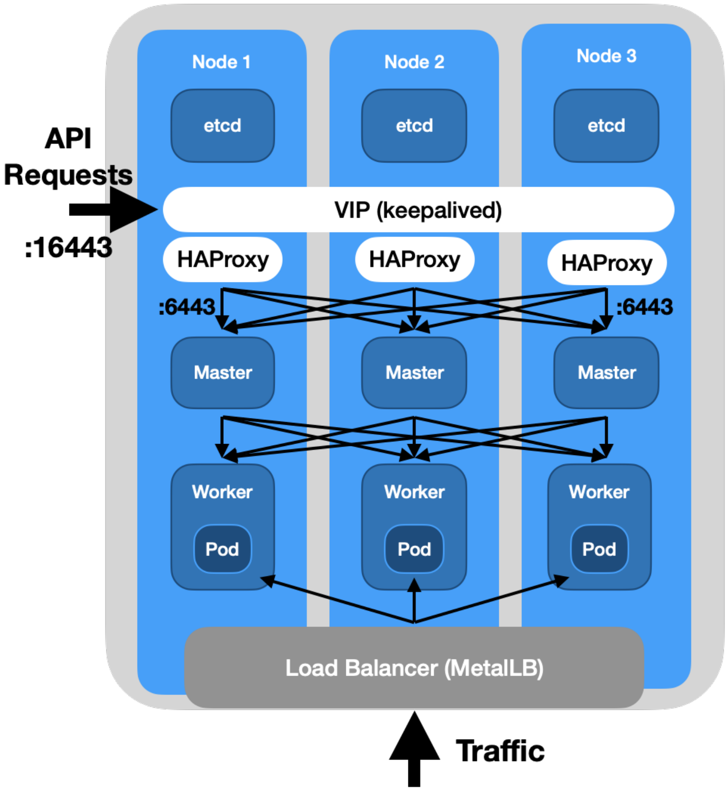
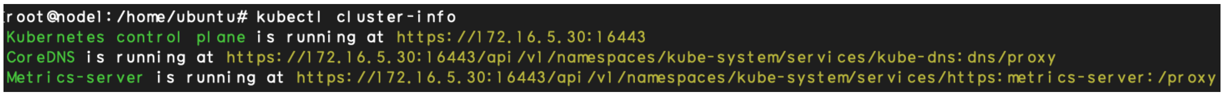


Figure 5.4: Architecture of K3s Cluster with Three Nodes

VIP and HAProxy

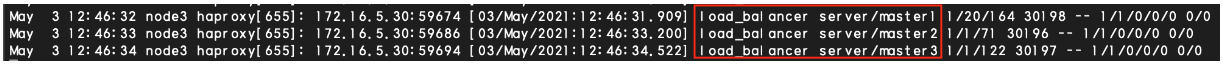
Basically, the configuration for VIP and HAProxy is same as section 5.4.3. Figure 5.4 shows, the destinations of the traffic are <Node 1's IP address>:6443, <Node 2's IP address>:6443, and <Node 3's IP address>:6443. As Figure 5.6 shows, HAProxy alternately routes between the three nodes.

We assigned 172.16.5.30 to VIP in the implementation as Figure 5.5 and Figure 5.6 show.



```
root@node1:/home/ubuntu# kubectl cluster-info
Kubernetes control plane is running at https://172.16.5.30:16443
CoreDNS is running at https://172.16.5.30:16443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Metrics-server is running at https://172.16.5.30:16443/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy
```

Figure 5.5: All API requests pass through the load balancer.



```
May  3 12:46:32 node3 haproxy[655]: 172.16.5.30:59674 [03/May/2021:12:46:31.909] load_balancer server/master1 1/20/164 30198 -- 1/1/0/0/0 0/0
May  3 12:46:33 node3 haproxy[655]: 172.16.5.30:59686 [03/May/2021:12:46:33.200] load_balancer server/master2 1/1/71 30196 -- 1/1/0/0/0 0/0
May  3 12:46:34 node3 haproxy[655]: 172.16.5.30:59694 [03/May/2021:12:46:34.522] load_balancer server/master3 1/1/122 30197 -- 1/1/0/0/0 0/0
```

Figure 5.6: HAProxy log shows that requests are distributed to backend server (3 nodes).

Nodes

As Figure 5.4 shows, VIP is assigned to all of the three nodes. We put priority 1 to Node 1, 2 to Node 2, and 3 to Node 3. Since Node 3 has the highest number, it takes VIP the first, and Node 2 can take it when it fails. As the top part of Figure 5.4 shows, each node has to have etcd since there is no external DB.

In the implementation, Node 1's IP address was 172.16.5.10, Node 2's one was 172.16.5.11, and Node 3's one was 172.16.5.12.

Load Balancer (MetalLB)

The configuration for MetalLB in this implementation is also same as section 5.4.3. We assigned 172.16.5.31-172.16.5.39 to be used as the external IP of a LoadBalancer type Service. Pods in any node can be accessed through the load balancer.

5.5 Experiments and Evaluation

5.5.1 High Availability

Test Method

For testing high availability, we killed one of the running VMs in an ungraceful way in order to see how each cluster reacts.

For worker nodes, a simple web service premist/inspekt[*] was used as the app running in Pods, which shows the node name and IP address the Pod is running in, and the name and namespace of the Pod. We set the replicas field of the Deployment controller as two, and Pods were manually distributed to each node for the test.

Worker Nodes

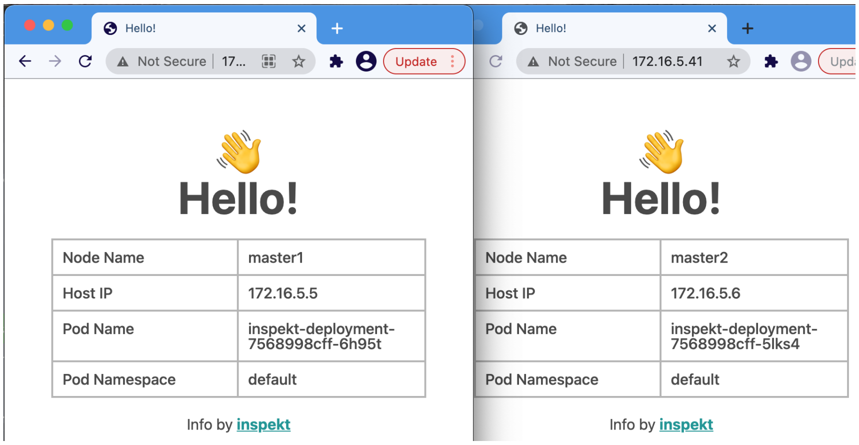
2 Nodes + External DB Cluster

Figure 5.7 shows that all Pods, the members of the LoadBalancer type Service, are accessible through the IP address of the load balancer (172.16.5.41).

We killed the node master2, and Figure 5.8 shows how the cluster reacts. Since the Pod running on the node master2 became dead when its node failed, the Pod was automatically re-created in the node master1. Though it was created soon after the cluster notices that,

---

[*] https://github.com/premist/inspekt/tree/master/deploy



Hello!

Node Name	master1
Host IP	172.16.5.5
Pod Name	inspekt-deployment-7568998cff-6h95t
Pod Namespace	default

Info by inspekt

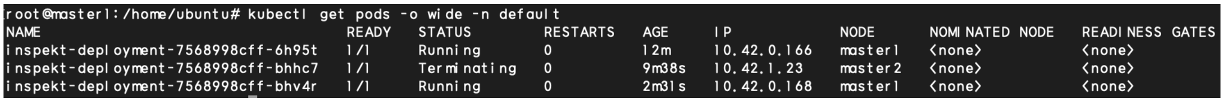
Hello!

