The Cooper Union
Albert Nerken School of Engineering

Edge Computing on Low Availability Devices with K3S in a Smart Home IoT System

by
Yingzhi Hao

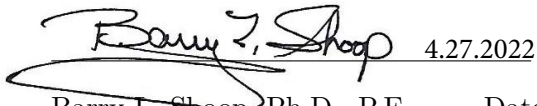
A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Engineering

April 2022

Advisor
Professor Carl Sable

The Cooper Union for the Advancement of Science and Art
Albert Nerken School of Engineering

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.



4.27.2022

Barry L. Shoop, Ph.D., P.E. Date

Dean, Albert Nerken School of Engineering

4.27.2022

Prof. Carl Sable Date

Candidate's Thesis Advisor

Acknowledgements

I would like to give my warmest thanks to my advisor, Professor Carl Sable, for his great guidance on this project. I learned a lot from him both as an undergraduate and a graduate student, and I appreciate all the support that he has given me.

I also want to express my gratitude to my parents, who spent a great effort supporting me, and to my friends, who have always been with me and made my academic life an enjoyable experience.

Abstract

In recent years, edge computing has drawn lots of attention in the development of Internet of Things applications as a way to bring computing resources close to data sources. Containerization technology provides an efficient virtualization solution to run computing services on edge devices. Kubernetes allows containers to be easily orchestrated on edge devices. Most of the existing research was done under the assumption that when an edge device is registered as part of an edge computing system, the device is always running with high availability. In this paper, we present a system that uses K3S, a lightweight Kubernetes implementation, to orchestrate computing services onto low availability edge devices to convert these devices into edge computing resources.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Edge computing has emerged in recent years as a new way of processing data. Traditionally, data generated by end-devices are sent to centralized computing servers hosted on private or public data centers for processing. However, some new technologies, such as the Internet of Things (IoT), have imposed new challenges on this traditional model. An IoT system can have many end-devices generating a significant amount of raw data. Sending all these data to a remote computing server puts lots of pressure on network infrastructures and computing servers. In addition, network latency becomes a hard problem to solve as the network traffic often has to be routed several times and travel a long distance before reaching the remote computing server. Edge computing can help to overcome these challenges by enabling edge devices that are close to the data source to process data.

Containerization is a virtualization technique that has been heavily researched for edge computing solutions. Containers enable edge computing applications to be quickly deployed and run on edge devices by creating an isolated running environment and providing the

dependencies that the application needs. As the scale of edge computing increases, the number of containers also increases. To ease the task of managing a large scale of containers, Kubernetes as a container orchestration tool has drawn lots of attention in recent years. There have been research on using Kubernetes to orchestrate containers on edge devices to provide edge computing solutions [13] [18]. However, all the research we have found assumes that once an edge device has been registered as an edge computing resource, this device should always be available to process computing tasks until the edge device gets removed from the edge computing system. In other words, to a client who needs edge computing resources to process data, the edge device within an edge computing system is expected to be highly available, and it should only be offline when unexpected accidents happen, such as software and hardware failures. Within a residential house, this assumption is valid for some edge devices such as network routers and some smart home appliances that are designed to be always online. However, there are also devices that are not intended to be constantly running. For example, smart TVs, gaming consoles, or even personal computers may only running when they are being used. In this paper, we refer to these devices that are intended to be online only for a period of time as low availability devices. This paper explores the possibility of utilizing a lightweight Kubernetes implementation, K3S, to orchestrate containers on low availability edge devices to use these devices as edge computing resources in a smart home IoT scenario.

In Chapter 2, we introduce relevant background material such as IoT, edge computing, containers, and Kubernetes. We discuss some previous research that has inspired our system design in Chapter 3. Chapter 4 presents our system in detail, and we provide the system evaluation results in Chapter 5. We conclude this paper and propose potential future work in Chapter 6.

Chapter 2

Background

2.1 Internet of Things and smart home

The Internet of Things (IoT) is a network of items that can be made intelligent if they are identified and addressed. These items usually are devices that have sensors to collect data from their environment and actuators to execute actions according to the collected data [31]. The pioneer implementation of an IoT device can be traced back to the early 1980s when a few students and researchers installed sensors on a Coke vending machine to monitor its inventory. The sensor collected data was periodically sent to a computer connecting to the ARPANET, which allowed the vending machine inventory to be monitored remotely [39]. Since then, IoT technologies have evolved, and IoT applications have been developed for different industries. The adoption of IoT systems has been continuously growing in various areas, among which, smart home is one of the most popular.

A residential house can have a network of IoT-capable home appliances. Examples of such

devices range from small devices, such as a lighting bulb or a security camera, to large home appliances, such as refrigerators. These IoT-capable appliances can also be adopted at a large scale to form an IoT system at home. Such a system allows appliances to be digitally connected and work together to provide more automated and enhanced services to house residents. Such an IoT system is usually referred to as a "smart home" system [34].

Different IoT architectures for smart home systems have been proposed and developed throughout the years. In general, a smart home IoT system includes a set of devices at endpoints, a network gateway, a central computing service, and an application interface for user control. The set of devices at the endpoints are usually small, embedded electronics with sensors and actuators that collect raw data from the environment and execute control commands. These devices are commonly referred to as edge devices since they are located at the leaves of the network topology. Edge devices usually have limited hardware resources and computing power. They cannot fully process all the collected data on-premises but need to send data to a computing service for data processing. The network gateway is responsible for receiving data from endpoint devices within the local network and sending these data to the computing service. The computing service processes the received data and sends feedback or control commands back to the edge devices, and it usually stores some data for future analytics or user use. The application interface is a user-friendly service that allows users to monitor and configure the smart home system. It is usually in the form of a front-end web UI or a mobile application.

The computing service can be hosted locally within the same network as the edge devices. Alternatively, it can be hosted remotely on the cloud. Depending on where it is hosted, a smart home IoT system can be categorized as local or cloud-centric.

2.1.1 Local managed smart home IoT system

The computing resource of a smart home IoT system can be placed locally in a residential house. Figure 2.1 shows an example of a local smart home system. In this setup, the computing service is hosted on a local private server. The server exchanges the data with IoT devices through its network gateway and processes the data. Compared with having the computing service hosted on the cloud, since the computing resource is close to the edge devices, the network latency is more negligible, and the computing service is more stable and reliable. In addition, since all data is transmitted, processed, and stored within the local network, data privacy and security are more manageable. However, users do need to spend additional effort on setting up the hardware and software of the computing service.

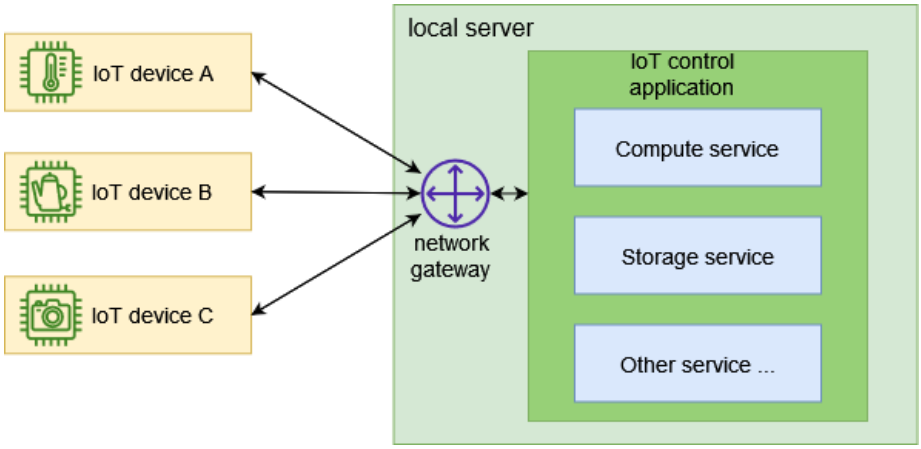


Figure 2.1: Example of an IoT System Local Setup

Among the popular home automation platforms, Home Assistance [4] is one platform that helps to build an entire smart home IoT system within a local network. Home Assistance's core is an application that runs state machines, an event bus, and a service registry. State machines keep track of the state of a variable, such as the brightness of a lighting bulb. A connected IoT device constantly sends state data to the state machine. When the state

is changed, the state machine notifies the event bus. The event bus then calls an event listener or a service program to execute specific action to react to the state change.

Home Assistance's core is usually installed with the Home Assistance Operating System, which is a customized Linux OS optimized for running Home Assistance applications. Another way of running Home Assistance's core is to run it as a Docker container. No matter which installation method to be used, since Home Assistance needs to be always available to keep the automation devices to work, it needs to be hosted on always-running hardware. Hence, it is highly recommended to have dedicated hardware, such as a Raspberry Pi, to host the Home Assistance OS or Docker container [5].

2.1.2 Cloud-centric Smart home IoT system

Cloud-centric IoT-based solutions use the cloud as the computing resource to process edge-collected data (see Figure 2.2). Cloud computing enables smart home users to avoid having an additional dedicated resource at home to provide computing power, which alleviates users from purchasing and maintaining additional hardware. Cloud computing's high scalability enables an IoT system to be expanded without concern of the hardware constraints. In addition, some cloud providers offer a rich set of features, such as AI and ML abilities, in their cloud ecosystem that can be integrated into IoT systems to provide more functionality. For example, Amazon provides AWS IoT Core as its managed cloud service to connect IoT devices with AWS cloud services [7]. IoT devices can send data to AWS IoT core service, which then routes the data to other AWS services, such as EC2 or Lambda Function for computing and S3 Bucket for data storage.

In the recent years, smart home IoT systems with a hybrid architecture of both local and cloud computing have gained popularity in both research and real-world applications. A

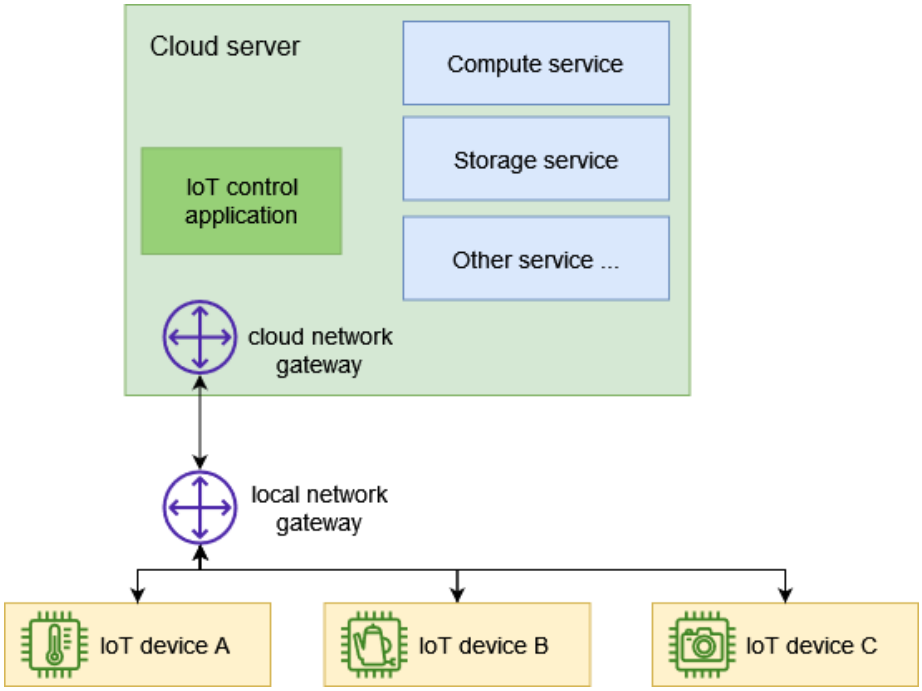


Figure 2.2: Example of an IoT System Cloud Setup

hybrid architecture can have some computation capability within the local network but also connect to cloud services. Such a setup combines the advantages of both local and cloud computing. Local computing can execute simple computational tasks and respond to IoT devices in a fast and reliable manner. It can also pre-process the collected IoT data to reduce the data throughput to the cloud. Cloud services can serve as a complementary part to provide additional computing power for more intensive tasks that local computing resources cannot handle and other functionalities, such as cloud storage solutions. One example of a hybrid IoT solution is Amazon's AWS IoT Greengrass framework. An AWS IoT Greengrass system can deploy AWS Lambda Functions or containers to the connected edge devices. Data collected by edge devices can be processed locally by these Lambda Functions and containers. Meanwhile, the IoT system can still connect with the AWS

cloud for data management, analytics, and durable storage [44].