Node Name	master2
Host IP	172.16.5.6
Pod Name	inspekt-deployment-7568998cff-5lks4
Pod Namespace	default

Info by inspekt

Figure 5.7: Node 1, 2

it took some time until the recovery. We deal with this limitation below in the content 'Limitations and an Idea for Solution' of this section.

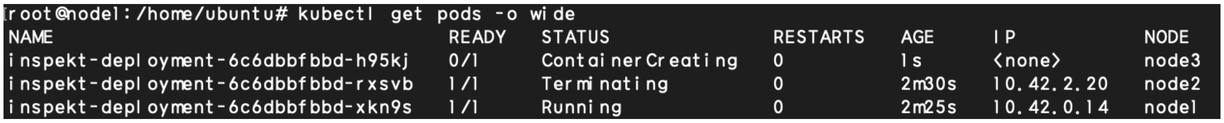


```
root@master1:/home/ubuntu# kubectl get pods -o wide -n default
NAME                                  READY   STATUS        RESTARTS   AGE     IP            NODE      NOMINATED NODE   READINESS GATES
inspekt-deployment-7568998cff-6h95t   1/1     Running       0          12m     10.42.0.166   master1   <none>           <none>
inspekt-deployment-7568998cff-bhhc7   1/1     Terminating   0          9m38s   10.42.1.23    master2   <none>           <none>
inspekt-deployment-7568998cff-bhv4r   1/1     Running       0          2m31s   10.42.0.168   master1   <none>           <none>
```

Figure 5.8: When the node master2 fails, a new Pod is made in the node master1.

3 Nodes Cluster

Basically, there is no difference for testing the high availability of the 3 nodes cluster from testing the 2 nodes + external DB cluster. As Figure 5.9 shows, the failed Pod in the failed node (node2) was re-created automatically in node3. The only difference is that there are two options for creating a new Pod since two nodes are alive.



```
root@node1:/home/ubuntu# kubectl get pods -o wide
NAME                                  READY   STATUS              RESTARTS   AGE     IP           NODE
inspekt-deployment-6c6dbbfbbd-h95kj   0/1     ContainerCreating   0          1s      <none>       node3
inspekt-deployment-6c6dbbfbbd-rxsvb   1/1     Terminating         0          2m30s   10.42.2.20   node2
inspekt-deployment-6c6dbbfbbd-xkn9s   1/1     Running             0          2m25s   10.42.0.14   node1
```

Figure 5.9: When the node node2 fails, a new Pod is made in the node node3.

Limitations and an Idea for Solution

- Delayed Pod Re-creation

  As mentioned in section 5.4.2, the taints and tolerations were set ideally that the Pod in a dead node can be re-created immediately. However, it took a few seconds to create a new Pod after the status of the failed node became "Not Ready." From 100 times tests, it took 4.8 seconds on average until the cluster creates a new Pod is another alive node.

- Node Health Check Interval

  There was another factor that delays a failed Pod to be re-created. It was the interval of the health check of nodes. If a node fails right after a health check, the cluster still regards the failed node as alive until the next health check. The frequency of health checks may depend on the settings of the cluster, and the maximum additional can be the interval between health checks.

- Possible Solution

  A possible solution to avoid problems due to the downtime of some Pods is to ensure that each node has at least one Pod. It can be fulfilled by using a DaemonSet controller. As explained in "Controller" part of section 4.3, DaemonSet generates a Pod in each node. By making each node have at least one running Pod, it is possible to provide a Service without downtime even if one node fails.

  However, since DaemonSet cannot create multiple Pods in one node, another controller like Deployment should be combined in order to generate additional Pods when it has to be scaled up/out.

Master Nodes

2 Nodes + External DB Cluster

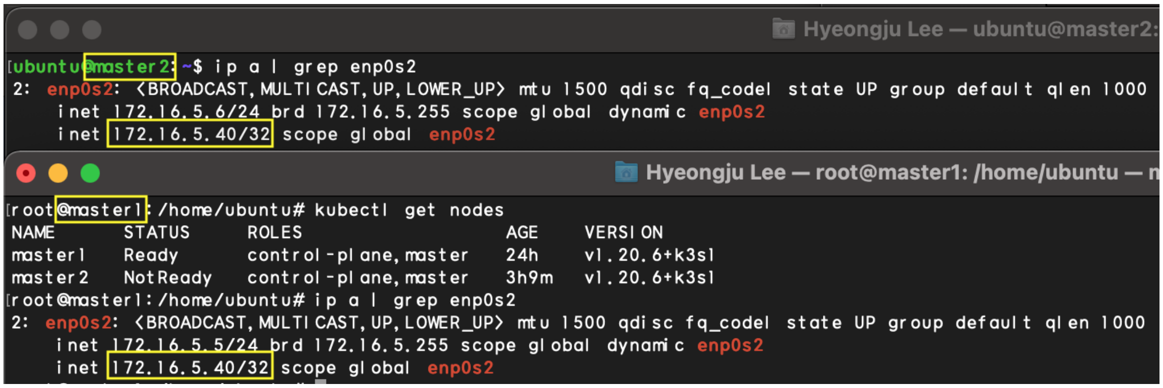
Figure 5.10 shows that the master2 node took VIP and used to work as a load balancer at first, and the master1 node took the VIP after the master2 node failed (status Not Ready). Since all traffic passes through the master1 node after taking the VIP, HAProxy of master1 started working as a load balancer and distributed the requests to master nodes of each node.

- Limitation

  All of the tests with the 2 nodes + external DB cluster were done on the premise that external DB will not fail. In reality, it can also fail.

  Already running Pods in workers were not actually affected by the failure of external DB. Even they could restart automatically and continue running the app when they failed due to the overload. The users of the application Pods may not even notice any failure.

  However, the status of Pods cannot be changed while the external DB is off because master nodes need to communicate with it to create a new Pod or delete a Pod.



Hyeongju Lee — ubuntu@master2:

```
[ubuntu@master2:~$ ip a l grep enp0s2
 2: enp0s2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
     inet 172.16.5.6/24 brd 172.16.5.255 scope global dynamic enp0s2
     inet 172.16.5.40/32 scope global enp0s2
```

Hyeongju Lee — root@master1: /home/ubuntu —

```
[root@master1:/home/ubuntu# kubectl get nodes
 NAME      STATUS    ROLES                  AGE    VERSION
 master1   Ready     control-plane,master   24h    v1.20.6+k3s1
 master2   NotReady  control-plane,master   3h9m   v1.20.6+k3s1
[root@master1:/home/ubuntu# ip a l grep enp0s2
 2: enp0s2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
     inet 172.16.5.5/24 brd 172.16.5.255 scope global dynamic enp0s2
     inet 172.16.5.40/32 scope global enp0s2
```

Figure 5.10: When the master2 node fails, the master1 node takes the VIP and continues providing a load balancer for master nodes.

When the external DB failed, the API server of master nodes could not do anything. It was possible to see that the k3s server services in each node stopped working, and all of the listening ports were closed right after the DB is down. It means that master nodes do not work without the external DB.

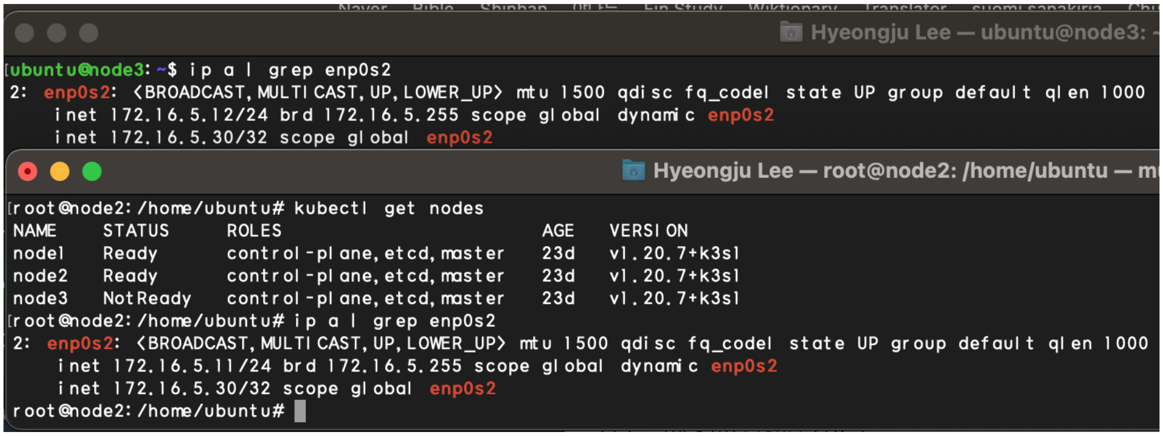
Therefore, the matter of availability of external DB can affect the availability of master nodes.

- Possible Solution

  It is possible to make external DB high available. PostgreSQL supports high availability with the similar approach of making master nodes high available. There have to be multiple DBs (3, 5, ...) and a load balancer that distributes traffic. By making DBs to be synchronized, it is possible to have a high available external DB cluster (PostgreSQL, 2021). With the high available DB, the Kubernetes cluster using external DB can be more highly available.

3 Nodes Cluster

As Figure 5.11 shows, the node3 node was the leader node at first among three nodes since it had the highest priority. After the node3 node fails, the node2 node that has a higher priority than the node1 node takes the VIP. And then, HAProxy of the node2 node started working as a load balancer. The 3 node cluster does not need to care about the high availability of databases as the 2 nodes + external DB cluster because etcd databases are embedded in each node. Master nodes are available as long as two nodes are alive regardless of network connection status, whereas the master nodes of the 2 nodes + external DB cluster cannot work if the network connection to the external DB is lost.


Hyeongju Lee — ubuntu@node3:
Hyeongju Lee — root@node2: /home/ubuntu —

```
[ubuntu@node3:~$ ip a | grep enp0s2
2: enp0s2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    inet 172.16.5.12/24 brd 172.16.5.255 scope global dynamic enp0s2
    inet 172.16.5.30/32 scope global enp0s2
```

```
[root@node2:/home/ubuntu# kubectl get nodes
NAME    STATUS     ROLES                        AGE   VERSION
node1   Ready      control-plane,etcd,master    23d   v1.20.7+k3s1
node2   Ready      control-plane,etcd,master    23d   v1.20.7+k3s1
node3   NotReady   control-plane,etcd,master    23d   v1.20.7+k3s1
[root@node2:/home/ubuntu# ip a | grep enp0s2
2: enp0s2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    inet 172.16.5.11/24 brd 172.16.5.255 scope global dynamic enp0s2
    inet 172.16.5.30/32 scope global enp0s2
root@node2:/home/ubuntu#
```

Figure 5.11: When the node3 node fails, the node2 node takes the VIP and continues providing a load balancer for master nodes.

5.5.2 Scalability

Test Method

- Autoscaling of Pods

  We used the same app inspekt used to test high availability for scalability test too.

  JMeter* was also used to simultaneously generate many HTTP requests to the IP address of the load balancer for working Pods. Tests were performed with ten steps, increasing the number of threads from 100 to 1000. We tested six times for each step, and the increment to the following step was 100. Therefore, the number of requests was 33000 in total. We set the ramp-up period as 1 second, which means that the number of threads becomes the set number in 1 second.

  In the spec of the inspekt Deployment, CPU request 0.5% and CPU limit 2% were assigned for each Pod. After then, we excuted the commands "kubectl autoscale deployment/inspekt-deployment –min=2 –max=10" for the 2 nodes + external DB cluster and "kubectl autoscale deployment/inspekt-deployment –min=3 –max=10" for the 3 nodes cluster. The number of Pods could be scaled automatically up to 10 according to the CPU usage of Pods.

- Scaling the cluster

  We put an additional worker node to the cluster and checked how easily it could join the cluster.

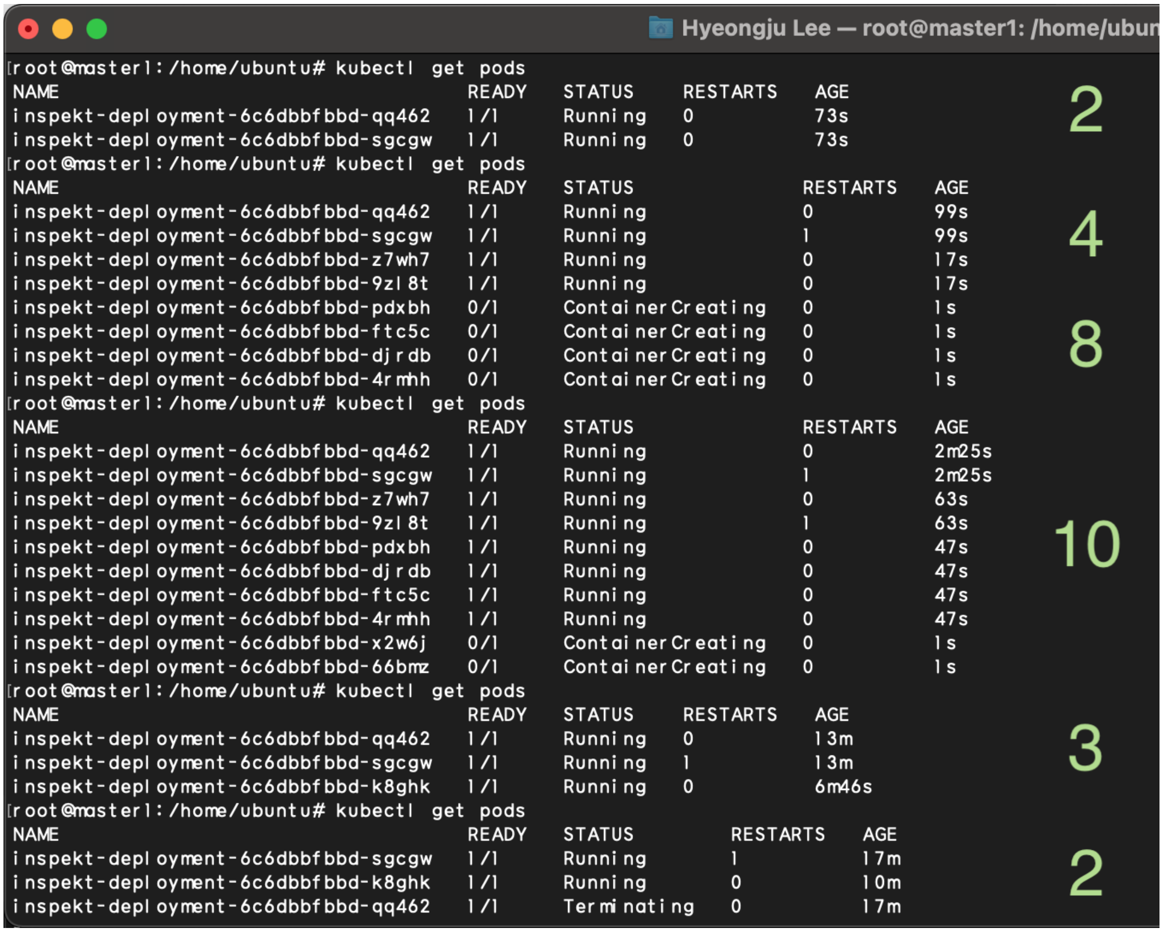
---

*https://jmeter.apache.org

Table 5.1: JMeter Results (Number of Samples, Failure of HTTP Requests, Throughput Each Cluster Process, the Number of Restarts of Pods)

Label	# of Samples	Error %	Throughput	Restarts of Pods
No Scaling	33000	10.5%	56.2 req/sec	12 times
2 Nodes & DB	33000	0.0%	139.1 req/sec	2 times
3 Nodes	33000	0.0%	150.0 req/sec	1 time