2.2 Edge computing

The rapid growth of IoT applications comes with the rapid increase of data generated by IoT devices. The International Data Corporation (IDC) estimates in a report that, in 2025, there will be 41.6 billion connected IoT devices generating 79.4 zettabytes (ZB) of data [1]. The explosive amount of data will present a significant challenge on the networking and cloud computing infrastructure if most of these data need to be sent and processed on the cloud. Edge computing techniques have been heavily researched and developed in recent years to provide an in-house computing solution and reduce the data flow to the cloud [32].

Edge computing refers to processing, analyzing, and storing data closer to where it is generated. It allows these data to be processed and analyzed in near real-time and enables rapid responses [43]. Edge computing is typically provided by edge devices, such as IoT devices, or local edge servers, such as a smart home control console (for example, Amazon Alexa).

One advantage that edge computing has over cloud computing is smaller network latency. Even the trivial user-application interactions with cloud computing resources have a considerable amount of latency. Network latency deteriorates users' experience, especially in scenarios where users have deeply immersive interactions with the application, such as augmented reality [33]. In addition, new technologies that improve other aspects of network communication often have a negative impact on network latency. For example, multiple server-client handshakes and transmission throttling are widely used to improve communication reliability, which increases the latency. Security applications, such as firewalls,

and software-defined networks, such as virtual switch and network overlay, also increase the transmission time of a network packet [33]. Using edge computing to bring the computing resource closer to the user side is the ultimate solution to reduce network latency while preserving the improved technologies on other network aspects such as security and reliability.

Edge computing also helps to improve data privacy. If enough edge computing resources are provided, confidential data regarding a user can be fully processed within the user's local network, which is beneficial to end users' privacy.

Edge computing still has many challenges to overcome before it can be widely implemented in real work applications. One challenge is the heterogeneity of edge devices. This heterogeneity challenge exists in both the hardware aspect and software aspect. Edge devices made by different manufacturers have a wide variety of hardware architectures. In terms of software heterogeneity, an edge device may have a highly customized operating system that is optimized for the device's designed functionalities. Such an operating system provides an efficient run-time environment for the device's dedicated use cases, but it may impose difficulty in executing additional computing tasks [42]. In addition, the company which developed the software to run edge computing applications may not be the same company that manufactures the edge devices. This means the edge computing application developers may have limited knowledge of the edge devices, which creates more obstacles to integrating different edge devices into one unified computing cluster.

Another challenge is the software maintainability of edge devices. Unlike cloud computing infrastructures that can be easily accessed by a datacenter's site reliability engineers, edge devices located within users' local networks can be hard to access for software maintainers. Existing firewalls and complicated network routing rules can make it hard for maintainers to

discover, monitor, or upgrade the edge computing software. This puts an extra maintenance burden on the end-users of the edge devices. End-users may have to perform software installation, configuration, and periodic upgrades on their own to ensure edge devices can be run appropriately as an edge computing resource.

The third challenge is the networking complications. Edge devices within a local network usually do not have a public IP address. They may not even have a static private IP address. For example, in a residential house, it is common to have a router running dynamic host configuration protocol (DHCP) to dynamically assign private IP addresses to connected devices. This IP management strategy makes it hard to access these computing resources remotely. Network routing techniques, such as network address translation (NAT), domain name system (DNS), or network proxy, may be required to ensure stable network communication [37].

2.2.1 Azure IoT Edge

Microsoft integrates IoT devices to its cloud services via the Azure IoT Hub. The IoT Hub is hosted on the Azure cloud and manages communication between IoT devices and Azure's cloud services [11]. To enable edge computing, Azure provides the Azure IoT Edge framework. The framework includes three main components: IoT Edge modules, IoT Edge run-time, and IoT Edge cloud interface (see Figure 2.3) [45].

- IoT Edge modules are Docker compatible containers that contain executable binaries to run business logic on edge devices [45].
- IoT Edge run-time also runs on edge devices. It transforms a regular device into an edge computing resource by providing edge module management and communication.

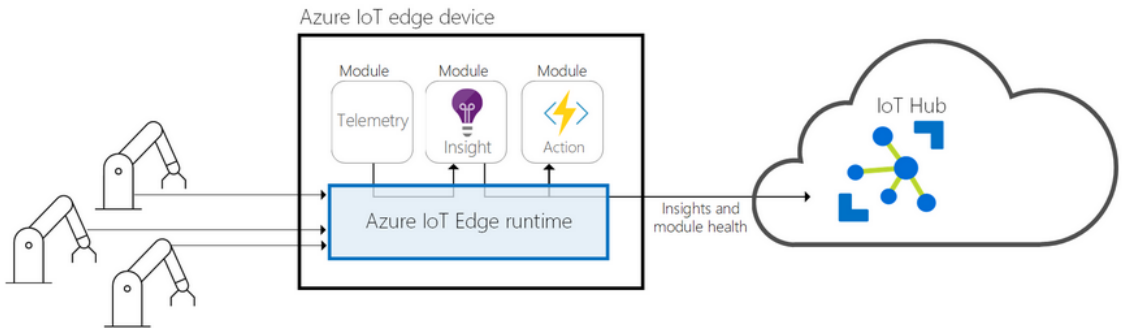


Figure 2.3: Azure IoT Edge Architecture [45]

IoT Edge run-time has two parts: IoT Edge agent and IoT Edge hub. IoT Edge agent manages the edge modules, including instantiating and monitoring the containers running on edge devices. IoT Edge hub serves as a counterpart of the Azure IoT hub. It functions as a local proxy to the IoT hub on the Azure cloud. This enables IoT devices to connect to edge computing services in the same way as how they would connect to Azure's cloud services [40].

- IoT Edge cloud interface is a cloud service to enable users to manage and monitor edge devices remotely [45].

2.2.2 AWS IoT Greengrass

Amazon provides its edge computing solution to IoT systems through the AWS IoT Greengrass [6]. Figure 2.4 shows how the AWS IoT Greengrass interacts with IoT devices and AWS cloud services. The IoT Greengrass core runs on an edge device within the local network of the IoT devices. It provides computing resources by deploying AWS Lambda functions or docker containers within the device. Once receiving data from an IoT device, the IoT Greengrass core will send the data to the corresponding Lambda function or con-

tainer to process the data. For data that cannot be handled with local services, the IoT Greengrass core sends the data to the AWS cloud through its local client. The IoT client on the cloud side will forward the data to the corresponding AWS cloud service for further processing.

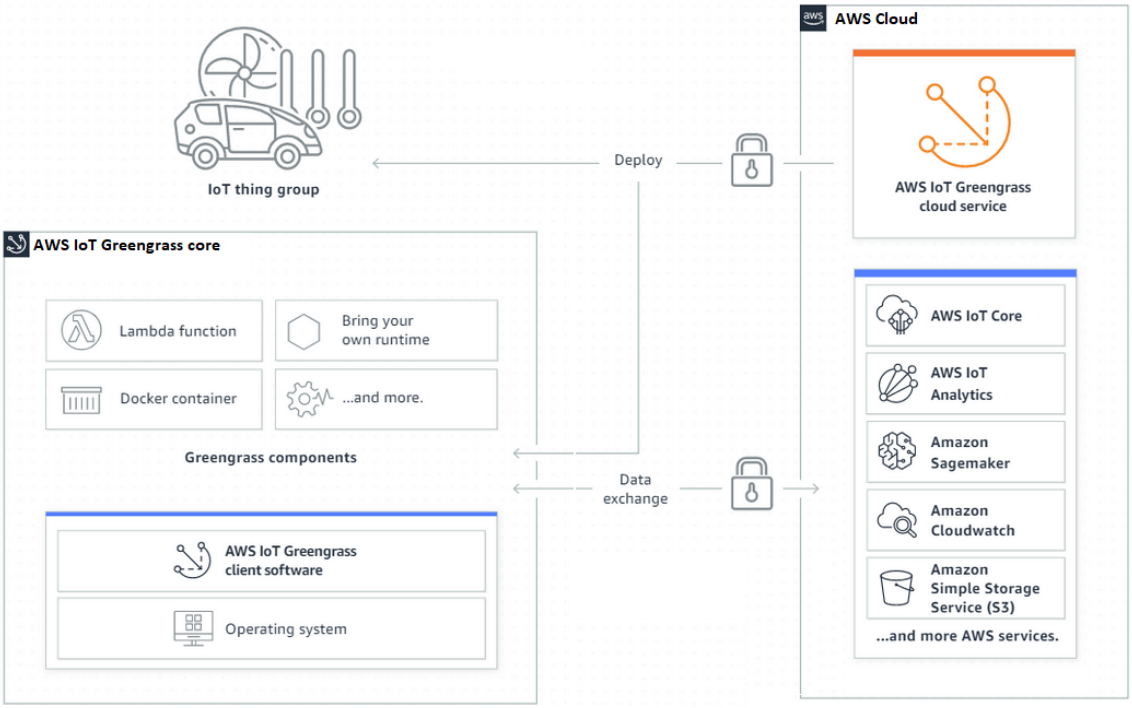


Figure 2.4: AWS IoT Greengrass Architecture [6]

2.2.3 Comparison of Azure IoT Edge and AWS IoT Greengrass

From an architectural point of view, Azure's edge computing solution for IoT and AWS's solution have a similar strategy, although each of them is implemented by the cloud provider's own in-house services. Both edge computing solutions run a management application on

an edge device (Azure's IoT Edge run-time or AWS's IoT Greengrass core). Each application communicates with both the IoT devices and the cloud. It also deploys and manages computing resources. In both solutions, the computing resources are deployed in the form of Docker containers or serverless computing services (Azure Function or AWS Lambda Function). Simple computing tasks will be executed on an edge device while the cloud extends the local capability with enhanced services.

2.3 Container

Container is an ideal technology to solve the challenges, such as device heterogeneity, in implementing edge computing [12]. Container is an OS-level virtualization technique. It provides an isolated environment for applications to run and makes these applications believe they have all the OS resources, such as CPU and memory. A container packages source code and its dependencies together to ensure that an application can access all the executable binaries that it needs. Compared to virtual machines, which virtualize at the hardware level, containers are lightweight in terms of both the size and the hardware resource requirement since they do not need to store the OS binaries or load OS binaries while running. This allows many containers to run on a single host operating system. And it allows containers to be efficiently built and easily shareable via files. Containers are widely used in micro-service architecture software applications. In such an architecture, each service runs in a separate container. Each container provides all dependencies that a service needs to run while isolating it from other running processes. This architecture allows each service to be independently deployable and maintainable. This is very well suited for today's agile software development strategy, where development cycles are short and service-upgrade processes need to be short and efficient.

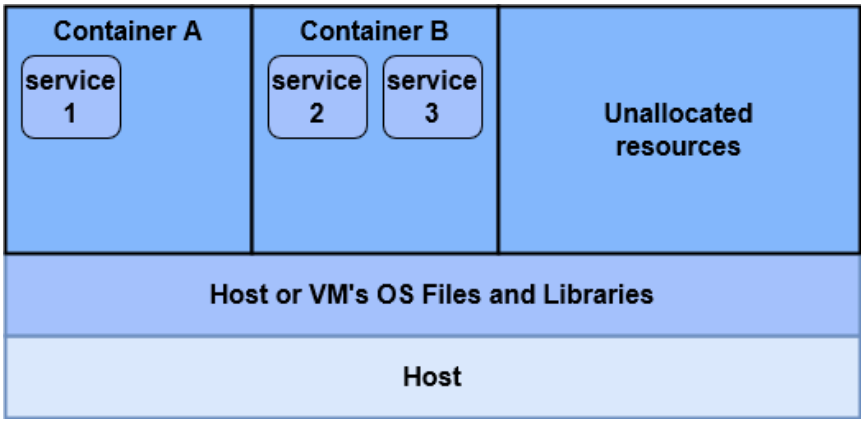


Figure 2.5: Container Overview [46]

2.4 Docker

Docker is currently the de facto container technology in the Linux world. Docker container was introduced in 2013 and quickly became popular in the software industry. Docker follows a client-server architecture. It consists of Docker client, Docker host, and Docker registry [3]. Figure 2.6 shows how these three components interact with each other.

- Docker client is used by users to interact with Docker daemon. It provides an easy-to-use command line interface (CLI) to make building and running containers very straightforward. The client converts command line inputs to Docker API requests and sends them to Docker daemon.