Autoscaling of Pods



```
[root@master1:/home/ubuntu# kubectl get pods
 NAME                                  READY   STATUS    RESTARTS   AGE
 inspekt-deployment-6c6dbbfbbd-qq462   1/1     Running   0          73s
 inspekt-deployment-6c6dbbfbbd-sgcgw   1/1     Running   0          73s
[root@master1:/home/ubuntu# kubectl get pods
 NAME                                  READY   STATUS              RESTARTS   AGE
 inspekt-deployment-6c6dbbfbbd-qq462   1/1     Running             0          99s
 inspekt-deployment-6c6dbbfbbd-sgcgw   1/1     Running             1          99s
 inspekt-deployment-6c6dbbfbbd-z7wh7   1/1     Running             0          17s
 inspekt-deployment-6c6dbbfbbd-9zl8t   1/1     Running             0          17s
 inspekt-deployment-6c6dbbfbbd-pdxbh   0/1     ContainerCreating   0          1s
 inspekt-deployment-6c6dbbfbbd-ftc5c   0/1     ContainerCreating   0          1s
 inspekt-deployment-6c6dbbfbbd-djrdb   0/1     ContainerCreating   0          1s
 inspekt-deployment-6c6dbbfbbd-4rmhh   0/1     ContainerCreating   0          1s
[root@master1:/home/ubuntu# kubectl get pods
 NAME                                  READY   STATUS              RESTARTS   AGE
 inspekt-deployment-6c6dbbfbbd-qq462   1/1     Running             0          2m25s
 inspekt-deployment-6c6dbbfbbd-sgcgw   1/1     Running             1          2m25s
 inspekt-deployment-6c6dbbfbbd-z7wh7   1/1     Running             0          63s
 inspekt-deployment-6c6dbbfbbd-9zl8t   1/1     Running             1          63s
 inspekt-deployment-6c6dbbfbbd-pdxbh   1/1     Running             0          47s
 inspekt-deployment-6c6dbbfbbd-djrdb   1/1     Running             0          47s
 inspekt-deployment-6c6dbbfbbd-ftc5c   1/1     Running             0          47s
 inspekt-deployment-6c6dbbfbbd-4rmhh   1/1     Running             0          47s
 inspekt-deployment-6c6dbbfbbd-x2w6j   0/1     ContainerCreating   0          1s
 inspekt-deployment-6c6dbbfbbd-66bmz   0/1     ContainerCreating   0          1s
[root@master1:/home/ubuntu# kubectl get pods
 NAME                                  READY   STATUS    RESTARTS   AGE
 inspekt-deployment-6c6dbbfbbd-qq462   1/1     Running   0          13m
 inspekt-deployment-6c6dbbfbbd-sgcgw   1/1     Running   1          13m
 inspekt-deployment-6c6dbbfbbd-k8ghk   1/1     Running   0          6m46s
[root@master1:/home/ubuntu# kubectl get pods
 NAME                                  READY   STATUS        RESTARTS   AGE
 inspekt-deployment-6c6dbbfbbd-sgcgw   1/1     Running       1          17m
 inspekt-deployment-6c6dbbfbbd-k8ghk   1/1     Running       0          10m
 inspekt-deployment-6c6dbbfbbd-qq462   1/1     Terminating   0          17m
```

Figure 5.12: When the number of HTTP requests increased, the Pods are scaled out automatically. After requests are finished, the number of Pods was scaled back to 2.

When a cluster does not have any configuration for scaling, the result was as 'No Scaling' shows in Table 5.1. Only two Pods had to process all of the 33000 requests. They restarted in total 12 times because of the CPU overload as Table 5.1 shows. It made 10.5% of the requests get failed. The throughput of requests the whole cluster could process was 56.2

per second.

Figure 5.12 is the result from the 2 node + external DB cluster. As the figure shows, the number of Pods increased up to 10 as the number of requests and CPU usage) grow. The number of Pods decreased automatically after finishing the test. The 3 nodes cluster also showed the same result.

As Table 5.1 shows, both clusters (the 2 nodes + external DB cluster and the 3 Nodes cluster) had no failure of response to requests. There were two times of restart of Pods in the 2 nodes + external DB cluster and one time in the 3 nodes cluster, but it did not make the failure of responses. All of the restarts happened before the number of Pods grew to 10, and restart did not happen after the Pods had enough resources to process the threads. The 2 nodes + external DB cluster could process 139.1 requests per second on average, and the 3 nodes cluster could process 150.0 requests per second which were a little bit higher. The reason is that the 3 nodes cluster had 3 Pods from the beginning.

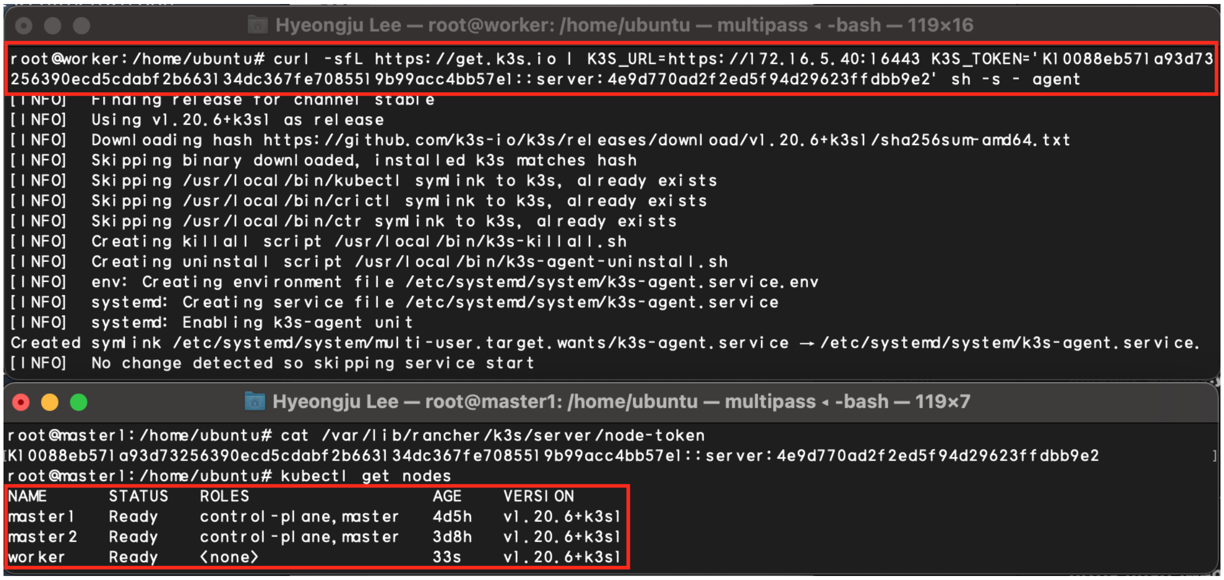
Scaling Out/Up

When some resources of the cluster are running out, more resources have to be put to the cluster. It can be accomplished by putting a new worker node or changing the existing node to a better one. Since the assigned VIP works as an entry point of master nodes' API servers, new worker nodes can join the cluster through the VIP. The command needed for joining the cluster is only one, "curl -sfL https://get.k3s.io | K3S_URL=https://<VIP>:<Port HAProxy uses> K3S_TOKEN='<TOKEN VALUE>' sh -s - agent". Figure 5.13 shows the case that a new worker node was added to the cluster of the 2 nodes + external DB cluster with the command.

In a similar way, it is possible to scale up the resources. After adding a new node to the cluster, the command "kubectl delete node <NODE NAME>" can peacefully remove the node and move running Pods to other nodes.

It shows that there are no dependencies when adding or deleting resources. These characteristics of Kubernetes make clusters easily scalable.

Scaling up by changing the existing node to a better one can also be fulfilled easily. After registering the new node, it is possible to delete the existing node using the command "kubectl delete node <NODE NAME>". Then the Pods running on the deleted node will automatically move to the other proper nodes according to the configuration of the cluster.


Hyeongju Lee — root@worker: /home/ubuntu — multipass ◂ -bash — 119×16
Hyeongju Lee — root@master1: /home/ubuntu — multipass ◂ -bash — 119×7

```
root@worker:/home/ubuntu# curl -sfL https://get.k3s.io | K3S_URL=https://172.16.5.40:16443 K3S_TOKEN='K10088eb571a93d73
256390ecd5cdabf2b663134dc367fe7085519b99acc4bb57e1::server:4e9d770ad2f2ed5f94d29623ffdbb9e2' sh -s - agent
[INFO]  Finding release for channel stable
[INFO]  Using v1.20.6+k3s1 as release
[INFO]  Downloading hash https://github.com/k3s-io/k3s/releases/download/v1.20.6+k3s1/sha256sum-amd64.txt
[INFO]  Skipping binary downloaded, installed k3s matches hash
[INFO]  Skipping /usr/local/bin/kubectl symlink to k3s, already exists
[INFO]  Skipping /usr/local/bin/crictl symlink to k3s, already exists
[INFO]  Skipping /usr/local/bin/ctr symlink to k3s, already exists
[INFO]  Creating killall script /usr/local/bin/k3s-killall.sh
[INFO]  Creating uninstall script /usr/local/bin/k3s-agent-uninstall.sh
[INFO]  env: Creating environment file /etc/systemd/system/k3s-agent.service.env
[INFO]  systemd: Creating service file /etc/systemd/system/k3s-agent.service
[INFO]  systemd: Enabling k3s-agent unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s-agent.service → /etc/systemd/system/k3s-agent.service.
[INFO]  No change detected so skipping service start
```

```
root@master1:/home/ubuntu# cat /var/lib/rancher/k3s/server/node-token
K10088eb571a93d73256390ecd5cdabf2b663134dc367fe7085519b99acc4bb57e1::server:4e9d770ad2f2ed5f94d29623ffdbb9e2
root@master1:/home/ubuntu# kubectl get nodes
NAME      STATUS   ROLES                  AGE    VERSION
master1   Ready    control-plane,master   4d5h   v1.20.6+k3s1
master2   Ready    control-plane,master   3d8h   v1.20.6+k3s1
worker    Ready    <none>                 33s    v1.20.6+k3s1
```

Figure 5.13: Addition of K3s Worker Node

5.6 Resource Utilization

Idle status

We checked how much CPU, memory, and Disk are used right after each K3s cluster is ready to provide Services. We checked when each cluster was idle.

Table 5.2 is the tabulated result for each cluster. "Embedded Etcd" means the 3 nodes cluster, and "External DB" means the 2 nodes + external DB cluster. In addition to the information on each cluster, we checked the resource usage of the node that has no Kubernetes component and wrote in the "Empty Node" part of the table. The "Empty Node" has the initial status after installing Linux OS. Lastly, we checked for the worker node that has nothing installed but only the components for a worker node and wrote in the "Additional Worker" part in the table.

As Table 5.2 shows, both clusters use very small amount of resources. CPU usage per node was less than 100 millicores, and RAM usage was about 5-600MB. However, the total amount of CPU usage was almost double in the 3 nodes cluster than the 2 nodes + external DB cluster, and the usage per node was also higher (96m vs. 74m). In addition, the usage of RAM and disk was also slightly higher in the 3 nodes cluster than the 2 nodes + external DB cluster. It is because the embedded etcd database uses resources.

Lastly, an additional worker node requires about 400MB more RAM and 300MB disk more.

Table 5.2: Resource usage of the clusters, an empty node (that has no Kubernetes components), and an additional worker when they are idle.

	CPU (m = millicores)	**RAM**	**Disk**
Embedded Etcd			
Node 1	287.6m / 3 CPUs, (min 231m, max 344m)	70-74%	2.0G
Node 2		73-75%	1.8G
Node 3		71-74%	1.8G
External DB			
Node 1	148m / 2 CPUs, (min 126m, max 167m)	71-73%	1.9G
Node 2		60-63%	1.8G
Empty Node	less than 5m	11-13%	1.25G
Additional Worker	16m	50%	1.56G

When creating Pods

While taking tests, we discovered that the 3 nodes cluster uses more CPU than the 2 nodes + external DB cluster when the same commands are given to the master nodes.

The left plot in Figure 5.14 shows the CPU usage of each cluster when executing a command that creates 10 Pods in its cluster. The 2 nodes + external DB cluster's CPU usage reached at most about 800 millicores, and the 3 nodes cluster's usage was about 1190 millicores.

The right plot in Figure 5.14 shows the CPU usage of each cluster when executing a command that creates 20 Pods in its cluster. The 2 nodes + external DB cluster's CPU usage reached at most about 1600 millicores, and the 3 nodes cluster's usage was about 2350 millicores. Though the 2 nodes + external DB cluster uses a little bit more CPU from 9-second to 13-second period, the CPU usage of the 3 nodes cluster is much higher in general.

5.7 Discussion

In section 5.5 and section 5.6, we showed that both Kubernetes clusters can provide high availability and scalability with limited resources in edge environments. However, there were different characteristics and limitations of each cluster.

First, let us summarize the traits of each cluster based on the results provided in previous

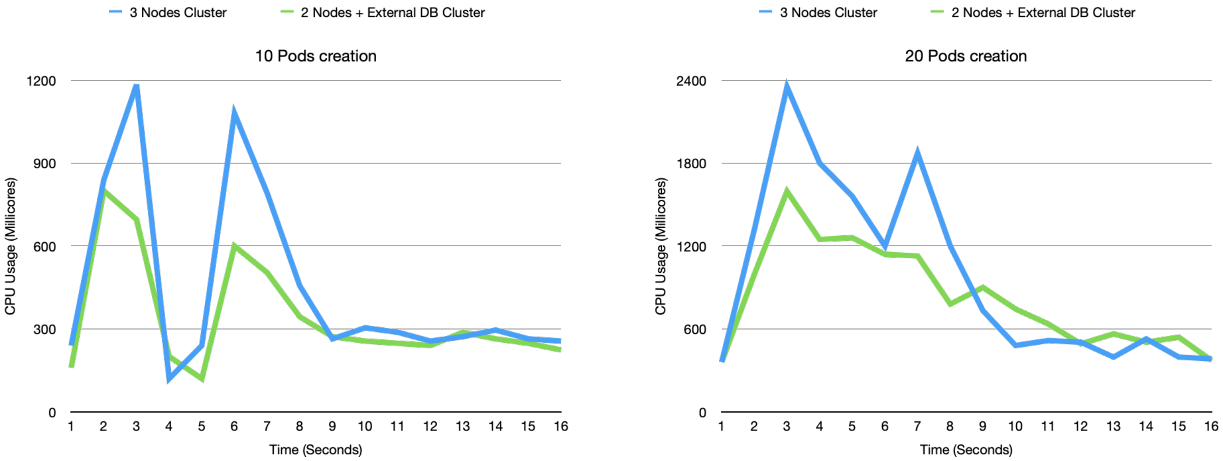


Figure 5.14: CPU usage when creating Pods (10 Pods for the left plot and 20 Pods for the right plot) in each cluster.

sections.

The 2 nodes + external DB cluster used less resources for executing commands to change the status of the cluster. However, it has to make the external DB also high available for master nodes to work properly. In addition, master nodes cannot work if the connection between the external DB and nodes is lost.

On the other hand, the 3 nodes cluster does not need any additional considerations for the high availability of databases as long as two nodes are alive. However, its resource usage was higher than the 2 nodes + external DB cluster when changing the status of the cluster.