- Docker host provides the environment to execute commands and run applications. It runs Docker daemon, which is responsible for managing Docker-objects such as containers and images [3].

- Docker registry is a storage service for accessing Docker images. A Docker registry can be set up publicly or privately. Docker provides Docker Hub as a public Docker

registry where developers can publicly store and share their built Docker images. By default, Docker daemon searches the needed images from Docker Hub when initiating a container [3].

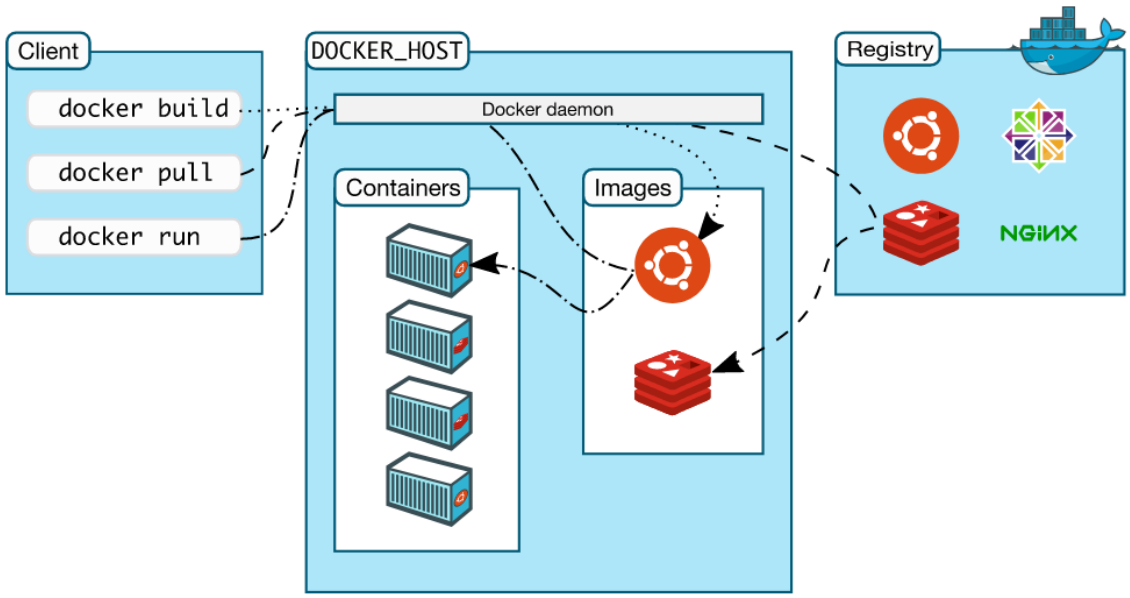


Figure 2.6: Docker Architecture Overview [3]

In a Linux environment, Docker container relies on two important Linux kernel features to achieve workspace isolation: cgroups and namespaces.

- "cgroups" stands for "control groups". Processes can be organized into these hierarchical groups whose resources usages, such as CPU and memory, can be limited and monitored [35]. In other words, cgroups can be used to set constrains on processes regarding how much resource each process can use.

- A namespace wraps a global system resource in an abstraction. For a process within that namespace, from the process' point of view, it has its own instance of the global

resource, such as its own network ports [26]. This abstraction makes a process appear to be isolated from other processes.

When running a container, Docker creates namespaces to provide resource isolation and uses cgroups to allocate computing resources to the container. To achieve file system isolation, Docker also mounts a dedicated volume and uses Linux kernel's chroot() command to use that volume as the root directory.

2.5 Kubernetes

Kubernetes is an open-source container orchestration system. As containers become popular in the software industry, because of the practice of micro-service application architecture and the isolation of one service per container, an entire software application usually ends up having lots of containers running. The number of containers continues to grow as the software scales up to meet the consumption demand from users. Kubernetes provides an automation solution for some container management tasks, including container deployment, scaling, rollouts and rollbacks, configuration management, network load balancing, and more.

Each Kubernetes system is called a Kubernetes cluster. A cluster can have one or more machines running containerized applications. Each machine is a Kubernetes node. For simplification purposes, from now to the end of the paper, we will refer to the Kubernetes cluster as "cluster" and Kubernetes node as "node".

There are two types of nodes: master nodes and worker nodes. At a minimum, each cluster must have one node: the master node. The main task of a master node is to manage all the worker nodes within the same cluster. A worker node is responsible for

running Kubernetes pods. A pod is a group of one or more containers. These containers share storage, networking resources, and container running configuration [21]. Figure 2.7 shows an overview of the Kubernetes cluster architecture.

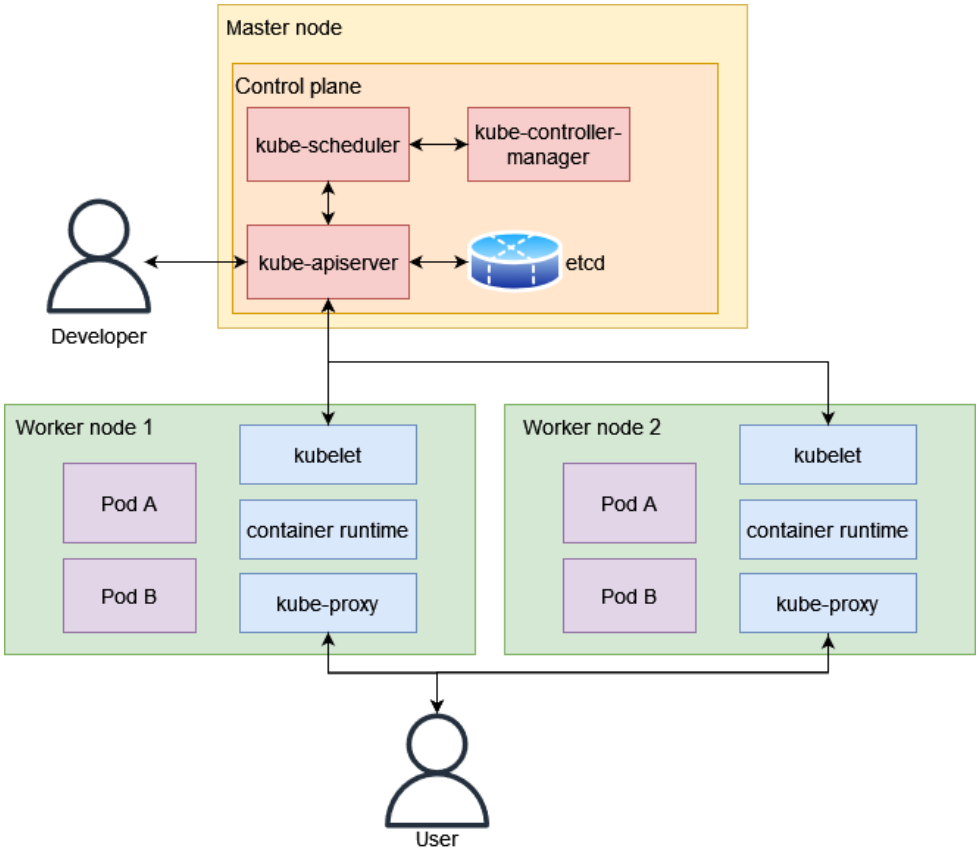


Figure 2.7: Kubernetes cluster architecture overview [10]

2.5.1 Master node

The Kubernetes master node manages its cluster's worker nodes. The master node assigns workload tasks to worker nodes. It makes decisions on which worker node to run which pod and monitors the status of each worker node and their pods. It also maintains a

description of the desired running state. Such desired state can be configured by the user. If an exception happens, the master node will try to adjust the cluster to bring its status back to the desired running state. For example, if a worker node suddenly crashes, the master node will quickly re-create the pods, which were previously running on the crashed node, on another worker node. The master node's control plane is the key to its node management functionalities. This paper will discuss the control plane in detail in Section 2.5.4.

2.5.2 Worker node

The Kubernetes worker nodes are designed to run pods that run application containers inside. Worker nodes perform task assignments from the master node and report the running status back. The most common task for a worker node is to run a pod with a container image or destroy a pod.

2.5.3 Pods

Pods are the smallest deployable units of computing that users can create and manage in Kubernetes [21]. A pod can have multiple containers running inside. These containers share the resources and running environment that the pod has. However, in the most common Kubernetes use case, each pod only runs one container.

2.5.4 Control plane

The master node has a control plane to provide important node management functionalities. The control plane consists of kube-apiserver, kube-controller-manager, cloud-controller-manager, kube-scheduler, and etcd [20].

- Kube-apiserver manages and configures data of the Kubernetes api objects, such as pods and services.

- Kube-controller-manager is a daemon that contains several Kubernetes controllers. Each controller runs as a non-terminating loop. Some examples of controllers that natively come with Kubernetes are replication controller and namespace controller [19]. These controllers monitor the state of the cluster via Kube-apiserver. If the actual state of the cluster deviates from the desired state, the responsible controller will try to make changes to bring the current state back to the desired state. In other words, controllers are essentially the operators who maintain Kubernetes' self-healing ability.

- Cloud-controller-manager is only available when running Kubernetes on cloud where cloud provider's API is needed to perform cloud specific tasks.

- Kube-scheduler is a scheduler responsible for assigning pods to nodes. After receiving a pod creation request, kube-scheduler checks the resource constraints and other placement requirements of the requested pod. It filters out the node candidates that fail to meet all placement requirements, ranks the remaining node candidates, and commands the highest-ranking node candidate to create the requested pod. The scheduler also allows extensions on multiple decision points during the node selection process, and it can also be completely replaced by a customized scheduler to meet specific user needs.

- Etcd is a strongly consistent and highly-available key-value store. Kubernetes uses etcd as the database to store all cluster data. This enables Kubernetes to keep track of the changes of the cluster state and be able to perform a rollback if needed.

2.5.5 Worker node components

Each worker node has the following components to maintain a running environment for pods: kubelet, kube-proxy and container runtime [20].

- Kubelet is a program that runs on every worker node. It manages all communications between the master node's control plane and the worker node. Kubelet receives pods specifications from the control plane's kube-apiserver, and runs the pod. It also constantly monitors the running status of pods and the status of the worker node. These health data are then sent back to the control plane for decision-making.

- Kube-proxy is a network proxy on each worker node. It maintains a set of network rules, such as an IP table, for pods. For example, when a new pod gets created in a worker node, the kube-proxy on the node will allocate an IP address for that pod. Kube-proxy is the responsible service to manage network traffic between pods within the same node and the traffic between pods and external services.

- Container runtime is the program to run a container within a pod. Typically, to run a container, the procedure includes pulling container images from an image registry, extracting the images, setting up a file system, allocating computing resources, setting up environment isolation, and more. Container runtime executes the procedure to run a container. Kubernetes supports not only Docker but also several other container runtime such as CRI-O.

2.5.6 Workloads

A software application runs on a Kubernetes cluster as a workload. A workload is in the form of one or more pods each containing one or more containers. Workloads provide an

abstraction layer on top of pods. Kubernetes has the ability to maintain a workload in its desired running state. This means, although an application's components are scattered across many pods, instead of putting effort into managing every single pod, application developers can work on the level of a single workload and let Kubernetes itself maintain the running pods under the hood. Kubernetes provides different types of workloads to meet different types of applications' requirements. These workload types include Deployment and ReplicaSet, StatefulSet, DaemonSet, Job, and CronJob.

- Deployment and ReplicaSet are suitable for running stateless applications. A ReplicaSet is a number of replicated pods. These pods have the same configuration and run the same containers inside. They usually run on different worker nodes. Incoming network traffic to a ReplicaSet of pods is usually load-balanced by the master node among the worker nodes. By using a ReplicaSet, Kubernetes can easily provide fault tolerance and high availability. Deployments are one of the most frequently used and managed components in Kubernetes. A deployment provides a way to manage a ReplicaSet, and it has other management capabilities such as updates, rollbacks, and scaling [9].

- StatefulSet is used to run stateful applications. A StatefulSet's pods can track their running states. This is done by providing a persistent storage to the pods.

- DaemonSet ensures that there is one pod running on every worker node of a cluster. This is helpful when a user wants to guarantee a service's availability locally within each worker node.

- Job and CronJob are used to run pods that only need to run at specific times. A Job runs according to a task description and stops when it completes the task. A

CronJob runs according to a schedule [24].