Based on the characteristics of clusters, we discuss which cluster will be more appropriate for specific IoT applications in this section.

Connected Vehicles

In section 2.2.1, we showed that the amount of data to be processed would fluctuate according to how many functions are used. It means that the amount of resources assigned for providing functions has to be scaled often. Consider a situation that a driver starts driving and enters the highway in five minutes. Cameras and sensors will start generating data to be processed and, the driving assistant function starts working soon after entering the highway. In such a case, the number of Pods has to grow continuously to process the surging data. Since the 3 nodes cluster uses more resources to do the works, the 2 nodes + external DB cluster will be appropriate in general.

Suppose that one more thing has to be considered in addition to the scenario above. There are many long tunnels on the way, and the internet connection may be poor. Then there can be benefits and drawbacks of each cluster. When choosing the 2 nodes + external DB cluster, we may need to give up the quality of functions since more resources cannot be assigned due to the malfunction of master nodes. On the other hand, the 3 nodes cluster can guarantee that master nodes will work even if there is no internet connection. However, resources will be used more whenever the amount of assigned resources has to be scaled. If the edge computers have enough resources or the functions are not heavy that much, 3 nodes cluster can be a better option. Furthermore, on the premise that there is enough physical space to put more resources, it is also good to consider putting an additional worker node to 3 worker nodes so that it can have both beneficial aspects.

Smart Factory

In section 2.2.2, we showed that smart factory generates a huge amount of data. However, the amount of data will not fluctuate as much as the case of the connected vehicle because all machines will generate data regularly. The scalability smart factory requires is more like the matter of scaling the whole amount of resources for the application.

In this case, 2 nodes + external DB cluster is a more appropriate choice than 3 nodes cluster. In case a smart factory has other computing resources that have more capacity than edges in the same network, it is possible to use it as the external DB. Then, it is possible to minimize the problem caused by the unavailability due to the lost connection between nodes and DB. In addition, it is possible to use the resources in the edges more efficiently since the Kubernetes cluster has only master nodes and worker nodes in edge devices.

6 Conclusion

IoT has grown fast, and many IoT applications such as connected vehicles and IIoT need high availability and scalability. In addition, the applications require low latency, security, etc. Edge computing technology can provide good solutions, but it is important to use the resources efficiently.

For this reason, in this thesis, we surveyed the two architectures of high available Kubernetes cluster that can fulfill the requirements. One of the architectures is the cluster using two nodes that contain both master and worker nodes with an external database. Another one is the cluster using three nodes that include master, worker, and embedded etcd. And then, we implemented them using K3s Kubernetes distribution, HAProxy with keepalived, MetalLB, etc.

Through the experiments in section 5.5, we showed that K3s clusters are useful for deploying applications in an edge environment for many IoT applications that need high availability and scalability. Though we could find some limitations of the implementations, K3s clusters in edge environments can provide powerful functions with a small amount of resources.

In addition, in section 5.6, we showed the usage of resources such as CPU, RAM, and disk. We can see that both clusters use a small amount of resources through the results, but 2 nodes + external DB cluster use less resources in general than 3 nodes cluster.

Based on the difference of characteristics of the two clusters, we discussed in section 5.7 which cluster will be more appropriate in some given applications. There are many considerations when deploying applications in edge devices. The 3 nodes cluster was suitable for the applications that require high availability even when the internet connection is lost, such as connected vehicles that have to work even in tunnels. On the other hand, the 2 nodes + external DB cluster was suitable for the applications that generate a huge amount of data in a short time and require resources to be used as efficiently as possible such as IIoT.

Bibliography

Aazam, M., Zeadally, S., and Harras, K. A. (2018a). “Deploying Fog Computing in Industrial Internet of Things and Industry 4.0”. In: *IEEE Transactions on Industrial Informatics* 14.10, pp. 4674–4682. DOI: 10.1109/TII.2018.2855198.

– (2018b). “Deploying Fog Computing in Industrial Internet of Things and Industry 4.0”. In: *IEEE Transactions on Industrial Informatics* 14.10, pp. 4674–4682. DOI: 10.1109/TII.2018.2855198.

Ashton, K. et al. (2009). “That ‘internet of things’ thing”. In: *RFID journal* 22.7, pp. 97–114.

Barrachina, J., Garrido, P., Fogue, M., Martinez, F., Cano, J.-C., Calafate, C., and Manzoni, P. (July 2015). “A V2I-based Real-Time Traffic Density Estimation System in Urban Scenarios”. In: *Wireless Personal Communications*. DOI: 10.1007/s11277-015-2392-4.

Bartje, J. (2016). *The top 10 IoT application areas – based on real IoT projects*. URL: https://iot-analytics.com/top-10-iot-project-application-areas-q3-2016/ (visited on 05/26/2021).

Basir, R., Qaisar, S., Ali, M., Aldwairi, M., Ashraf, M., Mahmood, A., and Gidlund, M. (Nov. 2019). “Fog Computing Enabling Industrial Internet of Things: State-of-the-Art and Research Challenges”. In: *Sensors* 19, p. 4807. DOI: 10.3390/s19214807.

Böhm, S. and Wirtz, G. (Mar. 2021). “Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes”. In:

businesswire.com (2021). *Growth Opportunities of IIoT in Small and Medium Scale Enterprises, 2020 Report - Data Communications and Connectivity Between Devices Encouraging Adoption of IIoT - ResearchAndMarkets.com*. URL: https://www.businesswire.com/news/home/20210216005990/en/Growth-Opportunities-of-IIoT-in-Small-and-Medium-Scale-Enterprises-2020-Report---Data-Communications-and-Connectivity-Between-Devices-Encouraging-Adoption-of-IIoT---ResearchAndMarkets.com (visited on 05/26/2021).

Busser, A. (2020). *From Docker to OCI: What is a container?* URL: https://www.padok.fr/en/blog/container-docker-oci (visited on 05/26/2021).

CAAT (2021). *Connected and Automated Vehicles*. URL: http://autocaat.org/Technologies/Automated_and_Connected_Vehicles/ (visited on 05/26/2021).

Caiza, G., Saeteros, M., Oñate, W., and Garcia, M. V. (2020). "Fog computing at industrial level, architecture, latency, energy, and security: A review". In: *Heliyon* 6.4, e03706. ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2020.e03706. URL: https://www.sciencedirect.com/science/article/pii/S240584402030551X.

Canonical (2021a). *Introduction to MicroK8s*. URL: https://microk8s.io/docs (visited on 05/28/2021).

– (2021b). *High availability (HA)*. URL: https://microk8s.io/high-availability (visited on 05/28/2021).

Caprolu, M., Di Pietro, R., Lombardi, F., and Raponi, S. (2019). "Edge Computing Perspectives: Architectures, Technologies, and Open Security Issues". In: *2019 IEEE International Conference on Edge Computing (EDGE)*, pp. 116–123. DOI: 10.1109/EDGE.2019.00035.

Choudhury, T., Gupta, A., Pradhan, S., Kumar, P., and Rathore, Y. S. (2017). "Privacy and Security of Cloud-Based Internet of Things (IoT)". In: *2017 3rd International Conference on Computational Intelligence and Networks (CINE)*, pp. 40–45. DOI: 10.1109/CINE.2017.28.

Cloud, G. (2021). *Containerd images*. URL: https://cloud.google.com/kubernetes-engine/docs/concepts/using-containerd (visited on 05/26/2021).

CNCF (2015). *New Cloud Native Computing Foundation to drive alignment among container technologies*. URL: https://www.cncf.io/announcements/2015/06/21/new-cloud-native-computing-foundation-to-drive-alignment-among-container-technologies/ (visited on 05/27/2021).

CONFIGURATION (2021). *INSTALLATION*. URL: https://metallb.universe.tf/configuration/ (visited on 05/31/2021).

cri-o (2021). *LIGHTWEIGHT CONTAINER RUNTIME FOR KUBERNETES*. URL: https://cri-o.io (visited on 05/28/2021).

Datadog (2018). *8 surprising facts about real docker adoption*. URL: https://www.datadoghq.com/docker-adoption/ (visited on 05/26/2021).

Docker (2021). *Docker frequently asked questions (FAQ)*. URL: https://docs.Docker.com/engine/faq/ (visited on 05/26/2021).

Engels, D., Sarma, S., Putta, L., and Brock, D. (Jan. 2002). "The Networked Physical World System." In: pp. 104–111.

Ferner, P. (2021). *Cloud Computing*. URL: https://scitis.io/index.php/flexibility-in-industry-4-0-why-scalability-is-so-important/?lang=en (visited on 05/26/2021).

Frankenfield, J. (2020). *Cloud Computing.* URL: https://www.investopedia.com/terms/c/cloud-computing.asp (visited on 05/26/2021).

G., N. (2021). *How Many IoT Devices Are There in 2021? [All You Need To Know].* URL: https://techjury.net/blog/how-many-iot-devices-are-there/#gref (visited on 05/26/2021).

Ghosh, A., Khalid, O., Rais, R. N. B., Rehman, A., Malik, S., and Khan, I. (Mar. 2019). “Data offloading in IoT environments: modeling, analysis, and verification”. In: *EURASIP Journal on Wireless Communications and Networking* 2019. DOI: 10.1186/s13638-019-1358-8.

Giang, N. K., Lea, R., Blackstock, M., and Leung, V. C. (2018). “Fog at the Edge: Experiences Building an Edge Computing Platform”. In: *2018 IEEE International Conference on Edge Computing (EDGE)*, pp. 9–16. DOI: 10.1109/EDGE.2018.00009.

Gillis, A. S. (2020). *internet of things (IoT).* URL: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT (visited on 05/26/2021).

Google (2021a). *Exposing applications using services.* URL: https://cloud.google.com/kubernetes-engine/docs/how-to/exposing-apps#using-gcloud-config (visited on 05/28/2021).

– (2021b). *Services.* URL: https://cloud.google.com/kubernetes-engine/docs/concepts/service (visited on 05/28/2021).

– (2021c). *CONTAINERS AT GOOGLE.* URL: https://cloud.google.com/containers (visited on 05/26/2021).

– (2021d). *What is Kubernetes?* URL: https://cloud.google.com/learn/what-is-kubernetes (visited on 05/27/2021).

H, J. (2020). *What Are Containerized Microservices?* URL: https://blog.dreamfactory.com/what-are-containerized-microservices/ (visited on 05/26/2021).

Holst, A. (2021). *Number of IoT connected devices worldwide 2019-2030.* URL: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/ (visited on 05/26/2021).

IBM (2018). “IBM Storage solutions for advanced driver assistance systems and autonomous driving”. In:

Ismail, B. I., Mostajeran Goortani, E., Ab Karim, M. B., Ming Tat, W., Setapa, S., Luke, J. Y., and Hong Hoe, O. (2015). “Evaluation of Docker as Edge computing platform”. In: *2015 IEEE Conference on Open Systems (ICOS)*, pp. 130–135. DOI: 10.1109/ICOS.2015.7377291.

Jahoda, P. (2021). *Benchmark databases in Docker: MySQL, PostgreSQL, SQL Server.* URL: https://itnext.io/benchmark-databases-in-docker-mysql-postgresql-sql-server-7b129368eed7 (visited on 05/26/2021).

Jorge Castro Duffie Cooley, e. (2020). *Don't Panic: Kubernetes and Docker.* URL: https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/ (visited on 05/26/2021).

Jorge Palma, K. S. (2020). *Azure Kubernetes Service (AKS) support for containerd runtime is in preview.* URL: https://azure.microsoft.com/en-us/updates/azure-kubernetes-service-aks-support-for-containerd-runtime-is-in-preview/ (visited on 05/26/2021).

Jovanović, B. (2021). *Internet of Things statistics for 2021 – Taking Things Apart.* URL: https://dataprot.net/statistics/iot-statistics/ (visited on 05/26/2021).

KIM, C. (2019). *Building a Highly Available Kubernetes Cluster.* URL: https://rancher.com/learning-paths/building-a-highly-available-kubernetes-cluster/ (visited on 05/26/2021).

Kubernetes (2020a). *Services, Load Balancing, and Networking.* URL: https://v1-18.docs.kubernetes.io/docs/concepts/services-networking/service/ (visited on 05/26/2021).

– (2020b). *Understanding Kubernetes Objects.* URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/ (visited on 05/27/2021).

– (2021a). *Container runtimes.* URL: https://kubernetes.io/docs/setup/production-environment/container-runtimes/ (visited on 05/26/2021).

– (2021b). *CronJob.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/ (visited on 05/31/2021).

– (2021c). *DaemonSet.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/ (visited on 05/31/2021).

– (2021d). *Deployment.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visited on 05/31/2021).

– (2021e). *Ingress.* URL: https://kubernetes.io/docs/concepts/services-networking/ingress/ (visited on 05/27/2021).

– (2021f). *Ingress Controllers.* URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/ (visited on 05/31/2021).

– (2021g). *Jobs.* URL: https://kubernetes.io/docs/concepts/workloads/controllers/job/ (visited on 05/31/2021).

– (2021h). *kube-controller-manager*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/ (visited on 05/27/2021).

– (2021i). *kube-proxy*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/ (visited on 05/28/2021).

– (2021j). *kubelet*. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/ (visited on 05/28/2021).

– (2021k). *Kubernetes Components*. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 05/31/2021).

– (2021l). *Namespaces*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (visited on 05/27/2021).

– (2021m). *Pods*. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 05/27/2021).

– (2021n). *ReplicaSet*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (visited on 05/31/2021).

– (2021o). *ReplicationController*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/ (visited on 05/31/2021).

– (2021p). *ReplicationController*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/ (visited on 05/26/2021).

– (2021q). *Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 05/27/2021).

– (2021r). *StatefulSets*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/ (visited on 05/31/2021).

– (2021s). *Taints and Tolerations*. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/ (visited on 05/28/2021).

– (2021t). *Volumes*. URL: https://kubernetes.io/docs/concepts/storage/volumes/ (visited on 05/27/2021).

Lab, D. K. (2021). *Linux Network Namespace Introduction*. URL: https://docker-k8s-lab.readthedocs.io/en/latest/docker/netns.html (visited on 05/26/2021).

Lai, A. (2020). *KubeEdge and Its Role in Multi-Access Edge Computing*. URL: https://thenewstack.io/kubeedge-and-its-role-in-multi-access-edge-computing (visited on 05/26/2021).

Lerner, A. (2014). *The Cost of Downtime*. URL: https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/ (visited on 05/26/2021).

Liberg, O., Sundberg, M., Wang, Y.-P. E., Bergman, J., Sachs, J., and Wikström, G. (2020). “Chapter 17 - Technical enablers for the IoT”. In: *Cellular Internet of Things*

(Second Edition). Ed. by O. Liberg, M. Sundberg, Y.-P. E. Wang, J. Bergman, J. Sachs, and G. Wikström. Second Edition. Academic Press, pp. 709–730. ISBN: 978-0-08-102902-2. DOI: https://doi.org/10.1016/B978-0-08-102902-2.00017-0. URL: https://www.sciencedirect.com/science/article/pii/B9780081029022000170.

Logic, S. (2019). *Designing a Microservices Architecture with Docker Containers*. URL: https://www.sumologic.com/insight/microservices-architecture-docker-containers/ (visited on 05/26/2021).

Manaouil, K. and Lebre, A. (Oct. 2020). *Kubernetes and the Edge?* Research Report RR-9370. Inria Rennes - Bretagne Atlantique, p. 19. URL: https://hal.inria.fr/hal-02972686.