2.5.7 Networking architecture

Considering the high scalability and the distributed nature of a Kubernetes cluster, networking plays an important role in the Kubernetes architecture. Kubernetes' different components use different networking strategies. According to the origin and destination, Kubernetes' network traffic can be generally put into four categories: container-to-container, pod-to-pod, pod-to-service, and external-to-service [2].

Container-to-container is the network traffic between containers within the same pod. Kubernetes follows a "one IP per pod" model [36]. A pod gets one internal IP address. Containers within the same pod share the same IP address on different ports. Hence, from a container's point of view, talking to another container in the same pod is localhost communication.

Pod-to-pod is the network traffic between pods within the same node and pods across different nodes. Pod-to-pod communication relies on Container Networking Interface (CNI). CNI creates network interfaces in container network namespaces. It then assigns IP addresses to the created network interfaces and sets up the IP routing table [25]. Pods on the same node use the network interface within the shared network namespace to communicate. Pods across different nodes rely on the IP routing table to direct traffic through nodes. The native Kubernetes provides a simple Kubenet CNI. There are many third-party CNIs in the market with different rich features.

Kubernetes' service is an abstraction of a set of pods and a network access policy [22]. By design, a Kubernetes pod has a life cycle and does not exist permanently. Pods are usually managed in the form of workloads, which can be dynamically scaled. If a pod

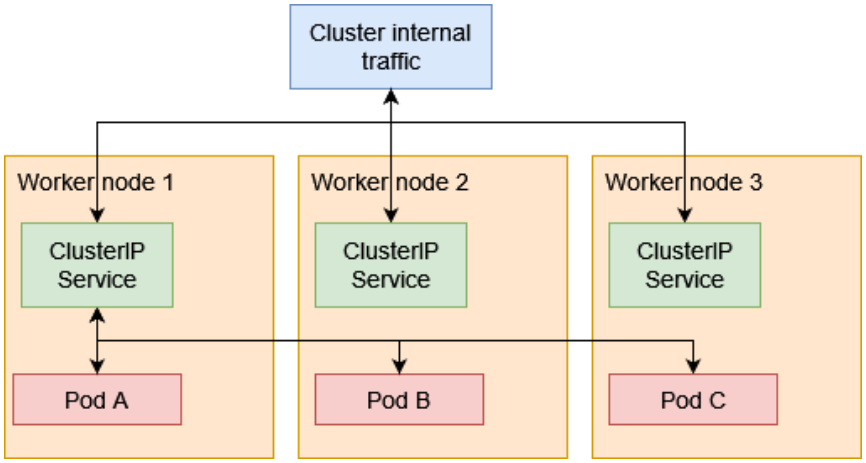


Figure 2.8: Kubernetes ClusterIP overview [23]

gets destroyed and re-created, there is also no guarantee that the newly created pod can get the same IP address as what the old pod has. Hence, managing networking policies and routing external traffic to pods can become complicated. Service alleviates the IP management complication by grouping pods together and providing a single entry point for network traffic. Kubernetes provides four different types of services: ClusterIP, NodePort, LoadBalancer, and ExternalName.

- ClusterIP service creates an internal IP address within the Kubernetes cluster. Pods managed by ClusterIP service can be reached from other workloads through this internal IP address (Figure 2.8). However, traffic from outside the cluster cannot reach ClusterIP managed pods.

- NodePort service builds on top of the ClusterIP service by opening a static port on each worker node to expose the ClusterIP service [23]. Traffic from outside of the cluster can go through the static port to reach the ClusterIP service and then be forwarded to pods (Figure 2.9).

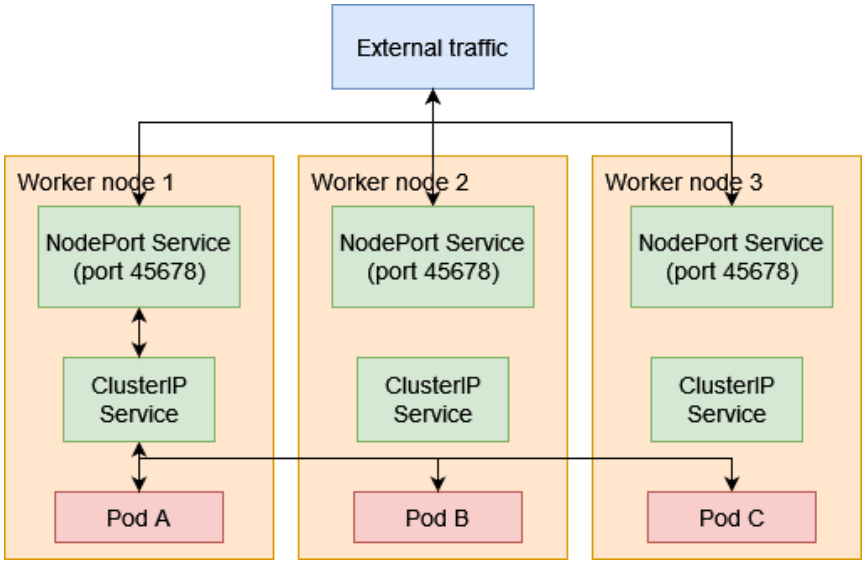


Figure 2.9: Kubernetes NodePort overview [23]

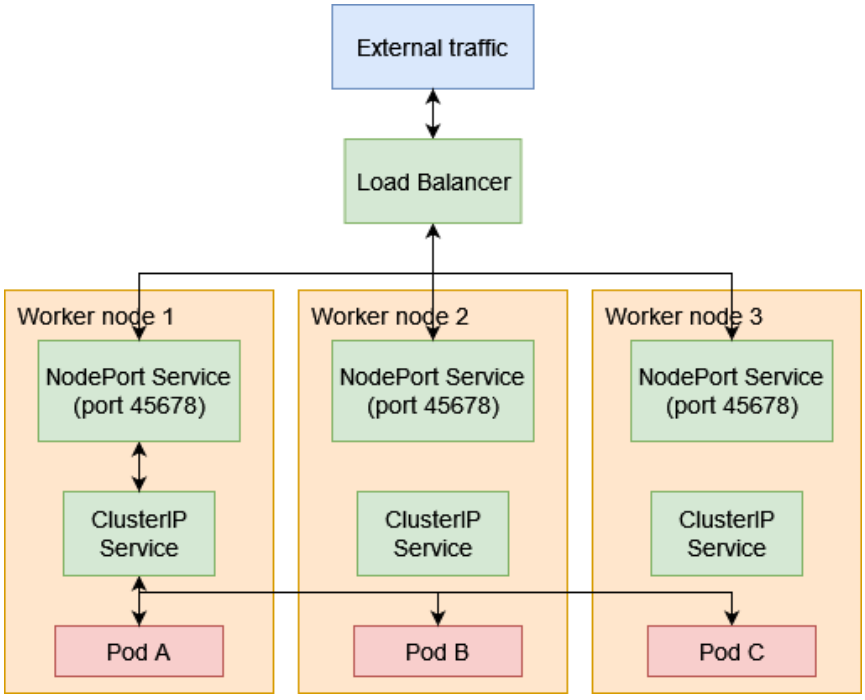


Figure 2.10: Kubernetes Loadbalancer overview [23]

- LoadBalancer service builds on top of the NodePort service. This service creates a load balancer targeting to the exposed node port of each worker node (Figure 2.10). External traffic will be load balanced to different worker nodes according to the load balance rule. The native Kubernetes doesn't include a default load balancer. Communities and different cloud providers have provided their own load balancer solutions [8].

- ExternalName service creates a DNS name for a service.

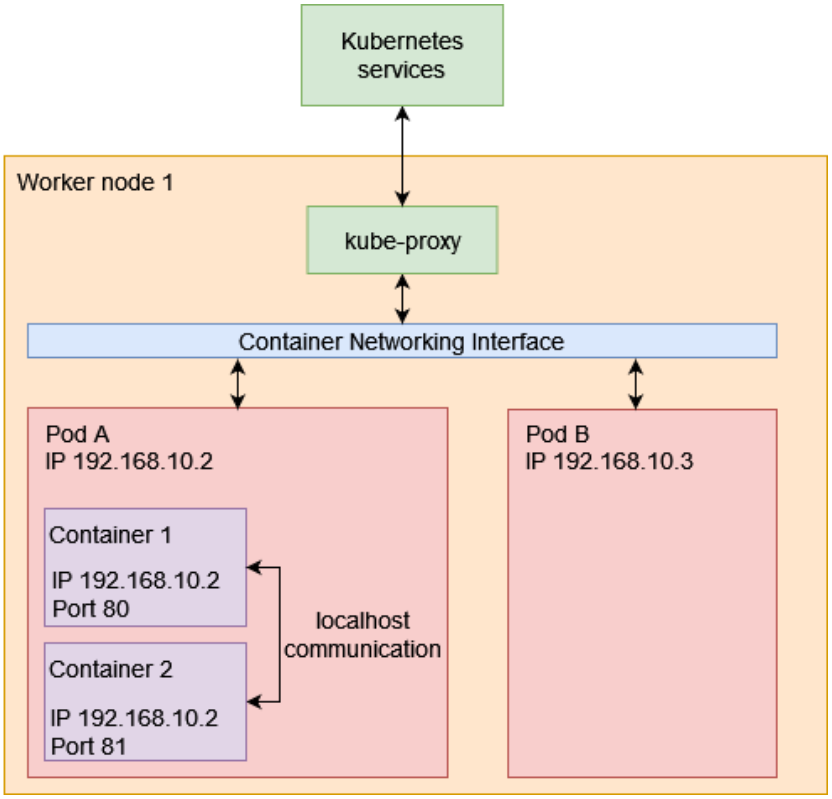


Figure 2.11: Kubernetes networking of a worker node [41]

Network traffic between pods and services relies on kube-proxy. Once created, a service will have an internal IP address. Network traffic sent to this internal IP will be redirected

to pods by kube-proxy. Kube-proxy constantly monitors the status of pods and makes adjustment on its network configurations, such as IP tables, to redirect network traffic according to what the service has specified.

External-to-service can be managed by ingress controller. With ingress controller, a user defines a series of ingress rules as the redirection rules. According to the rules, based on the network package type and URL, a network package will be redirected to a node port that the Kubernetes service listens to. From there, service-to-pod communication will forward the traffic to a pod.

2.6 Lightweight Kubernetes distributions

The native Kubernetes system is designed for large-scale container orchestration and management. Its large executable size and resource consumption during runtime are not always suitable to be deployed on edge devices with limited hardware resource. Some Kubernetes distributions aim to provide a lightweight Kubernetes solution for edge devices by stripping out some features that are rarely used in edge computing scenarios. Microk8s, Minikube and K3S are among the most popular lightweight Kubernetes implementations [28] [29] [15].

2.6.1 Microk8s

Microk8s [28] is developed by Canonical. In most cases, Microk8s is installed as a snap, which is also developed by Canonical. Snaps [38] are self-contained application packages that are built by bundling the application and its dependencies. Because of its self-contained nature, a snap package can work across various Linux distributions without

modification. This allows a snap application to be distributed and run easily across different Linux platforms. Having all dependencies within the installation package also makes it convenient to install, update and remove an application. Since Microk8s uses snap as the package manager, it can be easily installed with one CLI command under different operating systems. Considering that edge devices within a same household may be made by different manufactures and may have different hardware and software installation, having a simple, dependency-free installation method gives Microk8s an advantage.

At a minimum, Microk8s' recently released version (v1.21) needs 540MB memory to run [27]. Older versions require around 800MB to run. Considering the Kubernetes workload, the operating system and other programs' needs, Microk8s will need to be run on devices with at least 1GB memory.

Microk8s tries to keep in sync with the native Kubernetes upstream. By default, Microk8s keeps most of the bare-bone components of the native Kubernetes, including the control plane's kube-apiserver, kube-controller-manager, kube-scheduler, etcd and the worker node's kubelet, kube-proxy and container runtime.

2.6.2 Minikube

Minikube [29] is an open-source lightweight Kubernetes. It creates a virtual machine within the local host and deploys a simple, one-worker-node cluster in the VM. This method allows Minikube to run a Kubernetes cluster regardless of the operating system. Minikube is great for running Kubernetes locally for learning or testing purposes. However, it is not suitable for most of the production use cases since the production scenarios usually need a multi-node Kubernetes cluster.