Masters, K. (2017). *The Impact of Industry 4.0 on the Automotive Industry*. URL: https://blog.flexis.com/the-impact-of-industry-4.0-on-the-automotive-industry (visited on 05/26/2021).

Mell, P. and Grance, T. (2011-09-28 2011). *The NIST Definition of Cloud Computing*. en. DOI: https://doi.org/10.6028/NIST.SP.800-145.

MetalLB (2021). *INSTALLATION*. URL: https://metallb.universe.tf/installation/ (visited on 05/31/2021).

Morabito, R. (2017). “Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation”. In: *IEEE Access* 5, pp. 8835–8850. DOI: 10.1109/ACCESS.2017.2704444.

Naser, H. (2017). *kubernetes vs. docker swarm: a comparison of containerization platforms*. URL: https://vexxhost.com/blog/kubernetes-vs-docker-swarm-containerization-platforms/ (visited on 05/26/2021).

OCI (2021). *About the Open Container Initiative*. URL: https://opencontainers.org/about/overview/ (visited on 05/27/2021).

OpenShift, R. H. (2021). *Using the CRI-O Container Engine*. URL: https://docs.openshift.com/container-platform/3.11/crio/crio_runtime.html (visited on 05/26/2021).

Pahl, C., Helmer, S., Miori, L., Sanin, J., and Lee, B. (2016). “A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters”. In: *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pp. 117–124. DOI: 10.1109/W-FiCloud.2016.36.

Posey, B. (2021). *industrial internet of things (IIoT)*. URL: https://internetofthingsagenda.techtarget.com/definition/Industrial-Internet-of-Things-IIoT (visited on 05/26/2021).

PostgreSQL (2021). *Chapter 25. High Availability, Load Balancing, and Replication.* URL: https://www.postgresql.org/docs/9.5/high-availability.html (visited on 05/30/2021).

Pourqasem, J. (Dec. 2018). "Cloud-Based IoT: Integration Cloud Computing with Internet of Things". In: DOI: 10.22105/riej.2018.88380.

Qureshi, B., Kawlaq, K., Koubaa, A., Saeed, B., and Younis, M. (2019). "A Commodity SBC-Edge Cluster for Smart Cities". In: *2019 2nd International Conference on Computer Applications Information Security (ICCAIS)*, pp. 1–6. DOI: 10.1109/CAIS.2019.8769500.

Rancher (2021a). *K3s Resource Profiling.* URL: https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/resource-profiling/ (visited on 05/28/2021).

– (2021b). *K3s Server Configuration Reference.* URL: https://rancher.com/docs/k3s/latest/en/installation/install-options/server-config/ (visited on 05/28/2021).

RedHat (2021). *What is container orchestration?* URL: https://www.redhat.com/en/topics/containers/what-is-container-orchestration (visited on 05/27/2021).

Research, G. V. (2019). *Industrial Internet Of Things Market Size, Share & Trends Analysis Report By Component, By End-use (Manufacturing, Energy & Power, Oil & Gas, Healthcare, Logistics & Transport, Agriculture), And Segment Forecasts, 2019 - 2025.* Grand View Research.

Richard Viereckl Jörg Assmann, C. R. (2014). *Strategy and In the fast lane - The bright future of connected cars.* URL: https://web.archive.org/web/20180513185101/https://www.strategyand.pwc.com/media/file/Strategyand_In-the-Fast-Lane.pdf (visited on 05/26/2021).

Rijmenam, D. M. van (2013). *Self-driving Cars Will Create 2 Petabytes Of Data, What Are The Big Data Opportunities For The Car Industry?* URL: https://datafloq.com/read/self-driving-cars-create-2-petabytes-data-annually/172 (visited on 05/26/2021).

Sarkar, A. (2019). *Docker and OCI Runtimes.* URL: https://medium.com/@avijitsarkar123/docker-and-oci-runtimes-a9c23a5646d6 (visited on 05/26/2021).

Sarkar, S., Chatterjee, S., and Misra, S. (2018). "Assessment of the Suitability of Fog Computing in the Context of Internet of Things". In: *IEEE Transactions on Cloud Computing* 6.1, pp. 46–59. DOI: 10.1109/TCC.2015.2485206.

Sasaki, K., Suzuki, N., Makido, S., and Nakao, A. (2016). “Vehicle control system coordinated between cloud and mobile edge computing”. In: *2016 55th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pp. 1122–1127. DOI: 10.1109/SICE.2016.7749210.

Satyanarayanan, M. (2017). “The Emergence of Edge Computing”. In: *Computer* 50.1, pp. 30–39. DOI: 10.1109/MC.2017.9.

Scully, P. (2018). *New Research on 1,600 Enterprise IoT Projects: Upsurge in Smart City and Connected Building Related IoT Projects*. URL: https://iot-analytics.com/global-overview-1600-enterprise-iot-projects/ (visited on 05/26/2021).

– (2020). *Top 10 IoT applications in 2020*. URL: https://iot-analytics.com/top-10-iot-applications-in-2020/ (visited on 05/26/2021).

Shi, C., Ren, Z., Yang, K., Chen, C., Zhang, H., Xiao, Y., and Hou, X. (2018). “Ultra-low latency cloud-fog computing for industrial Internet of Things”. In: *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–6. DOI: 10.1109/WCNC.2018.8377192.

Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. (2016). “Edge Computing: Vision and Challenges”. In: *IEEE Internet of Things Journal* 3.5, pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

Shu, C., Zhao, Z., Han, Y., Min, G., and Duan, H. (2020). “Multi-User Offloading for Edge Computing Networks: A Dependency-Aware and Latency-Optimal Approach”. In: *IEEE Internet of Things Journal* 7.3, pp. 1678–1689. DOI: 10.1109/JIOT.2019.2943373.

Singh, S. (2017). “Optimize cloud computations using edge computing”. In: *2017 International Conference on Big Data, IoT and Data Science (BID)*, pp. 49–53. DOI: 10.1109/BID.2017.8336572.

StackRox (2021). *Kubernetes and Container Security and Adoption Trends*. URL: https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers/ (visited on 05/27/2021).

Varghese, B., Wang, N., Barbhuiya, S., Kilpatrick, P., and Nikolopoulos, D. S. (Nov. 2016). “Challenges and Opportunities in Edge Computing”. In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18.

VTS (2021). *Connected Vehicles*. URL: https://site.ieee.org/connected-vehicles/ieee-connected-vechicles/connected-vehicles/ (visited on 05/26/2021).

Vu, T., Mediran, C. J., and Peng, Y. (2019). “Measurement and Observation of Cross-Provider Cross-Region Latency for Cloud-Based IoT Systems”. In: *2019 IEEE World*

Congress on Services (SERVICES). Vol. 2642-939X, pp. 364–365. DOI: 10.1109/SERVICES.2019.00105.

Xiong, Y., Sun, Y., Xing, L., and Huang, Y. (2018). "Extend Cloud to Edge with KubeEdge". In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 373–377. DOI: 10.1109/SEC.2018.00048.

Xu, J., Palanisamy, B., Ludwig, H., and Wang, Q. (2017). "Zenith: Utility-Aware Resource Allocation for Edge Computing". In: *2017 IEEE International Conference on Edge Computing (EDGE)*, pp. 47–54. DOI: 10.1109/IEEE.EDGE.2017.15.

Yang, H. and Kim., Y. (2019). "Design and Implementation of High-Availability Architecture for IoT-Cloud Services". In: *Sensors (Basel, Switzerland)* 19.15, p. 3276. DOI: https://doi.org/10.3390/s19153276.

Zenlayer (2017). *The Importance Of Scalable and Flexible Bandwidth*. URL: https://www.zenlayer.com/blog/the-importance-of-scalable-and-flexible-bandwidth/ (visited on 05/26/2021).