2.6.3 K3S

K3S [15] was developed by Rancher and later donated to the Cloud Native Computing Foundation (CNCF). It is a production-ready lightweight Kubernetes designed for running on low-resourced devices. It also has binaries that are optimized for ARM. K3S's package can be a single small binary less than 40MB. K3S achieves the small size by removing some Kubernetes features that are rarely relevant for edge device use cases and by replacing some Kubernetes components with lightweight alternatives. For example, by default, K3S's master node, which is also called the server node, uses the ultra-lightweight SQLite as the database instead of the native Kubernetes' etcd storage. K3S supports multi-node clusters (see Figure 2.12). Unlike the native Kubernetes that does not provide a default load balancer implementation, K3S includes a layer 4 load balancer and a layer 7 ingress controller based on Traefik proxy by default. Since it is necessary for an edge computing cluster to communicate with devices outside of the cluster, the load balancer and ingress controller are must-have components when running applications with Kubernetes on edge devices. Having such support natively with K3S alleviates users from implementing their own solutions on resource-constrained devices.

Table 2.1 shows the hardware resource requirements and supported platforms to run each lightweight Kubernetes implementation. Among the three, K3S requires the fewest resources to run. Table 2.2 shows some important aspects of the three Kubernetes implementations. All three are open-source, which allows developers and communities to participate in future development. Given that Kubernetes on edge devices and the edge computing field, in general, are all relatively new and under-developed, open-source resources can greatly drive researchers' and developers' involvement and unleash a huge potential on the products. Hence, it is an important aspect to be considered. Minikube supports the

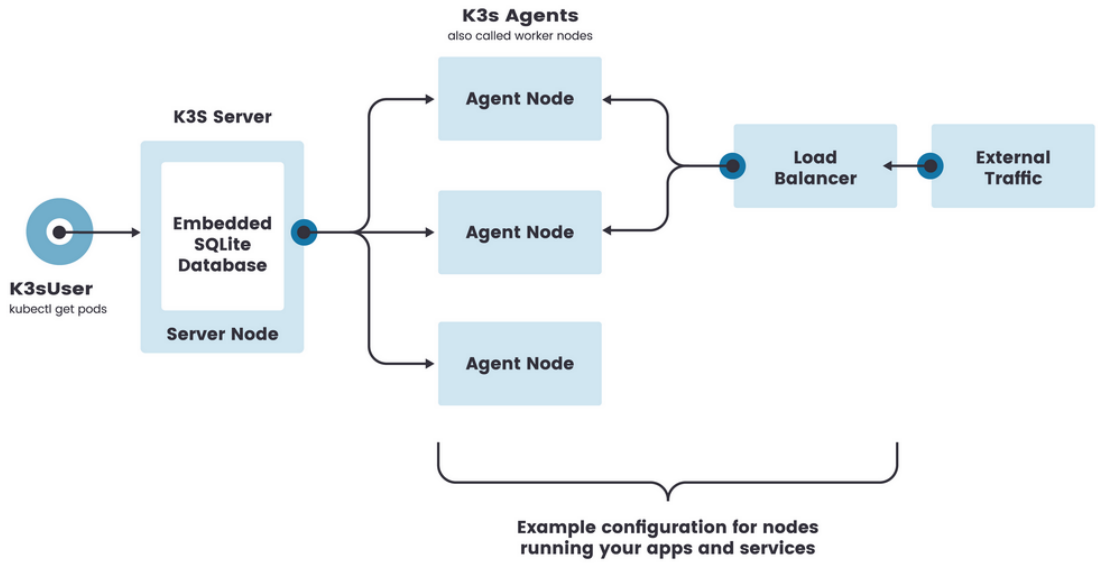


Figure 2.12: K3S Multi-node Architecture [14]

most hardware platforms out of the three Kubernetes implementations. However, it is not designed to be used in production, and it does not support running a multi-node cluster. So, it is hard for Minikube to be used in a production environment. K3S supports more hardware platforms than Microk8s, and it also has a lower hardware resource requirement than Microk8s, which gives K3S an advantage over Microk8s on edge devices.

Distribution	**Package size**	**CPU minimum**	**RAM minimum**	**Platforms**
Microk8s	Around 200MB	1 CPU	1 GB	x86-64, ARM64
Minikube	Less than 100 MB	1 CPU	1 GB	x86-64, ARM64, ARMv7, ppc64, S390x
K3S	Less than 100 MB	1 CPU	512 MB	x86-64, ARM64, ARMv7

Table 2.1: Lightweight Kubernetes distributions installation requirements [27] [16][29]

Distribution	**Multi-node cluster support**	**Production-ready**	**Open-source**
Microk8s	Yes	Yes	Yes
Minikube	No	No	Yes
K3S	Yes	Yes	Yes

Table 2.2: Lightweight Kubernetes distributions aspects

Chapter 3

Related Works

3.1 Fog-based architecture for smart home IoT system

Aykut Kanyilmaz and Aydin Cetin proposed a fog-based architecture where edge devices can be registered as private nodes to provide computing services within a local network [17]. Their system has three layers: a cloud layer, a fog layer, and a client layer (Figure 3.1).

The fog layer contains devices that provide computing resources to process data. The cloud layer is a remote service that manages the devices and services of the fog layer. The remote service handles fog device registration and authentication and provides data storage. Under the management of the cloud layer, devices in the fog layer can be dynamically added or removed from the system. The client layer contains the devices at an IoT system's

endpoints, such as sensors and motors. The idea of having computing resources that can be dynamically added and removed is similar to what our system can do. However, unlike Aykut's system of having the device management done on the cloud, our system runs management services within the local network. Moreover, our system uses Kubernetes to orchestrate containers, which Aykut's system does not.

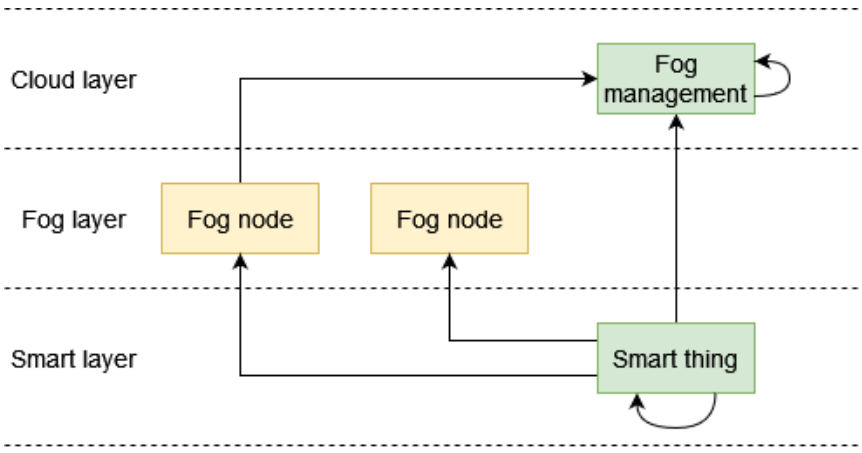


Figure 3.1: Fog-based architecture for smart home IoT systems [17]

3.2 Traffic-aware horizontal pod auto-scaler

Le Hoang Phuc, Linh-an Phan, and Taehong Kim proposed a traffic-aware horizontal pod auto-scaler (THPA) [30] that adjusts the number of Kubernetes pods on worker nodes based on network metrics. Their system has three layers: a cloud layer, an edge layer, and end devices (see Figure 3.2). The cloud hosts the Kubernetes master node, which manages Kubernetes deployments and contains a metrics server that records data for pod auto-scaling. The edge layer has edge devices that are used as worker nodes to provide computing resources. The end devices are the clients that collect data and send the data to edge devices for processing. In their system, each worker node gathers a network metric

to measure the traffic condition between the edge device and the connected client. Based on the network metric of each worker node, the master node increases the number of pods on traffic-intensive worker nodes and reduces the pod number on low-traffic worker nodes. The idea of using Kubernetes to horizontally scale pods on edge devices is similar to what our system can do. However, unlike their native Kubernetes design, our system uses the lightweight K3S, which is designed to be more suitable to run on edge devices. In addition, their system has the master node on the cloud and assumes edge devices are always available. Our system places the master node within the local network, and it aims to deploy Kubernetes pods on low availability edge devices.

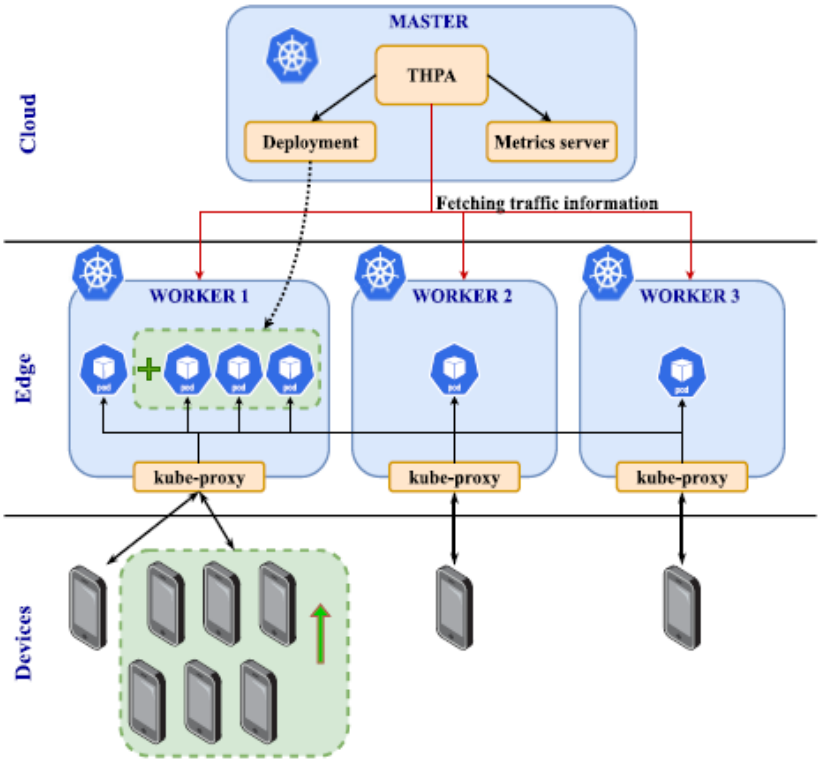


Figure 3.2: THPA in Kubernetes-based Edge computing Architecture [30]

3.3 Implementation of an Edge Computing Architecture Using OpenStack and Kubernetes

Endah Kristiani, Chao-Tung Yang, Yuan Ting Wang, and Chin-Yin Huang proposed an edge computing architecture where a Kubernetes cluster across the cloud and edge is deployed with a master node on the cloud side and worker nodes on the edge side [18]. Their system has a cloud layer, an edge layer, and a device layer (see Figure 3.3). The device side contains the IoT devices. The edge side has edge devices that run the Kubernetes worker nodes, which have containers to host some computing services such as data reception and pre-processing. The cloud side hosts the Kubernetes master node and additional computing services that handle more complex operations. The cloud side infrastructure is deployed with OpenStack. Compared with their system, our system uses K3S and places the master node on the local network.

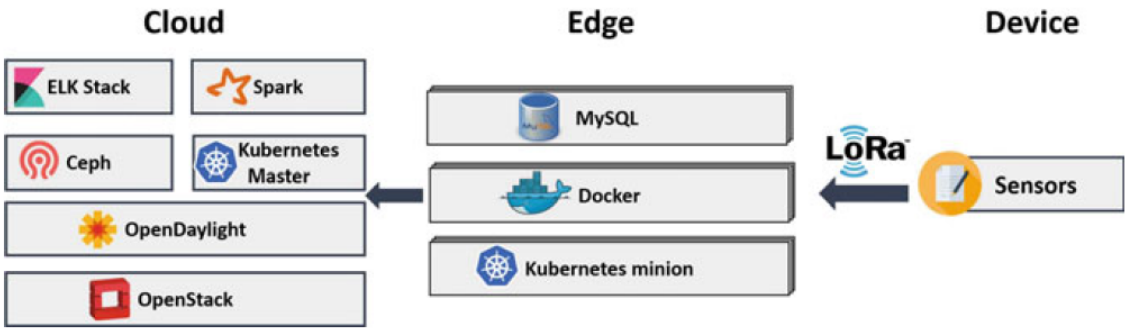


Figure 3.3: Edge computing Architecture with Kubernetes and OpenStack [18]

Chapter 4

System Description

4.1 Overview

We propose a system that can deploy computing tasks to low availability devices that are not dedicated to running as part of a smart home IoT system. By doing so, we enable heterogeneous home electronics, such as personal laptops, smart TVs, and gaming consoles, to be used as edge computing resources, even though these electronic devices are not designed to provide computing service to IoT applications. Our system uses K3S, a lightweight Kubernetes, as the container orchestration system to deploy and manage containers in which computing tasks will be executed. Given that low availability devices, such as smart TVs, are not always running, our system monitors the running status of these devices and automatically scales the computing container deployment based on edge device availability. The system consists of the following components:

1. A service to create, update and remove computing deployments on edge devices

through K3S based on the availability of edge devices.

2. A service to cache the IoT data and re-direct the data to compute resources on the cloud or local edge devices based on edge device availability.

3. A database to record edge device availability and existing computing deployments on available edge devices.

As discussed in Chapter 2, in a smart home IoT system, computing services can be hosted remotely on the cloud or hosted on a device within the local network. Hence, we will discuss how our system can work under two scenarios: 1. Extend a computing service from the cloud to a low availability edge device, 2. Extend a computing service from a dedicated device within the local network to a low availability edge device.

4.2 Extend computing services from the cloud to low availability edge device

Figure 4.1 shows the architecture of an IoT system with computing services hosted on the cloud. And Figure 4.2 shows how our system can be added onto a cloud-computing-only architecture to enable having additional computing services on low availability edge devices. In our system, each edge device within a home's local network has K3S installed and is registered as a K3S worker node. The K3S master node runs on a high-available device within the local network. This device can be an extra dedicated hardware component or a device that has already been part of the existing IoT system. Two services are also running in the same device as the K3S master node: an edge device monitoring service and an IoT traffic routing service.

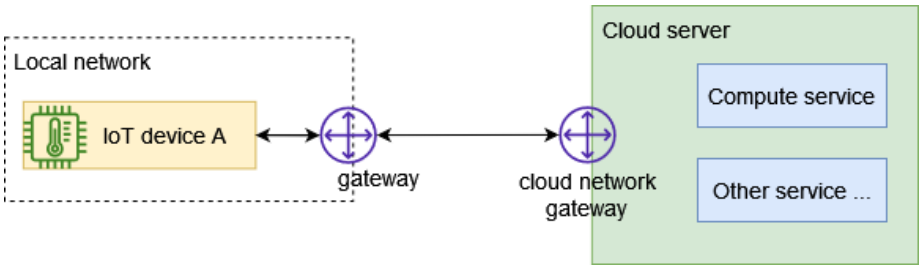


Figure 4.1: IoT system with computing service on cloud

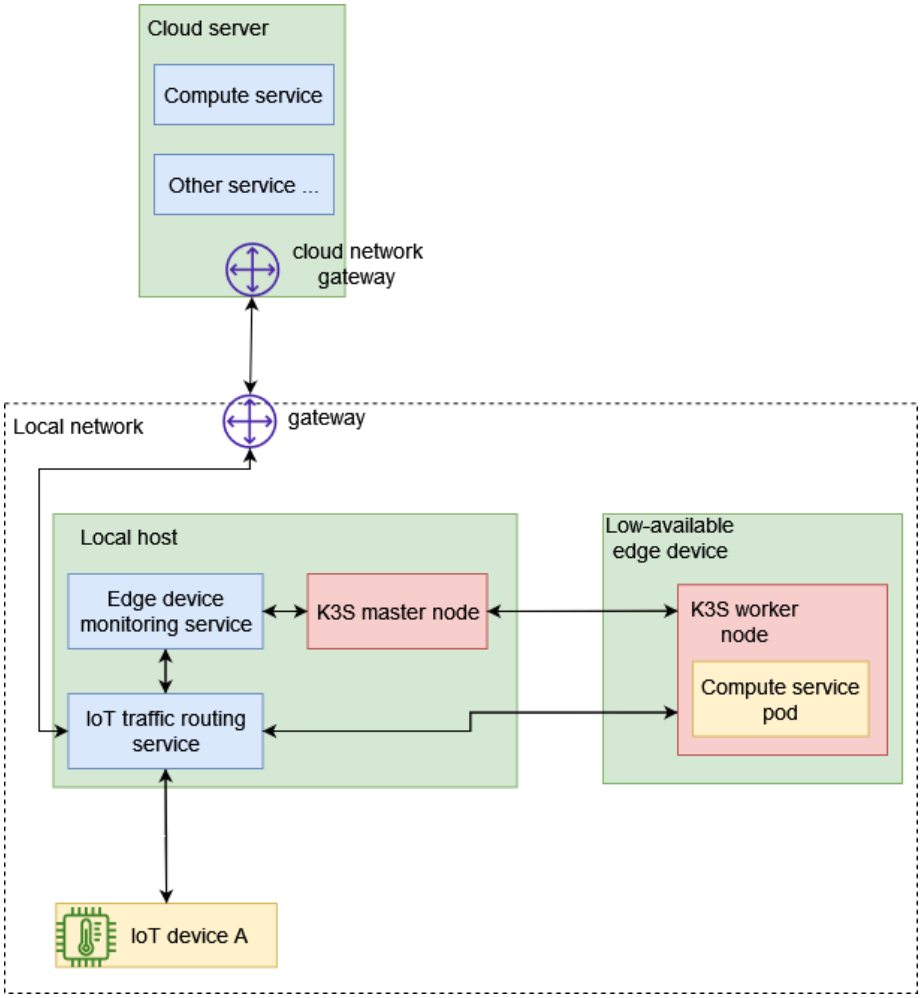


Figure 4.2: System of extending computing service from cloud to a low availability edge device

4.2.1 Edge device monitoring service

Edge device monitoring service periodically talks with the K3S master node to check the availability of edge devices in the local network. If it discovers an edge device that has become available, it will create a Kubernetes deployment in K3S to deploy pods to the discovered worker node running on the edge device. Each deployed pod contains a docker container to execute computing tasks. After a pod has been successfully deployed, the monitoring service will record the information of the discovered worker node and the newly deployed pod in a database. When the edge device goes offline, the monitoring service will retrieve the pod information from the database and delete the pod. Figure 4.4 shows the workflow explaining how the service creates a Kubernetes deployment to deploy a pod. Figure 4.5 shows how the service deletes a Kubernetes deployment when the edge device goes offline.

The edge device monitoring service uses the Kubernetes API with the command: *kubectl get nodes -o wide* to monitor worker node availability. By Kubernetes' design, the master node constantly monitors the status of worker nodes. When an edge device comes online, the API response of that device's worker node will change from "NotReady" to "Ready". This signals the monitoring service to start the procedure of deploying pods onto the worker node.

The pods are deployed as the Kubernetes deployment workload. The first step is to fetch a deployment YAML file that describes how the deployment will be created. A sample deployment YAML file is shown in Figure 4.3. This YAML file instructs Kubernetes to create a deployment named "*compute-image*". "*replicas*" specifies that there will be one pod created for this deployment. "*template*" specifies how the pod will be created. In this case, the pod will host a container with image "*my-docker-hub/compute-image:1.0.0*" on

port 8080. In our system, the deployment YAML files can be pre-defined and stored in local storage, or they can be fetched from a remote resource via some API calls.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: compute-image
  labels:
    app: compute-image
spec:
  selector:
    matchLabels:
      app: compute-image
  replicas: 1
  template:
    metadata:
      labels:
        app: compute-image
    spec:
      containers:
      - name: compute-image
        image: my-docker-hub/compute-image:1.0.0
        ports:
          - containerPort: 8080
```

Figure 4.3: Sample YAML file of a Kubernetes deployment

After fetching the deployment YAML file, the monitoring service calls "*kubectl apply -f deployment-file.yaml*" to instruct K3S to create a deployment and deploy a pod to the edge device worker node. After the pod has been successfully deployed, the monitoring service records the pod information in a local SQLite database. The database records ensure the monitoring service is fully aware of what Kubernetes pods have been deployed. The monitoring service also sends a command to the IoT traffic routing service to make it redirect the computing requests to the newly created computing service running on the pod.

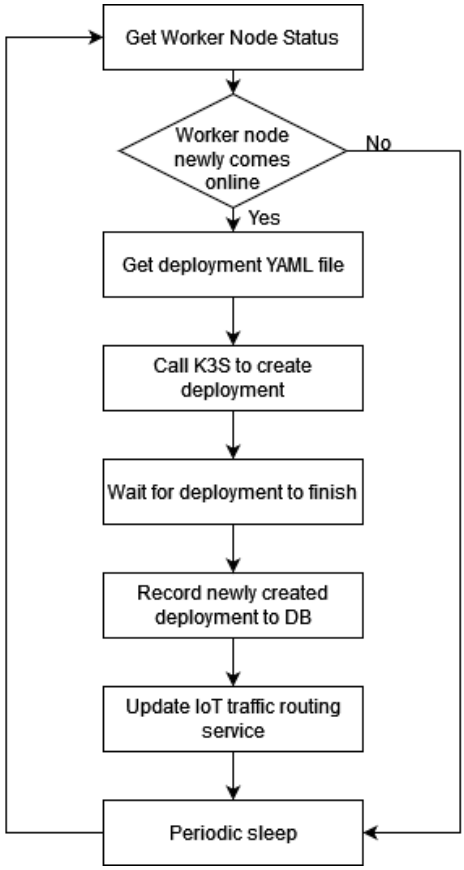


Figure 4.4: Edge device monitoring service workflow to create a deployment

The monitoring service has two methods to discover when an edge device goes offline. The first method is that the edge device runs a program that will be executed upon device shutdown, which sends the monitoring service a message about the shutdown. Another method is that the worker node's status changes from "Ready" to "NotReady" after the device shutdown. Once discovering that an edge device is going offline, the monitoring reads from the database about which deployment was created to run on that edge device and sends a command to K3S to delete that deployment. After the deployment gets deleted successfully, the monitoring service removes the deployment record from the database and

updates the IoT traffic routing service to redirect computing requests to the cloud.

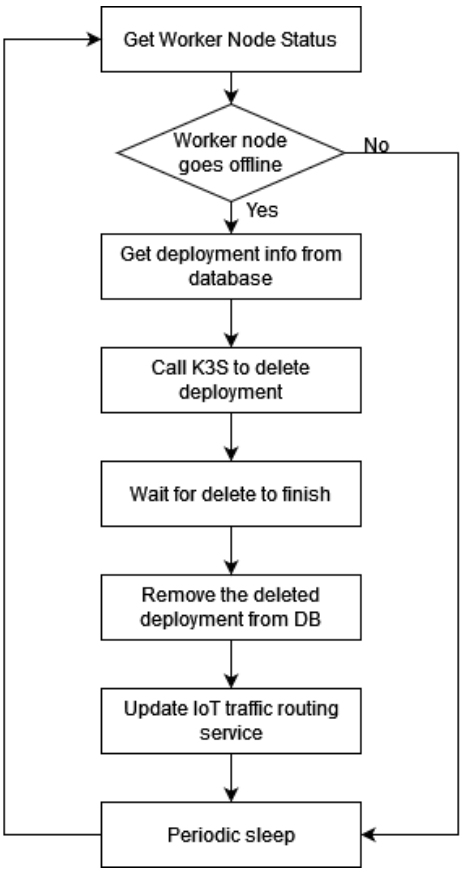


Figure 4.5: Edge device monitoring service workflow to delete a deployment

4.2.2 IoT traffic routing service

The IoT traffic routing service is responsible for routing the requests from IoT devices to computing services. It maintains a routing table of the computing services and destination IPs. This routing table gets updated when the monitoring service creates or removes Kubernetes deployments on edge devices. When there is no edge device online within the local network, the routing service directs all requests to the remote services on the cloud.

When a computing service is hosted on an edge device locally, the routing service directs requests to the local computing service.

4.3 Extend a computing service from devices in the local network to low availability edge devices

We also propose a method to extend computing service from a dedicated device in the local network to low availability edge devices. The setup is similar to what we designed in Section 4.2. We have a K3S master node running on a high-available device within the local network. Edge device monitoring service constantly checks edge device availability and manages Kubernetes deployments. The difference from Section 4.2 is, in this setup, there is a dedicated device locally to provide computing service. This dedicated device is configured as a K3S worker node, and it is always available. A load balancer is used to balance the traffic among the worker nodes. Figure 4.6 shows the overview of the setup.

Figure 4.8 shows the workflow of how the edge device monitoring device deploys a Kubernetes pod to a low availability edge device when it goes online. In this scenario, since there has been a dedicated worker node always running, the Kubernetes deployments and pods that host the computing services have existed. Hence, the monitoring service does not need to fetch a deployment YAML file to create a deployment. It only needs to instruct the K3S to increase the number of pod replicas of a deployment. As the number of pod replicas increases, a new pod will be created by K3S. To ensure that this new pod will be created on the low availability edge device, we can add a "pod-anti-affinity" scheduling

rule to the deployment configuration as shown in Figure 4.7. This scheduling rule instructs the Kubernetes scheduler to create the pod on a worker node where the same type of pod does not exist. After deploying the new pod to the edge device, the monitoring service records the new pod information in a database (see Figure 4.8). When an edge device goes offline, the monitoring service fetches the pod information from the database and instructs the K3S to reduce the number of pod replicas of the deployment.

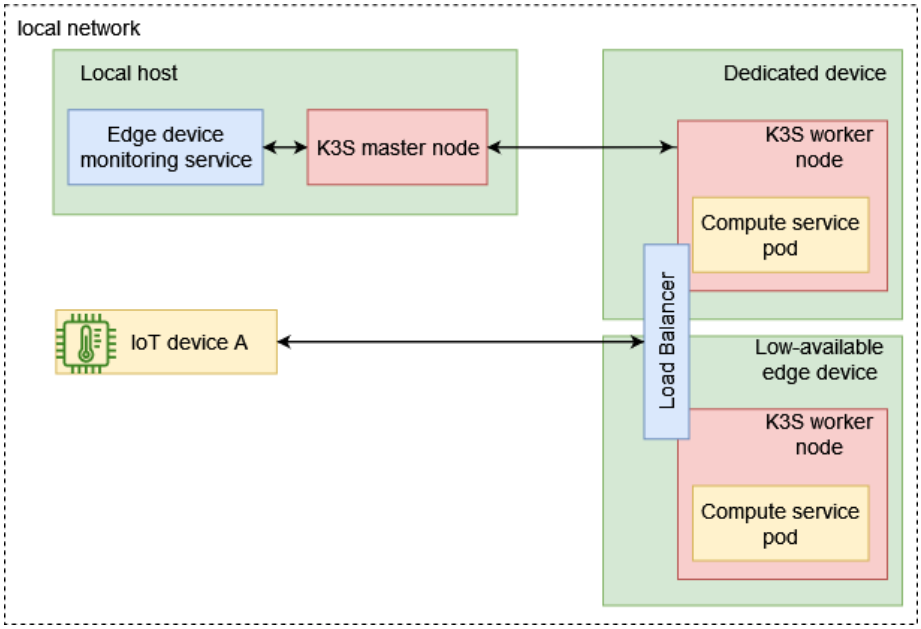


Figure 4.6: System of extending computing service from a local device to a low availability edge device

```
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - compute-image
          topologyKey: kubernetes.io/hostname
```

Figure 4.7: Sample YAML file of Kubernetes pod anti-affinity configuration

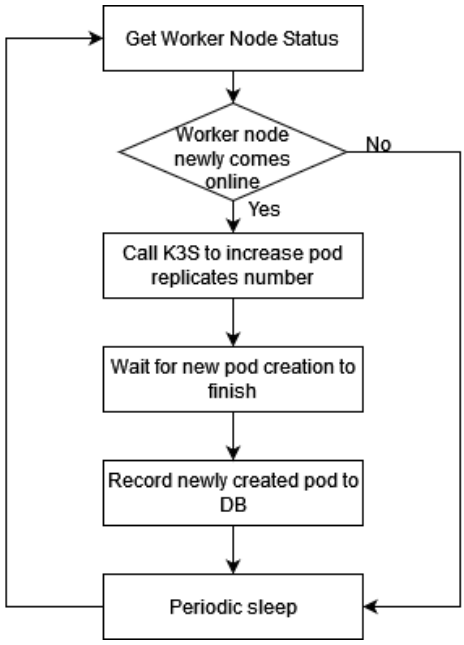


Figure 4.8: Edge device monitoring service workflow to increase pod replicas

Chapter 5

System Evaluation

To evaluate our system design, we built a system prototype with a set of Raspberry Pi devices. K3S was installed on these Raspberry Pi devices to form a Kubernetes cluster. The two scenarios, extending a computing service from the cloud to an edge device and extending a computing service from a local dedicated device to an edge device, were tested separately.

5.1 Testing for extending a computing service from the cloud to a low availability edge device

We installed K3S on two Raspberry Pi devices to form a Kubernetes cluster. These Raspberry Pi were both model-4 version with 4GB of RAM and a 16GB micro-SD card for read-only storage. The official x64 Raspbian OS image was used as the operating system. One Raspberry Pi hosted the Kubernetes master node. This device was always running.

The other Raspberry Pi simulated a low availability edge device. This low availability device could be online and offline without any specific schedule. A third Raspberry Pi device was used to simulate an IoT device. This device was not part of the Kubernetes cluster. Its only job was to send and receive data from computing services. These devices communicated with each other via Wi-Fi. In addition, we deployed an EC2 instance on the AWS us-east-1 region to simulate a cloud computing service. Figure 5.1 shows our overall testing setup.

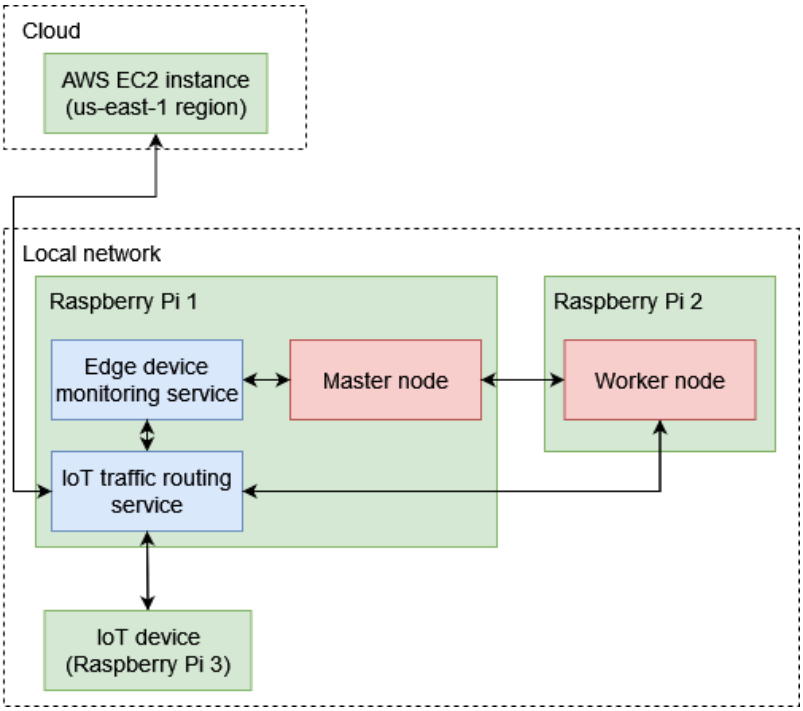


Figure 5.1: Overall setup for testing extending computing services on the cloud to a low availability edge device

We built a Docker container image of a simple REST API program, which was used as the computing service. At the beginning of the test, the container ran on the EC2 instance on the cloud, and our worker node (Raspberry Pi 2) was offline. Our IoT device (Raspberry

Pi 3) sent requests at the rate of 1 request per 10 seconds to the IoT traffic routing service, which then forwarded the requests to the EC2 instance. We then powered on Raspberry Pi 2 to bring the worker node online. The edge device monitoring service discovered that the worker node was online and instructed the master node to deploy a Kubernetes pod to run our container image. Once the pod was successfully deployed, all the IoT requests were directed to the local worker node instead of the cloud.

To show this transition process, we list the computing service response time and the service location during the experiment in Table 5.1. The requests sent by the IoT device were first directed to the cloud and later directed to the local worker node. Moreover, the response time from the computing service was significantly reduced after switching to the local worker node (see Figure 5.2). This reduction was because the worker node was in the same local network as the IoT device, which alleviated network traffic overheads.

Request timestamp (min:sec)	**Response time**	**Service location**
0:00	1.069422	Cloud
0:10	1.077199	Cloud
0:20	1.065179	Cloud
0:30	1.066359	Cloud
0:40	1.068678	Cloud
0:50	1.082160	Cloud
1:00	0.036583	Local
1:10	0.029707	Local
1:20	0.027555	Local
1:30	0.035993	Local
1:40	0.032074	Local

Table 5.1: Computing service response time and service location

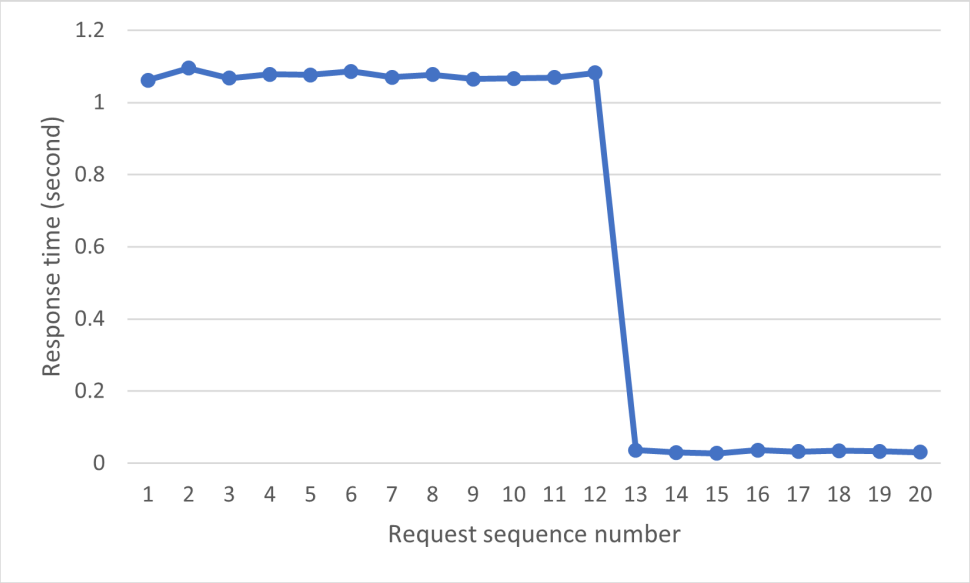


Figure 5.2: Computing service response time during the test

5.2 Testing for extending computing services from a dedicated local device to a low availability edge device

We installed K3S on three Raspberry Pi devices to form a Kubernetes cluster. One Raspberry Pi hosted the Kubernetes master node. Another Raspberry Pi hosted a worker node. This worker node was always running to represent a dedicated local device that provides computing services. A third Raspberry Pi hosted a second worker node as a low availability edge device. A fourth Raspberry Pi was set up outside of the Kubernetes cluster as the IoT device. We used K3S's default Traefik proxy service as the load balancer. Figure 5.3 shows the overall testing setup.

We used the same Docker image in Section 5.1 as the computing service. At the beginning

of the test, the dedicated device (Raspberry Pi 2) was online, but the low availability device (Raspberry Pi 3) was offline. We saw that the computing service response was all from Raspberry Pi 2. Then we powered on Raspberry Pi 3 to bring the low availability worker node online. We saw that a Kubernetes pod was deployed on the new worker node, and the computing service requests started to be load-balanced across two worker nodes. Table 5.2 shows this transition process. Before timestamp 0:50, all the computing service requests were handled by the dedicated device (Raspberry Pi 2). After 0:50, Raspberry Pi 3 was online and started to process requests.

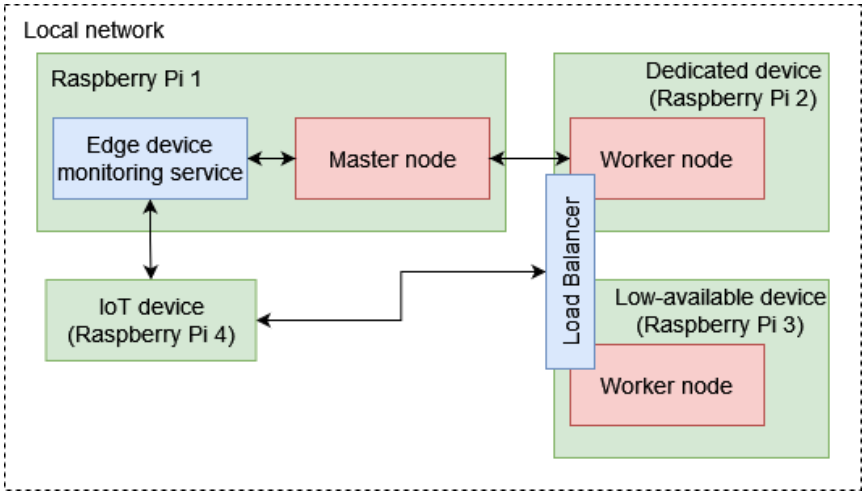


Figure 5.3: Overall setup for testing extending computing services on a dedicated local device to a low availability edge device

Since the dedicated device was in the local network, when the low availability edge device started to provide an additional Kubernetes pod, we did not see a significant reduction in network response time. However, we expected to see that the computing service was able to handle more load of network requests when the low availability device was online since the network traffic was load-balanced between the dedicated device and low availability device. To verify this, we did load testing on our system with Apache Jmeter. For a single

computing service program, we set the number of acceptable requests to be 60 requests per minute at maximum. This threshold simulates the maximum request load that a service can handle. Then we ran Jmeter to do a load test on our system.

Request timestamp (min:sec)	Computing Service location
0:00	Raspberry Pi 2
0:10	Raspberry Pi 2
0:20	Raspberry Pi 2
0:30	Raspberry Pi 2
0:40	Raspberry Pi 2
0:50	Raspberry Pi 3
1:00	Raspberry Pi 2
1:10	Raspberry Pi 3
1:20	Raspberry Pi 2
1:30	Raspberry Pi 3

Table 5.2: Computing service response location during the test

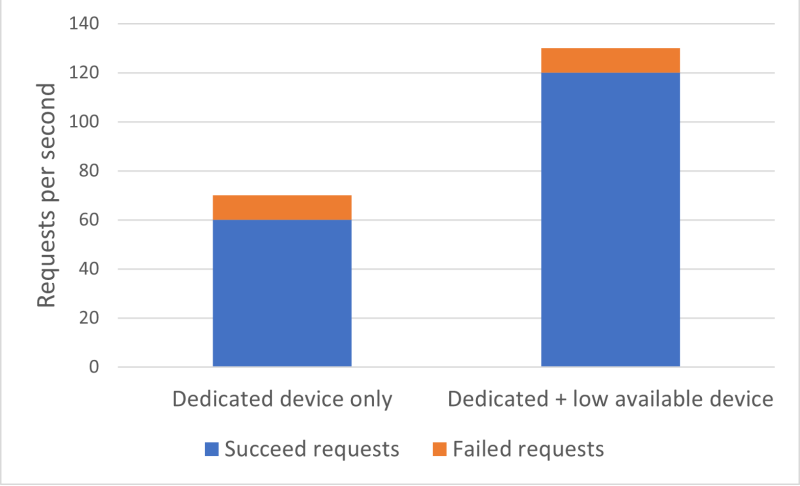


Figure 5.4: Computing service load testing result

Figure 5.4 shows the testing result. With only the dedicated device running, Jmeter sent 70 requests in a minute. 60 requests were successful, and 10 requests failed. With the

low availability device online, Jmeter was able to send 120 requests successfully. This demonstrates that having an additional computing service on the low availability device can help to alleviate the computing load.

Chapter 6

Conclusions and Future Work

We have presented a system that can enable low availability edge devices as an edge computing resource in a smart home IoT system. Our system uses K3S, a lightweight Kubernetes implementation, to orchestrate containers on edge devices to enable computing services to run on these devices. The system has two benefits. First, it can extend computing services from the cloud to a low availability edge device. Second, it can extend computing services from a dedicated local server to a low availability edge device. After the initial setup, the system can automatically monitor the running status of low availability devices and instruct Kubernetes to deploy, scale, or remove container deployments on these devices. Network traffic containing computing requests can be properly routed by our system to a low availability device when it is online.

We built a prototype system with Raspberry Pi devices and evaluated it. When extending a computing service from the cloud to a local low availability device, the system is able to re-direct computing service requests to the local device, which significantly reduced the

network response time between IoT devices and the computing service. When extending a computing service from a dedicated local server to a low availability edge device, the system can increase the maximum workload that the computing service can process.

Our system can be potentially improved in the future in several aspects. In terms of the initial system setup, we currently need to configure each edge device separately. Although this process is straightforward, it is repetitive work. So, this process can be automated with some scripts or by using IT infrastructure automation tools such as SaltStack and OpenStack. Regarding deploying Kubernetes pods to newly online edge devices, the current system deploys Kubernetes pods once it discovers a low availability device comes online. In the future, a more comprehensive matrix can be used to decide if would be beneficial to deploy Kubernetes pods on low availability devices. For example, the system can measure the load of computing requests and only deploys new Kubernetes pods when the computing load is high. In addition, optimizations can be made on the Kubernetes scheduling mechanism to enable the system to deploy different computing services on the different types of edge devices. For example, a gaming console with a good graphic card may be suitable for executing parallel computing tasks, whereas a laptop may be better for handling serial computing tasks.

Bibliography

[1] David Reinsel Carrie MacGillivray. *Worldwide Global DataSphere IoT Device and Data Forecast*. 2019.

[2] *Cluster Networking*. The Linux Foundation. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/ (visited on 02/20/2021).

[3] *Docker overview*. Docker Inc. URL: https://docs.docker.com/get-started/overview/ (visited on 02/20/2021).

[4] *Home Assistant Core Architecture*. Home Assistant, Inc. URL: https://developers.home-assistant.io/docs/architecture/core/ (visited on 02/20/2021).

[5] *Home Assistant Installation*. Home Assistant, Inc. URL: https://www.home-assistant.io/installation/ (visited on 02/20/2021).

[6] *How AWS IoT Greengrass works*. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/how-it-works.html (visited on 02/20/2021).

[7] *How AWS IoT works*. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html (visited on 02/20/2021).

[8] *Ingress Load Balancer Kubernetes Definition*. Avi Networks. URL: https://avinetworks.com/glossary/ingress-load-balancer-for-kubernetes/ (visited on 03/27/2021).

[9] *Introduction to Kubernetes Workloads*. Rancher. URL: https://www.suse.com/c/rancher_blog/introduction-to-kubernetes-workloads/ (visited on 02/20/2021).

[10] *Introduction to Kubernetes architecture*. Red Hat, Inc. URL: https://www.redhat.com/en/topics/containers/kubernetes-architecture/ (visited on 03/27/2021).

[11] *IoT Concepts and Azure IoT Hub*. Microsoft. URL: https://docs.microsoft.com/en-us/azure/iot-hub/iot-concepts-and-iot-hub (visited on 02/20/2021).

[12] Bukhary Ikhwan Ismail et al. "Evaluation of Docker as Edge computing platform". In: *2015 IEEE Conference on Open Systems (ICOS)*. 2015, pp. 130–135. DOI: 10.1109/ICOS.2015.7377291.

[13] Li Ju, Prashant Singh, and Salman Toor. "Proactive autoscaling for edge computing systems with kubernetes". In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. ACM,

2021. DOI: 10.1145/3492323.3495588. URL: https://doi.org/10.1145%2F3492323.3495588.

[14] *K3S Architecture*. Rancher. URL: https://rancher.com/docs/k3s/latest/en/architecture/ (visited on 02/20/2021).

[15] *K3s - Lightweight Kubernetes*. Rancher. URL: https://rancher.com/docs/k3s/latest/en/ (visited on 03/27/2021).

[16] *K3s Installation Requirements*. Rancher. URL: https://rancher.com/docs/k3s/latest/en/installation/installation-requirements/ (visited on 03/27/2021).

[17] Aykut Kanyilmaz and Aydin Cetin. "Fog Based Architecture Design for IoT with Private Nodes: A Smart Home Application". In: *2019 7th International Istanbul Smart Grids and Cities Congress and Fair (ICSG)*. 2019, pp. 194–198. DOI: 10.1109/SGCF.2019.8782400.

[18] Endah Kristiani et al. ""Implementation of an Edge Computing Architecture Using OpenStack and Kubernetes"". In: *Information Science and Applications 2018*. Ed. by Kuinam J. "Kim and Nakhoon" Baek. Singapore: Springer Singapore, 2019, pp. 675–685. ISBN: 978-981-13-1056-0.

[19] *Kube-controller-manager*. The Linux Foundation. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/ (visited on 02/20/2021).

[20] *Kubernetes Components*. The Kubernetes Authors. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 04/05/2021).

[21] *Kubernetes Pods*. The Linux Foundation. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 02/20/2021).

[22] *Kubernetes Service*. The Linux Foundation. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 02/20/2021).

[23] *Kubernetes Service*. IBM. URL: https://www.ibm.com/docs/en/cloud-private/3.1.1?topic=networking-kubernetes-service-types (visited on 02/20/2021).

[24] *Kubernetes Workloads*. The Linux Foundation. URL: https://kubernetes.io/docs/concepts/workloads/ (visited on 02/20/2021).

[25] Kedar Vijay Kulkarni. *A brief overview of the Container Network Interface (CNI) in Kubernetes*. Red Hat, Inc. URL: https://www.redhat.com/sysadmin/cni-kubernetes (visited on 02/20/2021).

[26] Eric W. Biederman Michael Kerrisk. *namespaces(7) — Linux manual page*. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 02/20/2021).

[27] *MicroK8s vs K3s vs minikube*. Canonical Ltd. URL: https://microk8s.io/compare (visited on 04/03/2021).

[28] *MicroK8s*. Canonical Ltd. URL: https://microk8s.io (visited on 04/03/2021).

[29] *Minikube start*. The Kubernetes Authors. URL: https://minikube.sigs.k8s.io/docs/start/ (visited on 03/27/2021).

[30] Le Hoang Phuc, Linh-An Phan, and Taehong Kim. "Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure". In: *IEEE Access* 10 (2022), pp. 18966–18977. DOI: 10.1109/ACCESS.2022.3150867.

[31] Domenico Rotondi Roberto Minerva Abyi Biru. *Towards a definition of the Internet of Things (IoT)*. IEEE. URL: https://iot.ieee.org/images/files/pdf/IEEE_IoT_Towards_Definition_Internet_of_Things_Revision1_27MAY15.pdf (visited on 02/20/2021).

[32] Mahadev Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[33] Mahadev Satyanarayanan et al. "The Case for VM-Based Cloudlets in Mobile Computing". In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23. DOI: 10.1109/MPRV.2009.82.

[34] Michael Schiefer. "Smart Home Definition and Security Threats". In: *2015 Ninth International Conference on IT Security Incident Management IT Forensics*. 2015, pp. 114–118. DOI: 10.1109/IMF.2015.17.

[35] Michael Kerrisk Serge Hallyn. *cgroups(7) — Linux manual page*. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 02/20/2021).

[36] *Services, Load Balancing, and Networking*. The Linux Foundation. URL: https://kubernetes.io/docs/concepts/services-networking/ (visited on 02/20/2021).

[37] Weisong Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[38] *Snaps in Ubuntu Core*. Canonical Ltd. URL: https://ubuntu.com/core/docs/snaps-in-ubuntu-core (visited on 04/03/2021).

[39] Jordan Teicher. *The little-known story of the first IoT device*. IBM. URL: https://www.ibm.com/blogs/industries/little-known-story-first-iot-device/ (visited on 02/20/2021).

[40] *Understand the Azure IoT Edge runtime and its architecture*. Microsoft. URL: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-runtime?view=iotedge-2020-11 (visited on 02/20/2021).

[41] *Understanding kubernetes networking: pods and services*. Devopstales. URL: https://devopstales.github.io/kubernetes/kubernetes-networking-1/ (visited on 04/03/2021).

[42] Blesson Varghese et al. "Challenges and Opportunities in Edge Computing". In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. 2016, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18.

[43] *What Is Edge Computing?* Intel Corporation. URL: https://www.intel.com/content/www/us/en/edge-computing/what-is-edge-computing.html (visited on 02/20/2021).

[44] *What is AWS IoT Greengrass?* Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/greengrass/v1/developerguide/what-is-gg.html (visited on 02/20/2021).

[45] *What is Azure IoT Edge*. Microsoft. URL: https://docs.microsoft.com/en-us/azure/iot-edge/about-iot-edge?view=iotedge-2020-11 (visited on 02/20/2021).

[46] *What is Docker?* Microsoft. URL: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined (visited on 04/03/2021).